



J A D E™

Replication Framework

User's Guide

VERSION 6.3



Copyright © 2009
Jade Software Corporation Limited
All rights reserved

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2009 Jade Software Corporation Limited.
All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

Before You Begin	vi
Who Should Read this Guide.....	vi
What Do You Need to Know	vi
What's Included in this Guide	vii
Conventions	vii
Related Documentation	viii
Chapter 1 JADE Replication Framework Concepts	9
Overview	10
Framework Packages	10
Server and Clients.....	11
Replication Agents.....	11
Registering Classes and Properties for Replication	12
Flattening the Class Structure of the Server on the Client	13
Client Groups.....	13
Transaction Tracing	14
Object Identity Mapping	14
Loading Objects from the Server.....	15
Captured Transaction Queues on the Server	15
Captured Transaction Queues on the Client.....	16
Applying Captured Transactions	16
User Messages	17
Selective Replication by Transaction Targeting	17
Limitations	17
Chapter 2 Replication Package	19
Overview	19
ReplicationPackage Interfaces	19
JIProgressCallback Interface	20
JIProgressCallback Methods	20
JIReplicable Interface	21
JIReplicable Methods	22
JIReplicateControl Interface	23
JIReplicateControl Methods	23

Chapter 2 Replication Package, continued

JITransfer Interface.....	26
JITransfer Methods	27
ReplicationPackage Classes	28
ReplicateAgent Class.....	28
ReplicateAgent Constants	28
ReplicateAgent Methods.....	29
ReplicateApplyObject Class.....	41
ReplicateApplyObject Methods	42
ReplicateApplyTransaction Class	45
ReplicateApplyTransaction Methods	46
ReplicateClientAgent Class.....	46
ReplicateClientAgent Methods.....	47
ReplicateConflictEntry Class	48
ReplicateConflictEntry Methods	48
ReplicateConflictObject Class	50
ReplicateConflictObject Constants	51
ReplicateConflictObject Properties	51
ReplicateFramework Class.....	54
ReplicateFramework Methods	54
ReplicateLoadClass Class	56
ReplicateLoadClass Methods.....	56
ReplicateMapClass Class.....	57
ReplicateMapClass Methods	57
ReplicateQueue Class	58
ReplicateQueue Methods	58
ReplicateQueueEntry Class	59
ReplicateQueueEntry Methods	59
ReplicateQueueTransaction Class	61
ReplicateQueueTransaction Methods	61
ReplicateServerAgent Class	63
ReplicateServerAgent Methods	63

Chapter 3 Replication Basic Comms Package

Overview	72
Replication Basic Comms Package Interfaces.....	73
JITCPClientCallback Interface.....	73
JITCPClientCallback Methods.....	73
JITCPServerCallback Interface.....	74
JITCPServerCallback Methods	74
ReplicationBasicCommsPackage Classes.....	75
JRFTCPClient Class.....	75
JRFTCPClient Methods.....	75
JRFTCPServer Class.....	77
JRFTCPServer Methods.....	78
[JRFCOMMS] Section of the JADE Initialization File	80
HostName.....	80

JADE Replication Framework

User's Guide

Chapter 3	Replication Basic Comms Package, continued	
	Port.....	80
	Timeout	80
	MinimumInUse	80
	MaximumInUse.....	80
	LogFileNames.....	81
	Sample [JRFCOMMS] Section.....	81
Chapter 4	Using the JADE Replication Framework	82
	Overview	82
	Coding Examples Using the JADE Replication Framework	83
	Registering a Client.....	83
	Registering the Server.....	84
	Registering Classes and Properties for Replication of Transactions on the Server	84
	Registering Classes and Properties for Replication of Transactions on a Client.....	84
	Capturing a Transaction for Replication.....	85
	Registering a Transfer Receiver on a Client.....	85
	Replicating Transactions from the Server to a Client.....	86
	Replicating Transactions from a Client to the Server.....	86
	Registering Classes on the Server for an Initial Load of Objects to a Client	86
	Requesting an Initial Load of Objects from the Server to a Client	86
Index		88

Before You Begin

The *JADE Replication Framework User's Guide* is intended as the main source of information when you are using the JADE Replication Framework to replicate selective changes between participating JADE systems; for example, physically different but logically similar JADE systems.

Who Should Read this Guide

The main audience for the *JADE Replication Framework User's Guide* is expected to be:

- Developers of JADE environments that require the definition, replication, and resynchronization of data from a server-side system to any number of offline clients, including Compact JADE single user node clients; for example, a Personal Digital Assistant (PDA).
- Runtime users of deployed JADE applications; for example, mobile devices or laptops that are not connected to the server.

What Do You Need to Know

To use the JADE Replication Framework, you should be familiar with:

- Concepts of JADE systems
- The JADE database

While familiarity with the JADE product would be beneficial when using the JADE Replication Framework, this guide has been produced with end-users in mind, who may not be familiar with JADE and who also may not have any computing experience beyond that of being an end-user of business applications.

What's Included in this Guide

The *JADE Replication Framework User's Guide* has four chapters.

Chapter 1	Provides an overview of JADE Replication Framework concepts and terminology
Chapter 2	Provides instructions for using the classes, methods, and interfaces contained in the <code>ReplicationPackage</code> , which is part of the JADE Replication Framework user Application Programming Interface (API)
Chapter 3	Provides instructions for using the classes, methods, and interfaces contained in the <code>ReplicationBasicCommsPackage</code> , which is part of the JADE Replication Framework user API
Chapter 4	Provides examples of the use of the JADE Replication Framework to achieve the replication and resynchronization of data from a server-side system with offline client systems

Conventions

The *JADE Replication Framework User's Guide* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (>)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type report , type all the bold characters exactly as they are printed. File, class, primitive type, and property names, menu commands, and screen controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.
<i>Italic</i>	Parameter values or placeholders for information that must be provided; for example, if instructed to enter <i>class-name</i> , type the actual name of the class instead of the word or words shown in italic type. Italic type also signals a new term. An explanation accompanies the italicized type. Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the " ReplicationPackage Interfaces " cross-reference to display that topic.
Bracket symbols ([])	Indicate optional items.
Vertical bar ()	Separates alternative items.
Monospaced font	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
SMALL CAPITALS	Keyboard keys.

Key combinations and key sequences appear as follows.

Convention	Description
KEY1+KEY2	Press and hold down the first key while pressing the second key. For example, “press SHIFT+F2” means to press and hold down the SHIFT key and press the F2 key. Then release both keys.
KEY1,KEY2	Press and release the first key, and then press and release the second key. For example, “press ALT+F,X” means to hold down the ALT key, press the F key, and then release both keys before pressing and releasing the X key.

In this document, the term Microsoft *Windows* refers to Windows 2003 Server, Windows Vista, Windows XP, Windows 2000, or Windows CE. When there are differences between the versions of Microsoft Windows, the specific version of Microsoft Windows is stated. This also applies to Linux, which is a specific version of UNIX developed by SUSE or Red Hat. The term *UNIX* is used when an issue is generic to all versions of UNIX and the term *Linux* is used if the issue is specific to the SUSE or Red Hat implementation of UNIX.

Related Documentation

Other documents that are referred to in this document, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

Title	Related to...
JADE Database Administration Guide	Administering JADE databases
JADE Development Environment Administration Guide	Administering JADE development environments
JADE Development Environment User's Guide	Using the JADE development environment
JADE Encyclopaedia of Classes	System classes (Volumes 1 and 2), Window classes (Volume 3)
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Installation and Configuration Guide	Installing and configuring JADE
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Java Developer's Reference	Using the Java framework to develop JADE applications
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)
JADE Object Manager Guide	JADE Object Manager administration
JADE Report Writer User's Guide	Using the JADE Report Writer to develop and run reports
JADE Thin Client Guide	Administering JADE thin client environments
JADE Web Application Guide	Implementing, monitoring, and configuring Web applications

Chapter 1

JADE Replication Framework Concepts

This chapter contains an overview of the components of the JADE Replication Framework and covers the following topics.

- [Overview](#)
- [Framework Packages](#)
- [Server and Clients](#)
- [Replication Agents](#)
- [Registering Classes and Properties for Replication](#)
- [Flattening the Class Structure of the Server on the Client](#)
- [Client Groups](#)
- [Transaction Tracing](#)
- [Object Identity Mapping](#)
- [Loading Objects from the Server](#)
- [Captured Transaction Queues on the Server](#)
- [Captured Transaction Queues on the Client](#)
- [Applying Captured Transactions](#)
- [User Messages](#)
- [Selective Replication by Transaction Targeting](#)
- [Limitations](#)

Overview

The JADE Replication Framework is intended to handle as much as possible of the work required to replicate selective changes between participating JADE systems. It handles the activity common to all such activity, with user hooks for application-specific processing.

The base functionality is selective replication of object activity between physically different but logically similar JADE systems. These systems may be identical but are not required to be so. If they are not identical, a user-defined mapping of classes and properties involved on each side of the replication is required.

The Replication Framework includes object synchronization and conflict resolution handling.

The Replication Framework allows for environments where one side (the *server*) is a full JADE system and the other side (the *client*) is a limited functionality JADE system, typically on a mobile device such as a Personal Digital Assistant (PDA). The client would run an application in a separate schema from the server, with a limited number of classes, which have only the properties required for the application.

The JADE Replication Framework allows for:

- Initial bulk loading to the client
- Subsequent replication of server changes to the client
- Replication of client changes to the server
- Requests for data from the client

Replication can be one-way, where updates are expected on the server only (that is, the client is read-only), or two-way, where both server and client can update objects.

Framework Packages

The JADE Replication Framework is implemented through the [ReplicationPackage](#) export package in the **JadeReplicationSchema** schema, which is released as a separate encrypted source schema, in the same way as the JADE Report Writer.

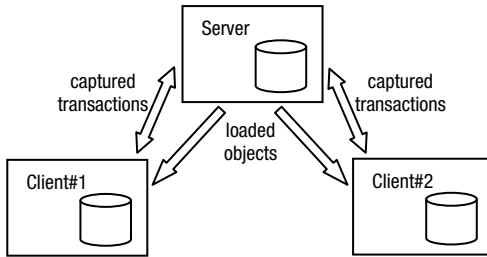
For details about the classes, methods, and interfaces provided in the replication package of the JADE Replication Framework, see Chapter 2, “[Replication Package](#)”.

Note The **JadeReplicationSchema** is not automatically loaded when you install JADE. It is contained in the **frameworks** folder of the JADE install directory.

An optional communications layer is provided as the [ReplicationBasicCommsPackage](#) export package in the **JadeReplicationSchema** schema. This package provides the basic functionality required to connect client and server systems using Transmission Control Protocol/Internet Protocol (TCP/IP), to simplify initial development using the JADE Replication Framework. For details about the classes, methods, and interfaces provided in the basic communication package of the JADE Replication Framework, see Chapter 3, “[Replication Basic Comms Package](#)”.

Server and Clients

One of the participating JADE systems in the replication is registered as a *server*. The other participants are registered as *clients*.



Server and clients are separate JADE systems that connect for replication using TCPIP or Web services

The server manages transaction and object identifiers (oids). The database on the server is logically the master for user data.

The client systems can use the same schema as the server, but the data is typically a subset of the server data, partitioned by some criteria. For example, if the server has a complete set of information for all customers (instances of the **Customer** class), each client could have a subset of customers for a specific region.

Data changes on the server are assumed to be of potential interest to a number of clients but data changes on clients of interest primarily to the server. Client changes that affect other clients are channeled through the server. Replication activity is initiated by clients.

The **systemName** parameter of the **ReplicateFramework** class **registerAsClient** and **registerAsServer** methods is used to distinguish persistent replication data between multiple JADE systems (independent schemas) in the same development or runtime environments. The value of the **systemName** parameter is typically a schema name.

Replication Agents

All interaction with the framework is through an instance of the **ReplicateClientAgent** class for a client or the **ReplicateServerAgent** class for the server.

When the application initializes, an instance of the **ReplicateFramework** class is created and its **registerAsClient** or **registerAsServer** method used to return a transient instance of the **ReplicateClientAgent** or **ReplicateServerAgent** classes, respectively. The framework assumes that there is an *agent* instance for each running copy of the user application. Typically, the agent instance can be accessed from user code, using a reference on the **app** environmental object.

After framework initialization, the **ReplicateLoadClass** and **ReplicateMapClass** methods are called for the newly created agent instance, to establish class and property mappings.

The agent instance is also used to *register* a receiver for callback messages that are sent during replication. This processing is typically carried out in the **initialize** method of the application.

Values set from methods in the **ReplicateAgent** class (for example, the **setTracingMode** and **setTransferPerTransaction** methods) are held on framework internal transient objects associated with the **ReplicateAgent** instance. If a copy of an application therefore has a number of agents defined through multiple calls to the **ReplicateFramework** class **registerAsClient** or **registerAsServer** method, each agent can have different values. The processing of an *agent* method depends on the specific agent instance for which the method is called.

As captured user transactions and queued input transactions are held persistently, they are available to all agents.

Registering Classes and Properties for Replication

The classes and properties for which changes are to be automatically captured by the framework must be declared on clients and servers. The class must implement the JADE Replication Framework **JIREplicable** interface. As this implementation can be inherited by subclasses, a convenient approach is to define all replicable classes as subclasses of a single class that implements the interface.

Each property whose change is to be captured and replicated must be registered. This is necessary because the framework uses transaction tracing functionality from the **RootSchema**, which records changes to registered properties only. For details about transaction tracing, see “Transaction Tracing”, in Chapter 24 of the *JADE Developer's Reference*.

Note When you use the JADE Replication Framework, transaction tracing functionality is automatically invoked without having to call methods on the **JadeTransactionTrace** class directly.

The **ReplicateAgent** class **replicateProperty** and **replicateAllProperties** methods register properties for tracing. Properties registered using the **ReplicateLoadClass** class **registerProperty** and **registerAllPropertiesUpTo** methods are used by the server agent **replicateLoad** method, which extracts data for registered properties of the class specified in the **obj** parameter.

Class and property names for equivalent JADE entities do not have to be the same. The JADE Replication Framework saves and applies object data changes by class and property name, and uses the same name for the client and server, by default. If the class or property names are different, the names must be registered on the server before any replication activity is executed. This is typically done in the application **initialize** method. Use the **ReplicateServerAgent** class **getMapClass** method and **ReplicateMapClass** class **mapProperty** method for this purpose.

The **getMapClass** method has a **systemName** parameter (usually the name of the schema), which allows for a number of different client database definitions.

These methods take class or property parameters, which can be coded using class and property names with properties using the **::** metaobject scope operator, so that changes to class or property names are automatically changed in these usages in logic, as shown in the following code fragment examples.

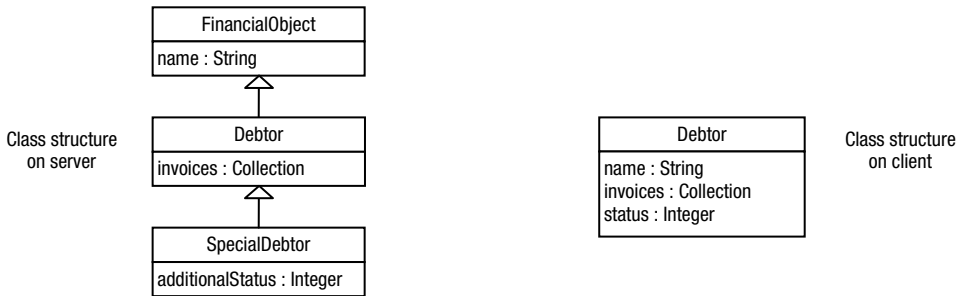
```
clientAgent.replicateProperty(JobStatusCode::codeValue);  
mapClass := serverAgent.getMapClass(JobTypeCode,systemName,"JobStatusCode");  
mapClass.mapProperty(JobTypeCode::code, "codeValue");
```

Note The mapped non-local names are literal values that must be maintained manually.

Flattening the Class Structure of the Server on the Client

When object changes are replicated both ways between the client and server, there must be a one-to-one mapping between each class and property involved, so that each individual property change is replicated uniquely. However, if the replication is one-way from the server to the client, you can map multiple server classes to a single client class, where the client schema is a simplification of the server schema.

The following example shows a server class structure where properties in a hierarchy of classes on the server are mapped to a single class on the client.



In this example, the object changes are applied only from the server to the client, and can be mapped as follows.

```
mapClass := serverAgent.getMapClass(FinancialObject, systemName, "Debtor");
mapClass.mapProperty(FinancialObject::name, "name");
mapClass := serverAgent.getMapClass(Debtor, systemName, "Debtor");
mapClass.mapProperty(Debtor::invoices, "invoices");
mapClass := serverAgent.getMapClass(SpecialDebtor, systemName, "Debtor");
mapClass.mapProperty(SpecialDebtor::additionalStatus, "status");
```

A change to an instance of any one of the three server classes is replicated to the single **Debtor** class on the client.

Client Groups

Client groups are an optional feature that enables a group of clients to be treated by certain operations as a single entity. For example, you could treat a number of client systems in the same geographical or logical region as a group so that they would receive only the changes that relate to their region.

If clients are declared to be in the same client group, object changes can be targeted at this group identifier instead of specifying the individual clients each time.

The following methods that take a **clientId** parameter check for a client group identifier value as well as a client identifier value.

- **ReplicateAgent** class **addUserMessage** method
- **ReplicateAgent** class **setTransactionTargetClients** method
- **ReplicateServerAgent** class **beginReplicateLoad** method
- **JIREplicable** interface **replicateToNominatedClients** method

For each method, the code checks that the specified client identifier is a valid client. If it is not, that it is a valid group, and if it is not a valid group, it raises an exception. Consequently every group identifier and client identifier must be unique within the set of all group identifiers and client identifiers.

Although you can set up client groups on a client, the main usage is on the server. Client groups on a client support transactions that explicitly target other clients instead of relying on the server for the targeting.

Transaction Tracing

The JADE Replication Framework uses the transaction tracing functionality provided in the **RootSchema** to automatically capture changes to registered properties of objects of registered classes and to persist these changes in a queue to be transferred to the relevant participant and applied there.

As tracing is off by default, you must enable it by using the **enableTracing** method of the **ReplicateAgent** class. Once enabled, it stays enabled for the current JADE process until it is disabled again, by the calling the **disableTracing** method. In this way, you can control the transactions that are replicated, even though the changed classes and properties have been registered for replication.

Notes Transaction tracing and traced property registration applies to a single JADE process.

Disabling property-level tracing in user logic by calling the **enableTransactionTracing** method of the **Property** class with the **enable** parameter set to **false** stops tracing in the same way as the **ReplicateAgent** class **unreplicateProperty** method. In addition, the **Process** class **stopTransactionTrace** method in user logic stops object changes from being replicated.

Object Identity Mapping

The JADE Replication Framework keeps track of equivalent objects on the client and server, using one of the following mechanisms.

- In an internal oid mapping, the server maintains a persistent mapping for each replicated object on each client. The assumption is that class numbers and instance identifiers of equivalent objects are different on the server and each client.

This is the default mechanism.

- In the specialized case where the client is set up as a full copy of the server, the schema definitions are identical and all objects on the client are identical to objects on the server. The client database can be a subset of the server database (for example, based on a specific geographical or logical region), but the objects that do exist on the client have the same object identifiers (oids) as the corresponding objects on the server. The oid mapping feature of the JADE Replication Framework is not required for these existing objects.

However, oid mapping is used for subsequent replication between the client and server. When the option is enabled by calling the server agent **setInitialOidEqualityMode** method, client changes applied on the server use the client oid as the server oid, if no oid map entry is found for the object.

- For user-defined unique identifiers, the **JJReplicable** interface **getUniqueId** method returns a user-defined unique identifier for a replicable object and the **JJReplicateControl** interface **getObjectForId** method returns an object for a specified identifier value.

Note This mechanism is not yet implemented.

All participating JADE systems using the JADE Replication Framework must use the same mechanism.

Loading Objects from the Server

The JADE Replication Framework tracks changes to objects that:

- Exist on both the client and server
- Are created on the server and the create operation is replicated to the client
- Are created on the client and the create operation is replicated to the server

Changes to objects that exist on one side only and are unknown to the other side are not handled. In addition, the transaction tracing feature alone does not replicate existing objects on the server as new objects on the client.

The concept of copying existing server objects to the client for subsequent replication of their changes is known as a *load* in the framework. The server agent **replicateLoad** method, together with the associated **beginReplicateLoad**, **endReplicateLoad**, and **replicateLoadTransaction** methods, writes an entry to an outgoing queue for existing server objects. This entry is applied on the client as a *create* object, to populate the client with replicable copies of server objects.

An example of the loading scenario is a simple mobile client that receives objects representing a *to do* list from the server. On the client, the activities are performed, the objects updated and replicated back to the server, and a new set of objects is requested.

The **replicateLoad** method copies existing server instances to the client, and regular transaction tracing on the client automatically captures activity as the *to do* list is updated. When the client queue is sent back to the server, you can delete the loaded objects, to make room for the next batch.

These deletions should not be replicated back to the server because the corresponding server objects are to be kept, but the server oid map should be updated to reflect that the objects no longer exist on the client. You should perform the object deletions from a traced transaction with tracing mode set, using the agent **setTracingMode** method set to the value of the **ReplicateAgent** class **TracingModeLocalCleanup** constant for this transaction only and then set to **TracingModeFull** for subsequent regular traced transactions.

Captured Transaction Queues on the Server

The server has the following sets of outgoing queues.

- A queue for activity saved from execution of the server agent **replicateLoad** method. This queue is held separately for each target client.
- A queue that is saved from captured replicable transactions.

There is one queue for all clients, which contains additional client target information, so there appears to be separate logical queues for each client. When queued changes are replicated to a client, any content in the load queue for the client is sent first, followed by any queued changes. The assumption is that loaded objects could be required by changes, and not the other way around.

The server has one set of incoming queues, for deferred input, held separately for each client that has sent deferred changes.

All queues are persistent.

Entries in the server load queue are automatically removed when they have been replicated to the target client and the client has successfully processed them.

The server queue of captured replicable transactions is not automatically cleaned up as changes are replicated to clients. The replication uses last-sent and last-applied identifiers as pointers into this queue, so it will not resend transactions that have already been processed.

Removing old entries as the replication is done is an additional source of contention, and it is not done by default. If you require automatic removal, call the `ReplicateServerAgent` class `setInlineQueueRemovalMode` method with the value of the `mode` parameter set to `true`.

You can remove the already-sent entries in the captured replicable transactions queue by calling the `ReplicateServerAgent` class `removeSentQueueEntries` method. Call this method when replicable transactions are not being captured and replication to clients is not taking place. The method locks the queue at the start and it exits immediately, if the lock cannot be applied.

Captured Transaction Queues on the Client

The client has one outgoing queue of captured replicable transactions and it has one incoming queue for deferred input from the server.

All queues are persistent.

Applying Captured Transactions

As each captured transaction from the other side is applied on the client and server, there are a number of optional processing steps.

If there is a registered `preApply` receiver object and the `preApply` mode is set appropriately, `ReplicateApplyTransaction` and `ReplicateApplyObject` transient instances are populated for the transaction details and the `preApply` method of the `JJReplicateControl` interface is called. This enables you to inspect the incoming transaction and optionally process it fully, to override the default processing of the JADE Replication Framework. If you use this option, the method returns a value of negative one (-1) and the rest of the framework processing of the transaction is skipped.

If you do *not* use this option, the framework code replays the captured details, creating, modifying, and deleting object instances, as required. If there are changes or deletions of objects (or reference property changes of objects) that cannot be found using the oid mapping and there is a registered conflict receiver, the `resolveConflict` method of the `JJReplicateControl` interface is called with the `ReplicateConflictObject` parameter populated from the conflict details.

Similarly, if an exception is raised from the replay of transactions, the `resolveConflict` method is called and persistent error details are saved in the JADE Replication Framework. You can access these error details by calling the `getConflicts` method of the agent.

Following the replay, if there is a registered `postApply` receiver object and the post-apply mode is set appropriately, `ReplicateApplyTransaction` and `ReplicateApplyObject` transient instances are populated for the transaction details if they are not already populated and the `postApply` method of the `JIREplicateControl` interface is called. This enables additional user processing, in particular where the client is a simplified version of the server and the corresponding server objects have additional references to set or collections to populate.

User Messages

User messages are additional messages inserted into the outgoing transaction capture queues at the time of calling the agent `addUserMessage` method. They are inserted and processed as separate transactions from the other entries in the queue. The intent of these messages is to trigger specific processing on the receiving end.

When a user message is processed by the receiving end, the object registered by the `registerUserMessageReceiver` method of the agent is called with details from the message in the `userMessage` method of the `JIREplicateControl` interface. If there is no registered receiver, the message is ignored.

Selective Replication by Transaction Targeting

On the client, captured transactions are fully targeted to the server, by default. The `JIREplicable` interface `replicateToClient` and `replicateToNominatedClients` methods are not called on the client. If it is known that a client transaction is to be sent on to one or more other clients, you can use the `setTransactionTargetClients` method to specify these clients.

On the server, captured transactions are expected to be sent to selected clients and object activity in the same transaction can be sent to different clients for different objects. The default behavior is for the `replicateToClient` or `replicateToNominatedClients` method to be called, depending on the value of the `mode` parameter passed to the `setCallReplToNominatedClients` method of the server agent. The `replicateToClient` or `replicateToNominatedClients` methods are called with the transaction capture, to determine the clients to which each object change is sent.

You can use an alternative mechanism when it is known that the whole transaction is to be sent to one or more clients, by calling the `setTransactionTargetClients` method of the agent to specify the clients. If you call this method, the `replicateToClient` or `replicateToNominatedClients` methods are not called for objects in the transaction.

Limitations

The transaction tracing functionality from the `RootSchema` captures the setting of property values, and not necessarily changes to property values. If update methods are coded as a series of unconditional setting of property values, each property setting is captured and replicated, regardless of whether the object property value had actually changed.

As there is no transaction tracing of primitive array updates, there is no capture of these for replication, although the `replicateLoad` method copies primitive array content. Additions and removals from reference array properties are captured and applied, but not array changes with explicit index values.

Only the manual side of a property value change with inverses is captured to the replication queues. The assumption is that a receiving schema has a matching definition for update mode for an inverse reference (that is, whether it is manual, automatic, or the manual/automatic combination), and that applying the manual capture results in the required automatic processing on the receiver.

A purge of a non-inversed collection is not replicated.

Many-to-many reference property changes are not replicated.

This chapter covers the following topics.

- [Overview](#)
- [ReplicationPackage Interfaces](#)
- [ReplicationPackage Classes](#)

Overview

The core JADE Replication Framework functionality is contained in the **ReplicationPackage** package exported from the **JadeReplicationSchema** schema into your user-defined schema.

You can import the **ReplicationBasicCommsPackage** package from the **JadeReplicationSchema** schema if you want replication messages to be sent between participating systems using TCP/IP. For details, see Chapter 3, “[Replication Basic Comms Package](#)”.

This chapter is a reference to the Application Programming Interface (API) provided by the **ReplicationPackage** package. For examples of the use of the classes, methods, and interfaces in this package, see Chapter 4, “[Using the JADE Replication Framework](#)”.

Note The entities documented in this chapter are those that are available when you import the **ReplicationPackage** package into a user-defined schema (that is, they do *not* include all entities in the **JadeReplicationSchema** itself, many of which are system-specific).

ReplicationPackage Interfaces

The **ReplicationPackage** package provides the interfaces summarized in the following table.

Interface	Provides callbacks ...
JIProgressCallback	To report progress during replication
JIReplicable	To target clients to receive replication changes for a persistent user class
JIReplicateControl	For events that occur during the replication process and if a conflict arises
JITransfer	To transfer replication data between client and server

For details, see the following subsections.

Note A de-serialized transaction does not have to be applied immediately.

The parameters for this method are listed in the following table.

Parameter	Description
fromId	Identifier of the client or server from which the message came
tranId	Transaction identifier of the original transaction
sequence	Sequence within the transaction, which is usually one (1)
size	Estimate of the transaction size, in bytes

serializedTransaction

Signature `serializedTransaction(targetId: String;
 tranId: Integer64;
 sequence: Integer;
 size: Integer);`

The **serializedTransaction** method of the **JIPProgressCallback** interface is called for a registered progress receiver when a transaction is serialized for transmission.

The parameters for this method are listed in the following table.

Parameter	Description
targetId	Identifier of the client or server from which the message came
tranId	Transaction identifier of the original transaction
sequence	Sequence within the transaction, which is usually one (1)
size	Estimate of the transaction size, in bytes

JIREplicable Interface

The **JIREplicable** interface is implemented by a user persistent class that is to be replicated.

On the server *and* clients, you must register the classes and properties for which changes are to be automatically captured by the JADE Replication Framework.

The class must implement the **JIREplicable** interface of the framework.

Hint As implementation is inherited from superclasses, a convenient approach for new systems is to define all replicable classes as subclasses of a single class that implements the interface.

Register each property whose change is to be captured and replicated. For details, see “[Registering Classes and Properties for Replication](#)”, in Chapter 1.

For details about the methods defined in the **JIREplicable** interface, see “[JIREplicable Methods](#)”, in the following subsection.

JIREplicable Methods

The methods defined in the **JIREplicable** interface are summarized in the following table.

Method	Description
getUniqueld	Not implemented
replicateToClient	Returns true if an update to the receiver is to be replicated to the specified client
replicateToNominatedClients	Specifies the clients targeted for replication of an update to the receiver

getUniqueld

Signature `getUniqueId(): String;`

The **getUniqueId** method of the **JIREplicable** interface is used when objects are identified by user-supplied unique identifiers.

This feature is not yet implemented.

replicateToClient

Signature `replicateToClient(clientId: String): Boolean;`

The **replicateToClient** method of the **JIREplicable** interface determines whether an update to the receiver is replicated to the client specified by the **clientId** parameter when the transaction commits. The update is creating or deleting the receiver, or modifying a property registered for replication.

Your implementation of this method on the server should return **true** for the change to be replicated; otherwise **false**. The method is called for each client registered on the server by using the **registerClient** method of the **ReplicateServerAgent** class.

The **replicateToClient** method is the default way of targeting clients to receive replication changes. It is not called if the **ReplicateServerAgent** class **setCallReplToNominatedClients** method has been called with the **mode** parameter set to **true**, in which case the **replicateToNominatedClients** method is called once, to allow targeted clients to be specified instead of individual **replicateToClient** calls for each registered client.

If you call the **ReplicateAgent** class **setTransactionTargetClients** method, callbacks to the **replicateToClient** or **replicateToNominatedClients** method do not occur.

replicateToNominatedClients

Signature `replicateToNominatedClients(ids: StringArray input): Boolean;`

The **replicateToNominatedClients** method of the **JIREplicable** interface specifies clients that are targeted for replication when a transaction that updates the receiver commits. The update is creating or deleting the receiver, or modifying a property registered for replication.

To replicate the update to all clients registered on the server using the **registerClient** method of the **ReplicateServerAgent** class, return **true**. To selectively replicate the update, return **false** and add the client identifiers of targeted clients to the value of the **ids** parameter.

For the **replicateToNominatedClients** method to be called in place of the default the **replicateToClient** method, call the **ReplicateServerAgent** class **setCallReplToNominatedClients** method with the **mode** parameter set to **true**.

User-defined unique identifiers provide an alternative way to map object identifiers of corresponding objects on the server and clients, which are maintained by the JADE Replication Framework. Before you call this method, you must register the receiver by calling the [registerIdController](#) method of the [ReplicateAgent](#) class.

beginDBTransaction

Signature `beginDBTransaction();`

The **beginDBTransaction** method of the [JIReplicateControl](#) interface enables additional processing when a [beginTransaction](#) instruction is executed within the JADE Replication Framework. Before you call this method, you must register the receiver by using the [ReplicateAgent](#) class [registerTransactionReceiver](#) method.

The JADE Replication Framework calls the **beginTransaction** method when a **beginTransaction** action is carried out for a user update (but not an internal update). Because the framework has a default implementation of this method, registration is needed only if additional user processing is required.

commitDBTransaction

Signature `commitDBTransaction();`

The **commitDBTransaction** method of the [JIReplicateControl](#) interface enables additional processing when a [commitTransaction](#) instruction is executed within the framework. Before this method is called, the receiver must be registered using the [registerTransactionReceiver](#) method of the [ReplicateAgent](#) class.

The JADE Replication Framework calls the **commitTransaction** method when a **commitTransaction** action is carried out for a user update (but not an internal update). Because the framework has a default implementation of this method, registration is needed only if additional user processing is required.

getObjectForId

Signature `getObjectForId(id: String): Object;`

The **getObjectForId** method of the [JIReplicateControl](#) interface returns the replicated object corresponding to the identifier specified by the **id** parameter. User-defined unique identifiers provide an alternative way to map object identifiers of corresponding objects on the server and clients, which are maintained by the JADE Replication Framework.

Before you call this method is called, you must register the receiver by using the [ReplicateAgent](#) class [registerIdController](#) method.

Note This functionality is not yet implemented.

postApply

Signature `postApply(transaction: ReplicateApplyTransaction);`

The **postApply** method of the [JIReplicateControl](#) interface enables additional processing of the transaction specified by the **transaction** parameter after it is applied. Before you call this method, you must register the receiver by using the [ReplicateAgent](#) class [registerPostapplyReceiver](#) method.

In addition, set the apply mode to **ApplyModeAll** or **ApplyModeAllButLoad**, using the **ReplicateAgent** class **setPostapplyMode** method.

Post-apply processing is particularly useful when objects on the client are simplified versions of corresponding objects on the server. In this case, you often need to set additional references and add objects to collections on the server.

preApply

Signature `preApply(transaction: ReplicateApplyTransaction): Integer;`

The **preApply** method of the **JIREplicateControl** interface enables processing of the transaction specified by the **transaction** parameter before the transaction is applied. Before you call this method, you must register the receiver by using the **ReplicateAgent** class **registerPreapplyReceiver** method.

In addition, set the apply mode to **ApplyModeAll** (the default value) or to **ApplyModeAllButLoad** by using the **ReplicateAgent** class **setPreapplyMode** method.

If your implementation of the **preApply** method does the actual updating, it must return a value of negative one (-1), to skip the automatic updating by the JADE Replication Framework and the calling of the **postApply** method, if implemented.

resolveConflict

Signature `resolveConflict(details: ReplicateConflictObject): Integer;`

The **resolveConflict** method of the **JIREplicateControl** interface enables user handling of conflicts and exceptions detected when replication changes are applied. Before you call this method, you must register the receiver by using the **ReplicateAgent** class **registerConflictReceiver** method.

The **resolveConflict** method returns one of the **ReplicateConflictObject** class constant values listed in the following table.

Class Constant	Integer Value	Description
<code>ConflictActionContinue</code>	1	Sets the correctedObject reference to the value to be used and continues.
<code>ConflictActionDropItem</code>	2	Processes the remaining items after dropping this item; for example, the deletion of an object that is already deleted (if businesses rules allow it).
<code>ConflictActionDropTransaction</code>	3	Processes the remaining transactions after dropping this transaction.
<code>ConflictActionRetry</code>	5	Retries once, to apply the transaction (valid only for a resumable exception). Use this when the resolveConflict method has modified the database so that the transaction will succeed on retry; for example, removing a collection entry after a duplicate key exception. If the retry also raises an exception, this method is called again, and a return value of ConflictActionRetry is processed as a ConflictActionDropTransaction .
<code>ConflictActionStopProcessing</code>	4	Stops processing the transaction and subsequent transactions.

The JADE Replication Framework calls the **resolveConflict** method when it detects a conflict or exception. The **details** parameter, which is an instance of the **ReplicateConflictObject** class, contains information about the conflict.

If a conflict receiver is not registered and a conflict is detected, the framework raises an exception when the conflicted item is a **null** reference or an invalid object reference.

The framework also writes a persistent internal error queue entry from each conflict or exception. You can access these entries by calling the **ReplicateAgent** class **getConflicts** method and you can remove them by calling the **clearConflicts** method.

userLog

Signature `userLog(message: String);`

The **userLog** method of the **JIReplicateControl** interface enables logging of internal replication activity for development and debugging purposes. Before you call this method, register the receiver by using the **ReplicateAgent** class **registerUserLogReceiver** method.

In addition, the level of the activity must be at least that set by the **ReplicateAgent** class **setUserLogLevel** method.

The **message** parameter contains a string containing details of the replication action.

userMessage

Signature `userMessage(type: Integer;
 id: String;
 messageText: String);`

The **userMessage** method of the **JIReplicateControl** interface processes a received user message that was added by using the **ReplicateAgent** class **addUserMessage** method.

Before you call this method, you must register the receiver by using the **ReplicateAgent** class **registerUserMessageReceiver** method.

The **type** and **id** and **messageText** parameters contain the information that was supplied when the message was created by using the **addUserMessage** method.

JITransfer Interface

The **JITransfer** interface provides callbacks during the transfer of replication data between the client and server. Replication is initiated by a client, regardless of whether the replication is from a client to the server or from the server to a client. You must therefore implement the **JITransfer** interface by an instance on the client system.

Register an instance by using the **ReplicateClientAgent** class **registerTransferReceiver** method. If an instance is not registered, an exception is raised when replication is attempted.

The **ReplicationBasicCommsPackage** provides the **JRFTCPClient** class, which implements the **JITransfer** interface using the TCP/IP communications protocol. For details, see “**Replication Basic Comms Package**”, in Chapter 3.

Each method implementation must send the input message parameter content to the server by using a mechanism such as TCP/IP or a Web service, read the response string from the server, and then return this value from the method. This enables the server to identify the client method from which the message originated. For example, if you are using a Web service, a specific Web service method is called on the server, and if you are using TCP/IP, each message is prefixed with a unique identifier so that the TCP/IP receiver on the server can call the appropriate framework method.

For details about the methods defined in the **JITransfer** interface, see “**JITransfer Methods**”, in the following subsection.

ReplicationPackage Classes

The **ReplicationPackage** provides the classes summarized in the following table.

Class	Description
ReplicateAgent	Abstract superclass containing functionality common to client and server replication agents
ReplicateApplyObject	Consolidated changes to an object during a replicated transaction
ReplicateApplyTransaction	Replicated transaction being processed by preApply or postApply receivers
ReplicateClientAgent	System that is participating in replication as a client
ReplicateConflictEntry	Proxy for the internal details of conflict and exception errors raised during replication
ReplicateConflictObject	Used for the parameter object in the resolveConflict callback method
ReplicateFramework	Entry point to the framework, which is used to create the ReplicateClientAgent or ReplicateServerAgent instances that initiate replication
ReplicateLoadClass	Nominates a class on the server whose instances can be loaded to client systems
ReplicateMapClass	Nominates a class on the server whose instances can be loaded to a client system with different names for mapped classes and properties
ReplicateQueue	Proxy for the client and server queues of objects to be loaded and transactions to be applied
ReplicateQueueEntry	Proxy for the client and server object-level or property-level changes within a transaction
ReplicateQueueTransaction	Proxy for the client and server transactions for a queue
ReplicateServerAgent	A system that is participating in replication as the server

ReplicateAgent Class

The **ReplicateAgent** class is an abstract superclass containing functionality that is common to replication agents on the server and the client.

For details about the constants and methods defined in the **ReplicateAgent** class, see “[ReplicateAgent Constants](#)” and “[ReplicateAgent Methods](#)”, in the following subsections.

Inherits From: [JRFUserClasses](#)

Inherited By: [ReplicateClientAgent](#), [ReplicateServerAgent](#)

ReplicateAgent Constants

The constants defined in the **ReplicateAgent** class are summarized in the following table.

Constant	Integer Value
ApplyModeAll	0
ApplyModeAllButLoad	2
ApplyModeNone	1
File_Transfer	1
OperationCollAdd	9
OperationCollRemove	11
OperationCreate	4
OperationDelete	6

Constant	Integer Value
OperationLoadObject	98
OperationLoadProperty	99
OperationLocalOnlyDelete	100
OperationSetProperty	7
OperationUpdate	3
TracingModeFull	1
TracingModeIgnoreLocalDeletes	2
TracingModeLocalCleanup	3
TracingModeNone	0
Web_Service_Transfer	0

ReplicateAgent Methods

The methods defined in the [ReplicateAgent](#) class are summarized in the following table.

Method	Description
addClientGroup	Adds a new client group from information specified
addClientToGroup	Adds a client to the specified client group
addOidMap	Adds an oid mapping on the server for the specified object if the framework is bypassed
addUserMessage	Adds a user message to the outgoing captured transaction queue
clearConflicts	Deletes persistent instances of the ReplicateConflictObject class
disableTracing	Turns off the automatic capture of changes at commit time
doFrameworkLocks	Acquires locks required by the framework
enableTracing	Turns on the automatic capture of changes at commit time
getClientGroupDescription	Returns the description of the group with the specified id
getClientGroupIds	Populates a string array parameter with the identifiers of all groups
getClientsInGroup	Populates a string array parameter with the identifiers of all clients for the specified group
getConflicts	Populates an array with instances of the ReplicateConflictEntry class
getCurrentQueue	Sets the queue parameter as a proxy for the internal transaction capture queue
getCurrentReplicateQueueSize	Returns the number of replicated transactions in the current queue
getMaximumTransferSize	Returns the upper limit before a queue of captured transactions is segmented for transfer
getPostapplyMode	Returns an integer indicating whether the framework calls registered postApply receivers
getPreapplyMode	Returns an integer indicating whether the framework calls registered preApply receivers
getTracingMode	Returns an integer specifying the operations captured as a result of user logic updates
getTransferPerTransaction	Returns true if replication queues transfer one message for each captured transaction
getUserLogLevel	Returns the level at which internal replication activity is logged
isInReplication	Returns true if user logic is executed from processing of replicated changes
processResponse	Maintains the mapping of oids on the server after replication on the client has completed
purgeCurrentQueue	Removes all captured replication changes
registerConflictReceiver	Registers the callback receiver for resolveConflict messages during replication

Method	Description
registerIdController	Registers the receiver for addToUniquelds messages (not yet implemented)
registerPostapplyReceiver	Registers the callback receiver for postApply messages
registerPreapplyReceiver	Registers the callback receiver for preApply messages
registerProgressReceiver	Registers the callback receiver for replication and transfer progress callback messages
registerTransactionReceiver	Registers the callback receiver for abortDBTransaction , beginDBTransaction , and commitDBTransaction methods when transaction control is required
registerUserLogReceiver	Registers the callback receiver for userLog messages for a framework action
registerUserMessageReceiver	Registers the callback receiver for userMessage methods
removeClientFromGroup	Removes the client from the specified client group
removeClientGroup	Removes the specified client group
removeOidMap	Removes an oid mapping on the server for the specified object
replicateAllProperties	Registers all properties defined in a specified class and superclasses for replication
replicateProperty	Registers the specified property for replication
setMaximumTransferSize	Sets the upper limit before a queue of captured transactions is segmented for transfer
setPostapplyMode	Specifies whether postApply receivers are called for replication and for loading
setPreapplyMode	Specifies whether preApply receivers are called for replication and for loading
setTracingMode	Specifies the operations that are captured as a result of user logic updates
setTransactionTargetClients	Sets target client ids from user logic within a transaction
setTransferPerTransaction	Specifies that replication queues transfer one message for each captured transaction
setUserErrorText	Stores additional information of the error or action taken
setUserLogLevel	Sets the level at which internal replication activity is logged
unreplicateProperty	Unregisters the specified property for replication

addClientGroup

Signature `addClientGroup(id: String; description: String): Boolean;`

The **addClientGroup** method of the [ReplicateAgent](#) class adds a new client group to the current client or server, if one with this group identifier value does not exist. This method returns **true** if the group was added and **false** if the group was not added because the identifier value was **null** or there is already a group with the specified **id** value.

addClientToGroup

Signature `addClientToGroup(groupId: String; clientId: String): Boolean;`

The **addClientToGroup** method of the [ReplicateAgent](#) class adds the client with the value specified in the **clientId** parameter to the client group with the specified value of the **groupId** parameter, if this group exists. This method returns **true** if the group exists and the client is not already in the group; otherwise **false**. The client identifier is validated against the list of registered clients on the server. However, as the client has no such list of other clients, any **clientId** value other than **null** is accepted.

addOidMap

Signature `addOidMap(incomingOid: String;
 newObject: Object);`

The **addOidMap** method of the **ReplicateAgent** class manually adds an oid mapping on the server for the object specified by the **newObject** parameter when the framework application is bypassed.

This method would be called from your implementation of the **JIReplicateControl** interface **preApply** method.

You can obtain the value for the **incomingOid** parameter by calling the **ReplicateApplyObject** class **getLocalId** method.

addUserMessage

Signature `addUserMessage(type: Integer;
 id: String;
 messageText: String;
 targetAllClients: Boolean;
 targetClientIds: StringArray);`

The **addUserMessage** method of the **ReplicateAgent** class is called when a user message is to be inserted into the outgoing captured transaction queue.

Note The method enters transaction state to add the message as a separate persistent queue entry. Consequently, you should call the **addUserMessage** method only when you are *not* in transaction state.

The **type** and **id** parameters identify individual messages. If the **id** parameter value exceeds 20 characters, only the first 20 characters of the value are used. The value of the **messageText** parameter is the content of the user message. The **type**, **id**, and **messageText** parameters are passed to the registered user method receiver (that is, the **JIReplicateControl** interface **userMessage** method), if there is one, when the message is processed during replication.

If all registered clients are the target of the message, set the **targetAllClients** parameter to **true**. To set a number of clients, populate the **targetClientIds** array parameter with the identifiers for each of the target clients. (You can use client group identifiers as well as client identifiers.) The **targetClientIds** parameter can be **null**, in place of an empty array.

A user message call on the client does not need to set target client values to pass the message to the server.

clearConflicts

Signature `clearConflicts();`

The **clearConflicts** method of the **ReplicateAgent** class deletes persistent instances of the **JRFEErrorDetail** class for the client or server. Do not call this method in transaction state.

disableTracing

Signature `disableTracing();`

The **disableTracing** method of the **ReplicateAgent** class turns off the automatic capture of changes at commit time to the JADE Replication Framework. By default, changes are not automatically captured.

getCurrentQueue

Signature `getCurrentQueue(queue: ReplicateQueue input);`

The **getCurrentQueue** method of the **ReplicateAgent** class sets the transient **queue** parameter instance as a proxy for the internal transaction capture queue on the client or server.

getCurrentReplicateQueueSize

Signature `getCurrentReplicateQueueSize(): Integer;`

The **getCurrentReplicateQueueSize** method of the **ReplicateAgent** class returns the number of replicated transactions contained in the current queue.

getMaximumTransferSize

Signature `getMaximumTransferSize(): Integer;`

The **getMaximumTransferSize** method of the **ReplicateAgent** class returns the upper limit in bytes before the queue of captured transactions is segmented into multiple transmissions for transfer.

If the value is zero (0), all queue entries are sent as one transmission. As individual transactions in the queue are always transferred as a single unit, this transfer size may be exceeded for large transactions.

getPostapplyMode

Signature `getPostapplyMode(): Integer;`

The **getPostapplyMode** method of the **ReplicateAgent** class returns an integer indicating whether the JADE Replication Framework calls registered **postApply** receivers when processing changes. You can compare the returned integer value with the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
ApplyModeAll	0
ApplyModeAllButLoad	2
ApplyModeNone	1

The **ApplyModeAllButLoad** value means that the **postApply** method is called from applied regular user changes but not from applied loads.

getPreapplyMode

Signature `getPreapplyMode(): Integer;`

The **getPreapplyMode** method of the **ReplicateAgent** class returns an integer indicating whether the framework calls registered **preApply** receivers when processing changes. You can compare the returned integer value with the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
ApplyModeAll	0
ApplyModeAllButLoad	2
ApplyModeNone	1

The **ApplyModeAllButLoad** value means that the **preApply** method is called from applied regular user changes but not from applied loads.

getTracingMode

Signature `getTracingMode(): Integer;`

The **getTracingMode** method of the **ReplicateAgent** class returns an integer specifying the operations that are captured as a result of user logic updates. You can compare the returned integer value with one of the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
TracingModeFull	1
TracingModeIgnoreLocalDeletes	2
TracingModeLocalCleanup	3
TracingModeNone	0

The **TracingModeIgnoreLocalDeletes** and **TracingModeLocalCleanup** partial tracing modes are intended for use on the client.

The **TracingModeIgnoreLocalDeletes** mode records all update operations apart from deletions on replicable classes and properties for later replication. A local-only delete message is recorded in place of a standard delete operation, and the oid is removed from the server's oid map for the client when this transaction is replicated. This stops the server sending the client further replication data about the object. By contrast, a standard delete operation causes the object to be deleted on the server, which in turn is replicated to all clients interested in that object.

The **TracingModeLocalCleanup** mode records only delete operations for later replication and it ignores other types of update. However, delete operations are interpreted as local-only rather than standard delete operations (similar to the **TracingModeIgnoreLocalDeletes** tracing mode), ensuring that objects on the server are not deleted.

getTransferPerTransaction

Signature `getTransferPerTransaction(): Boolean;`

The **getTransferPerTransaction** method of the **ReplicateAgent** class returns **true** if the captured transaction queues are serialized with one message for each single captured transaction; otherwise it returns **false**.

getUserLogLevel

Signature `getUserLogLevel(): Integer;`

The **getUserLogLevel** method of the **ReplicateClientAgent** class returns the level at which framework replication activity is to be logged to any registered user log receiver.

Logging is made available so that activity within the JADE Replication Framework can be traced for development and debugging purposes.

Logging occurs for a registered receiver instance at a level listed in the following table.

Returned Value	Level of Logging
0	None
1	Transaction-level activity
2	Individual property-level activity
3	The maximum available detail

The value of the user logging level is set by using the [setUserLogLevel](#) method.

isInReplication

Signature `isInReplication(): Boolean;`

The **isInReplication** method of the [ReplicateAgent](#) class returns **true** if the user logic is being executed from processing of replicated changes; otherwise **false**. You can use this method to bypass processing such as the generation of a unique identifier in a *create* method because the identifier value is replicated, not generated, in a replication scenario.

processResponse

Signature `processResponse(response: String);`

The **processResponse** method of the [ReplicateAgent](#) class is called after the server has executed the [replicateServerToClient](#) method and the client has executed the corresponding [replicateClientFromServer](#) method.

Use this method to enable the server to maintain its mappings of oids on the client.

purgeCurrentQueue

Signature `purgeCurrentQueue();`

The **purgeCurrentQueue** method of the [ReplicateAgent](#) class removes all captured replication changes from the current queue.

registerConflictReceiver

Signature `registerConflictReceiver(receiver: JIReplicateControl);`

The **registerConflictReceiver** method of the [ReplicateAgent](#) class registers the instance of the [JIReplicateControl](#) interface specified by the **receiver** parameter as the method receiver for [resolveConflict](#) messages that are automatically called from the JADE Replication Framework when applying replication changes and replication conflicts or when exceptions are detected.

If you do not register a conflict receiver, default processing of the conflict or exception is performed. In all cases, the conflict or exception details are saved persistently, and you can access them by calling the [getConflicts](#) method.

registerIdController

Signature `registerIdController(controller: JIReplicateControl) updating;`

The `registerIdController` method of the `ReplicateAgent` class registers the instance of the `JIReplicateControl` interface specified by the `controller` parameter as the method receiver for `getObjectForId` messages that are automatically called from the JADE Replication Framework if user-defined unique identifiers are in use.

Note This option is not yet implemented.

registerPostapplyReceiver

Signature `registerPostapplyReceiver(receiver: JIReplicateControl) updating;`

The `registerPostapplyReceiver` method of the `ReplicateAgent` class registers the instance of the `JIReplicateControl` interface specified by the `receiver` parameter as the method receiver for `postApply` messages that are automatically called from the JADE Replication Framework when replication changes are applied.

If a receiver is not registered, `postApply` processing is not performed.

registerPreapplyReceiver

Signature `registerPreapplyReceiver(receiver: JIReplicateControl) updating;`

The `registerPreapplyReceiver` method of the `ReplicateAgent` class registers the instance of the `JIReplicateControl` interface specified by the `receiver` parameter as the method receiver for `preApply` messages that are automatically called from the JADE Replication Framework when replication changes are applied.

If a receiver is not registered, `preApply` processing is not performed.

registerProgressReceiver

Signature `registerProgressReceiver(progressReceiver: JIProgressCallback) updating;`

The `registerProgressReceiver` method of the `ReplicateAgent` class registers the instance of the `JIProgressCallback` interface specified by the `progressReceiver` parameter as the method receiver for replication and transfer progress callback messages that are automatically called from the JADE Replication Framework.

registerTransactionReceiver

Signature `registerTransactionReceiver(receiver: JIReplicateControl) updating;`

The `registerTransactionReceiver` method of the `ReplicateAgent` class registers the instance of the `JIReplicateControl` interface specified by the `receiver` parameter as the method receiver for the `abortDBTransaction`, `beginDBTransaction`, and `commitDBTransaction` methods that are automatically called from the JADE Replication Framework when transaction control is required.

If a receiver is not registered, the standard JADE `abortTransaction`, `beginTransaction`, and `commitTransaction` actions are performed.

setPreapplyMode

Signature `setPreapplyMode(mode: Integer);`

The **setPreapplyMode** method of the **ReplicateAgent** class specifies whether the JADE Replication Framework calls registered **preApply** receivers when processing changes.

The calls, which are enabled by default, must be disabled if they are not required for a specific set of changes.

The value of the **mode** parameter can be one of the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
ApplyModeAll	0
ApplyModeAllButLoad	2
ApplyModeNone	1

In **ApplyModeAllButLoad** mode, the **preApply** method is called from applied regular user changes but not from applied loads.

setTracingMode

Signature `setTracingMode(mode: Integer);`

The **setTracingMode** method of the **ReplicateAgent** class specifies the operations that are captured as a result of user logic.

The value of the **mode** parameter can be one of the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
TracingModeFull	1
TracingModeIgnoreLocalDeletes	2
TracingModeLocalCleanup	3
TracingModeNone	0

The **TracingModeIgnoreLocalDeletes** and **TracingModeLocalCleanup** partial tracing modes are intended for use on the client.

The **TracingModeIgnoreLocalDeletes** mode records all update operations apart from deletions on replicable classes and properties for later replication. A local-only delete message is recorded in place of a standard delete operation, and the oid is removed from the oid map that the server maintains for the client. This stops the server sending the client further replication data about the object. By contrast, a standard delete operation causes the object to be deleted on the server, which in turn is replicated to all clients interested in that object.

The **TracingModeLocalCleanup** mode records only delete operations for later replication and it ignores other types of update. However, delete operations are interpreted as local-only rather than standard deletes (similar to the **TracingModeIgnoreLocalDeletes** tracing mode), ensuring that objects on the server are not deleted.

The values for the **level** parameter are listed in the following table.

Value	Level of Logging
0	None
1	Transaction-level activity
2	Individual property-level activity
3	The maximum available detail

unreplicateProperty

Signature `unreplicateProperty(prop: Property);`

The **unreplicateProperty** method of the **ReplicateAgent** class unregisters the property specified by the **prop** parameter for replication; that is, changes to the property are no longer replicated.

Tip You can selectively register properties without having to register each property individually, by calling the **replicateAllProperties** method and then unregistering the properties that are not required, by calling the **unreplicateProperty** method.

ReplicateApplyObject Class

All changes for each user object within a replicated transaction are consolidated into a single transient instance of the **ReplicateApplyObject** class for processing by a registered **preApply** or **postApply** receiver.

Transient instances of the **ReplicateApplyTransaction** and **ReplicateApplyObject** classes are populated for the transaction details under the following conditions.

- A captured transaction on the client or server is applied from the other side
- A **preApply** or **postApply** receiver is registered
- The **preApply** or **postApply** mode is set

If a single-valued property is assigned different values within one transaction for the same object instance, only the last assignment is represented in the **ReplicateApplyObject** instance. If an object is created and deleted again in a single transaction, a **ReplicateApplyObject** is not created.

ReplicateApplyObject instances relating to a transaction are deleted when that transaction is fully applied. For details about the methods defined in the **ReplicateApplyObject** class, see “**ReplicateApplyObject Methods**”, in the following subsection.

Inherits From: JRFUserClasses

Inherited By: None

ReplicateApplyObject Methods

The methods defined in the [ReplicateApplyObject](#) class are summarized in the following table.

Method	Description
getClassName	Returns the class name in the current schema of the object created, changed, or deleted
getInstance	Returns the corresponding user object in the current schema
getLocalId	Returns the internal oid value for use with the addOidMap and removeOidMap methods
getOperation	Returns the internal operation code for the action
getOperationForProperty	Returns an integer indicating whether the specified property was updated
getOperationForPropertyAt	Returns the operation with a specified index on the specified multi-valued property
getPropertyNames	Adds the names of all properties that are changed to a specified string array
getUserClass	Returns the local class of the object being created, changed, or deleted
getValueCountForProperty	Returns the number of operations on the specified multi-valued property
getValueForProperty	Returns the changed value of the specified single-valued property
getValueForPropertyAt	Returns the element added to the specified multi-valued property or removed from it
hasChangeForProperty	Returns true if the specified property has been changed

Note Single-valued properties are primitive type attributes and non-collection references. Multi-valued properties are collection references.

getClassName

Signature `getClassName(): String;`

The [getClassName](#) method of the [ReplicateApplyObject](#) class returns the mapped class name of the object being created, changed, or deleted; that is, the name in the current schema rather than the originating schema.

getInstance

Signature `getInstance(): Object;`

The [getInstance](#) method of the [ReplicateApplyObject](#) class returns the corresponding user object in this schema if it exists; that is, the object represented by the receiver.

The method returns **null** when it is used with the [preApply](#) method for a transaction in which the object is created. The method returns an invalid reference when it is used with the [postApply](#) method for a transaction in which the object is deleted.

getLocalId

Signature `getLocalId(): String;`

The [getLocalId](#) method of the [ReplicateApplyObject](#) class returns the internal object identifier (oid) value for use with the [ReplicateAgent](#) class [addOidMap](#) and [removeOidMap](#) methods.

All changes are represented in the **ReplicateApplyObject** instance. The operations on a collection are to add or remove an element. The first operation has an index number of one (1), the second two (2), and so on.

The method returns the **ReplicateAgent** class **OperationCollAdd** constant if an element is added during the transaction and the **OperationCollRemove** constant if an element is removed during the transaction.

getPropertyNames

Signature `getPropertyNames(names: StringArray input);`

The **getPropertyNames** method of the **ReplicateApplyObject** class adds the names of all properties that are changed to the **names** string array parameter. The array is cleared before any names are added.

The property names are mapped; that is, they are the names in the current schema, not the originating schema.

getUserClass

Signature `getUserClass(): Class;`

The **getUserClass** method of the **ReplicateApplyObject** class returns the local class of the object being created, changed, or deleted.

getValueCountForProperty

Signature `getValueCountForProperty(propertyName: String): Integer;`

The **getValueCountForProperty** method of the **ReplicateApplyObject** class returns the number of operations on the multi-valued property specified by the **propertyName** parameter.

Note This method is useful for multi-valued properties; that is, for collection references.

All changes are represented in the **ReplicateApplyObject** instance. The operations on a collection are to add or remove an element. The first operation has an index number of one (1), the second two (2), and so on.

getValueForProperty

Signature `getValueForProperty(propertyName: String): Any;`

The **getValueForProperty** method of the **ReplicateApplyObject** class returns the value of the single-valued property specified by the **propertyName** parameter, if it has changed.

Note This method is useful for single-valued properties; that is, primitive type attributes and non-collection references.

If a single-valued property is changed more than once in a transaction, only the last change is represented in the **ReplicateApplyObject** instance.

For a primitive type attribute, the changed value is returned. If the attribute has not changed, zero (0) is returned.

ReplicateApplyTransaction Methods

The methods defined in the [ReplicateApplyTransaction](#) class are summarized in the following table.

Method	Returns ...
getEntries	An array of ReplicateApplyObject instances for the transaction
getSequence	The sequence number of the transaction
getStartTime	The date and time of the original replicated transaction
getTransactionId	The transaction identifier number of the original replicated transaction

getEntries

Signature `getEntries(): ReplicateApplyObjectArray;`

The [getEntries](#) method of the [ReplicateApplyTransaction](#) class returns a reference to an array of [ReplicateApplyObject](#) instances: one for each unique user object created, changed, or deleted in the transaction.

getSequence

Signature `getSequence(): Integer;`

The [getSequence](#) method of the [ReplicateApplyTransaction](#) class returns the sequence number of the transaction, which is one (1) except for entries captured using the [replicateLoad](#) method of the [ReplicateServerAgent](#) class.

Each replicate load transaction started with the [replicateLoadTransaction](#) method created inside a single JADE transaction has a sequence number, with the first load transaction having sequence number one (1), the second two (2), and so on.

getStartTime

Signature `getStartTime(): Timestamp;`

The [getStartTime](#) method of the [ReplicateApplyTransaction](#) class returns the date and time of the original replicated transaction.

getTransactionId

Signature `getTransactionId(): Integer64;`

The [getTransactionId](#) method of the [ReplicateApplyTransaction](#) class returns the transaction identifier number of the original replicated transaction.

ReplicateClientAgent Class

The [ReplicateClientAgent](#) class represents a system that is participating in replication as a *client*. The client JADE system could have a subset of the data stored on the server. Replication of changes can occur in both directions.

A [ReplicateClientAgent](#) instance is created by calling the [ReplicateFramework](#) class [registerAsClient](#) method.

For details about the methods defined in the **ReplicateClientAgent** class, see “[ReplicateClientAgent Methods](#)”, in the following subsection.

Inherits From: [ReplicateAgent](#)

Inherited By: None

ReplicateClientAgent Methods

The methods defined in the **ReplicateClientAgent** class are summarized in the following table.

Method	Description
getClientId	Returns the client identifier for the receiver
getDeferredInputQueue	Sets the specified queue as a proxy for the internal queue of deferred transactions
getReplicateQueueSize	Returns the number of transactions in the replicate queue to be transferred to the server
processQueuedTransactions	Applies replication changes that were received previously with directApply set to false
registerTransferReceiver	Registers a receiver for callbacks to the receiveFromServer , sendResponse , and sendToServer methods
replicateClientFromServer	Receives queued changes from the server and specifies whether they are applied immediately
replicateClientToServer	Queues changes to be replicated on the server

getClientId

Signature `getClientId(): String;`

The **getClientId** method of the **ReplicateClientAgent** class returns the client identifier for the client specified in the **ReplicateFramework** class [registerAsClient](#) method.

getDeferredInputQueue

Signature `getDeferredInputQueue(queue: ReplicateQueue input);`

The **getDeferredInputQueue** method of the **ReplicateClientAgent** class sets the transient instance of the **ReplicateQueue** class specified by the **queue** parameter instance as a proxy for the internal queue of deferred transactions received from the server.

getReplicateQueueSize

Signature `getReplicateQueueSize(): Integer;`

The **getReplicateQueueSize** method of the **ReplicateClientAgent** class returns the number of message transactions contained in the replicate queue for the client that are to be transferred to the server.

processQueuedTransactions

Signature `processQueuedTransactions();`

The **processQueuedTransactions** method of the **ReplicateClientAgent** class applies replication changes that have been received by a client previously by executing the [replicateClientFromServer](#) method with the **directApply** parameter set to **false**.

registerTransferReceiver

Signature `registerTransferReceiver(receiver: JITransfer);`

The **registerTransferReceiver** method of the **ReplicateClientAgent** class registers the instance of the **JITransfer** interface specified by the receiver parameter as the method receiver for the **receiveFromServer**, **sendResponse**, and **sendToServer** callback methods that are automatically called from the framework when the **replicateClientFromServer** or **replicateClientToServer** method is executed.

If a receiver is not registered, an exception is raised by replication processing.

replicateClientFromServer

Signature `replicateClientFromServer(directApply: Boolean);`

The **replicateClientFromServer** method of the **ReplicateClientAgent** class directly applies the changes transferred from the server changes if the value of the **directApply** parameter is **true**; otherwise the transferred changes are queued and applied later by calling the **processQueuedTransactions** method.

replicateClientToServer

Signature `replicateClientToServer();`

The **replicateClientToServer** method of the **ReplicateClientAgent** class queues changes to be replicated on the server.

ReplicateConflictEntry Class

The **ReplicateConflictEntry** class is a proxy for the internal details of conflict and exception errors occurring during replication. This class provides an understandable representation of the saved conflict details from the persistent **JRFErroDetail** instances.

Transient instances of the **ReplicateConflictEntry** class are created in the framework when the **getConflicts** method of the **ReplicateAgent** class is called.

For details about the methods defined in the **ReplicateConflictEntry** class, see “**ReplicateConflictEntry Methods**”, in the following subsection.

Inherits From: JRFFrameworkClasses

Inherited By: None

ReplicateConflictEntry Methods

The methods defined in the **ReplicateConflictEntry** class are summarized in the following table.

Method	Returns ...
getActivityDetails	Details about the activity
getExceptionDetails	Details about the exception
getIds	Transaction identifier and client identifier
getUserText	Any text added to the conflict details by user logic

getIds

Signature `getIds(errorType: Integer output;
 tranId: Integer64 output;
 sequence: Integer output;
 clientId: String output);`

The **getIds** method of the **ReplicateConflictEntry** class sets the values of the output parameters to return the transaction identifier and client identifier, as follows.

Parameter	Value
errorType	An integer representing the type of conflict or error corresponding to one of the following ReplicateConflictObject class constants. <ul style="list-style-type: none">■ ChangeInvalidObject (6)■ ChangeNoMappedOid (4)■ ChangeNoUniqueld (5)■ ChangeReferenceInvalidObject (9)■ ChangeReferenceNoMappedOid (7)■ ChangeReferenceNoUniqueld (8)■ DeleteInvalidObject (3)■ DeleteNoMappedOid (1)■ DeleteNoUniqueld (2)■ ExceptionOnApply (10)
tranId	Transaction identifier of the captured transaction.
sequence	Sequence number of the captured transaction.
clientId	On the server, the client identifier of the client where the transaction originated; on a client, a null value.

getUserText

Signature `getUserText(): String;`

The **getUserText** method of the **ReplicateConflictEntry** class returns any text added to the error details by calling the **ReplicateAgent** class **setUserErrorText** method from within the **resolveConflict** callback method of a registered conflict receiver, which implements the **JIREplicateControl** interface.

ReplicateConflictObject Class

The **ReplicateConflictObject** class creates a parameter object for the **resolveConflict** callback method of a registered conflict receiver that implements the **JIREplicateControl** interface. The parameter object avoids passing individual parameters corresponding to each of the properties.

One transient instance is reused internally each time the **resolveConflict** method is called.

For details about the constants and properties defined in the **ReplicateConflictObject** class, see “[ReplicateConflictObject Constants](#)” and “[ReplicateConflictObject Properties](#)”, in the following subsections.

Inherits From: JRFUserClasses

Inherited By: None

ReplicateConflictObject Constants

The constants defined in the **ReplicateConflictObject** class are summarized in the following table.

Constant	Integer Value
ChangeInvalidObject	6
ChangeNoMappedOid	4
ChangeNoUniqueld	5
ChangeReferenceInvalidObject	9
ChangeReferenceNoMappedOid	7
ChangeReferenceNoUniqueld	8
ConflictActionContinue	1
ConflictActionDropItem	2
ConflictActionDropTransaction	3
ConflictActionRetry	5
ConflictActionStopProcessing	4
DeleteInvalidObject	3
DeleteNoMappedOid	1
DeleteNoUniqueld	2
ExceptionOnApply	10

ReplicateConflictObject Properties

The properties defined in the **ReplicateConflictObject** class are summarized in the following table.

Property	Description
applyAction	Operation that caused the exception
applyTransaction	Transaction that caused the exception
className	Class of the object in conflict or causing the exception
clientId	Identifier of the client
conflictType	Type of the conflict
correctedObject	Object to be used in place of the invalidObject reference
exceptionObject	Exception instance
invalidObject	Object being applied that caused the conflict
oidValue	Oid value of the object

Property	Description
parentObject	Object with the reference property that caused the conflict
propertyName	Name of the reference property that caused the conflict

applyAction

Type: Integer

The **applyAction** property of the [ReplicateConflictObject](#) class is set when the [resolveConflict](#) callback method is invoked from an exception. The integer value corresponds to one of the [ReplicateAgent](#) class constants listed in the following table.

Constant	Integer Value
OperationCollAdd	9
OperationCollRemove	11
OperationCreate	4
OperationDelete	6
OperationLoadObject	98
OperationLoadProperty	99
OperationLocalOnlyDelete	100
OperationSetProperty	7
OperationUpdate	3

applyTransaction

Type: [ReplicateApplyTransaction](#)

The **applyTransaction** property of the [ReplicateConflictObject](#) class is set when the method is called from an exception and is the same as the parameter to the [JIReplicateControl](#) interface [preApply](#) or [postApply](#) methods.

You can use this method to investigate and possibly correct the existing data so that the transaction can be retried.

className

Type: String

The **className** property of the [ReplicateConflictObject](#) class contains the class of the object in conflict or causing the exception

clientId

Type: String

The **clientId** property of the [ReplicateConflictObject](#) class contains the identifier of the client, which is the current client on the client or the originator on the server.

conflictType

Type: Integer

The **conflictType** property of the **ReplicateConflictObject** class contains the type of the conflict.

The integer value corresponds to one of the **ReplicateConflictObject** class constants listed in the following table.

Constant	Integer Value
ChangeInvalidObject	6
ChangeNoMappedOid	4
ChangeNoUniqueld	5
ChangeReferenceInvalidObject	9
ChangeReferenceNoMappedOid	7
ChangeReferenceNoUniqueld	8
DeleteInvalidObject	3
DeleteNoMappedOid	1
DeleteNoUniqueld	2
ExceptionOnApply	10

correctedObject

Type: Object

The **correctedObject** property of the **ReplicateConflictObject** class is a property that you can set in your user logic to provide an alternative for the object set as the **invalidObject** reference value.

exceptionObject

Type: Exception

The **exceptionObject** property of the **ReplicateConflictObject** class is a reference to the exception instance that is set when a method is called from an exception.

invalidObject

Type: Object

The **invalidObject** property of the **ReplicateConflictObject** class is a reference to the object whose application caused the conflict.

oidValue

Type: String

The **oidValue** property of the **ReplicateConflictObject** class is the oid value of the object. This is useful if the object itself is a **null** reference.

parentObject

Type: Object

The **parentObject** property of the **ReplicateConflictObject** class is set only for a property-level conflict and it contains a reference to the instance whose reference property has the conflict.

propertyName

Type: String

The **propertyName** property of the **ReplicateConflictObject** class is set only for a property-level conflict and is the name of the reference property that has the conflict.

ReplicateFramework Class

The **ReplicateFramework** class is the initial point of entry to the JADE Replication Framework.

A transient instance of this class is created and used throughout the life of the application. When this instance is deleted, it automatically deletes all associated internal framework transient instances; for example, the **ReplicateClientAgent** and **ReplicateServerAgent** instances together with all **ReplicateLoadClass** and **ReplicateMapClass** instances created by the calls to the **registerLoadClass** and **getMapClass** methods.

For details about the methods defined in the **ReplicateFramework** class, see “[ReplicateFramework Methods](#)”, in the following subsection.

Inherits From: JRFUserClasses

Inherited By: None

ReplicateFramework Methods

The methods defined in the **ReplicateFramework** class are summarized in the following table.

Method	Description
initializeFramework	Deletes all persistent replication data for the entire framework
registerAsClient	Registers the application with the framework as a client
registerAsServer	Registers the application with the framework as a server
removeSystem	Deletes all persistent replication framework data for the specified system only

initializeFramework

Signature `initializeFramework();`

The **initializeFramework** method of the **ReplicateFramework** class deletes all persistent replication data for the entire JADE Replication Framework.

Note Use this method during development and testing rather than in production.

registerAsClient

Signature

```
registerAsClient(systemName: String;
                 clientId: String;
                 serverId: String;
                 userSchema: Schema;
                 userAppContext: ApplicationContext):
                 ReplicateClientAgent;
```

The **registerAsClient** method of the **ReplicateFramework** class registers the application with the framework as a client. This method returns a reference to the **ReplicateClientAgent** transient instance, which is used for the majority of interactions with the JADE Replication Framework.

The parameters for this method are listed in the following table.

Parameter	Description
systemName	Identifies transactions from this schema or application from other schemas using the framework. This is needed because the framework persistent data is all held in a common map file, for all participating schemas. The system name must be unique across these schemas.
clientId	Identifies a client participating in replication for this system and must be unique within the system. It is used to distinguish replicated server transactions for each client.
serverId	The identifier of the associated server with which this client exchanges replication data.
userSchema	The schema for the current user application. As it is used by the framework to find class and property references, it must be the lowest-level schema in the current hierarchy with persistent classes participating in replication.
userAppContext	Reference to the current appContext value. It is used when the framework calls back to the user schema.

registerAsServer

Signature

```
registerAsServer(systemName: String;
                 serverId: String;
                 userSchema: Schema;
                 userAppContext: ApplicationContext):
                 ReplicateServerAgent;
```

The **registerAsServer** method of the **ReplicateFramework** class registers the application with the framework as a server. This method returns a reference to the **ReplicateServerAgent** transient instance, which is used for the majority of interactions with the JADE Replication Framework.

The parameters for this method are listed in the following table.

Parameter	Description
systemName	Identifies transactions from this schema or application from other schemas using the framework. This is needed because the framework persistent data is all held in a common map file, for all participating schemas. The system name must be unique across these schemas.
serverId	Distinguishes the server from clients and must be a distinct value from those of the clients.
userSchema	The schema for the current user application. As it is used by the framework to find class and property references, it must be the lowest-level schema in the current hierarchy with persistent classes participating in replication.
userAppContext	The current appContext value. It is used when the framework calls back to the user schema.

removeSystem

Signature `removeSystem(systemName: String): Boolean;`

The **removeSystem** method of the **ReplicateFramework** class deletes all persistent replication framework data for the system specified by the **systemName** parameter, without affecting the data for other systems. This method returns **false** if the system is not present; otherwise it returns **true** if the system exists.

ReplicateLoadClass Class

The **ReplicateLoadClass** class represents a class on the server whose instances can be loaded to client systems. The **ReplicateServerAgent** class **registerLoadClass** method creates an instance of the **ReplicateLoadClass** class that is associated with a specified user class.

The **ReplicateLoadClass** class methods specify the properties that are replicated when the **replicateLoad** method of the **ReplicateServerAgent** class is called.

For details about the methods defined in the **ReplicateLoadClass** class, see “[ReplicateLoadClass Methods](#)”, in the following subsection.

Inherits From: `JRFUserClasses`

Inherited By: `ReplicateMapClass`

ReplicateLoadClass Methods

The methods defined in the **ReplicateLoadClass** class are summarized in the following table.

Method	Description
registerAllPropertiesUpTo	Registers all properties up to the specified superclass for loading
registerProperty	Registers a specified property for loading
unregisterProperty	Unregisters a specified property for loading

registerAllPropertiesUpTo

Signature `registerAllPropertiesUpTo(cls: Class);`

The **registerAllPropertiesUpTo** method of the **ReplicateLoadClass** class registers all properties in the user class associated with the receiver up to and including the superclass specified by the **cls** parameter. Registered properties are loaded for an object by calling the **ReplicateServerAgent** class **replicateLoad** method.

In the following code fragment, **Customer** is a user class that inherits from **Person**.

```
// Register the Customer class for loading
replicateLoadClass := serverAgent.registerLoadClass(Customer);
// Register all properties in Customer and its superclass Person for loading
replicateLoadClass.registerAllPropertiesUpTo(Person);
```

Tip You can selectively register properties without having to register each property individually, by calling the **registerAllPropertiesUpTo** method and then unregistering the properties that are not required, by calling the **unregisterProperty** method.

registerProperty

Signature `registerProperty(prop: Property);`

The **registerProperty** method of the **ReplicateLoadClass** class registers a property in the user class associated with the receiver or one of its superclasses. Only properties that are registered have their current values captured when the **replicateLoad** method of the **ReplicateServerAgent** class is executed.

unregisterProperty

Signature `unregisterProperty(prop: Property);`

The **unregisterProperty** method of the **ReplicateLoadClass** class unregisters a property of the user class associated with the receiver or one of its superclasses. Only properties that are registered have their current values captured when the **replicateLoad** method of the **ReplicateServerAgent** class is executed.

ReplicateMapClass Class

The **ReplicateMapClass** class represents a class on the server whose instances can be replicated to a client system with different names for mapped classes and properties.

The **getMapClass** method of the **ReplicateServerAgent** class creates an instance of the **ReplicateMapClass** class that maps a specified server class to its corresponding client class.

For details about the methods defined in the **ReplicateMapClass** class, see “[ReplicateMapClass Methods](#)”, in the following subsection.

Inherits From: [ReplicateLoadClass](#)

Inherited By: None

ReplicateMapClass Methods

The method defined in the **ReplicateMapClass** class is summarized in the following table.

Method	Description
mapProperty	Specifies the names of corresponding properties on the server and client for replication

mapProperty

Signature `mapProperty(prop: Property;
 propertyName: String) updating;`

The **mapProperty** method of the **ReplicateMapClass** class enables you to specify the names of corresponding properties on the server and client.

This method is used when applying replication changes from client to server or server to client and the class or property names differ in each schema for corresponding items.

In the following code fragment, instances of the **Customer** class on the server are replicated on the client as instances of the **Company** class.

```
// Register Customer instances for replication as Business instances
replicateMapClass := serverAgent.getMapClass(Customer,systemName,"Company");
// Map the 'Customer::address' property to Company::street'
replicateMapClass.mapProperty(Customer::address, "street");
```

ReplicateQueue Class

The **ReplicateQueue** class is a proxy for the internal client and server queues of objects to be loaded, captured transactions for immediate application, and captured transactions for deferred application.

A transient instance of this class is created in your user logic and passed to the **ReplicateAgent** class **getCurrentQueue** method, the **ReplicateClientAgent** class **getDeferredInputQueue** or **getDeferredInputQueue** method, or the **ReplicateServerAgent** class **getLoadQueue** method, where it is initialized with entries for that queue.

For details about the methods defined in the **ReplicateQueue** class, see “[ReplicateQueue Methods](#)”, in the following subsection.

Inherits From: JRFUserClasses

Inherited By: None

ReplicateQueue Methods

The methods defined in the **ReplicateQueue** class are summarized in the following table.

Method	Returns ...
getIds	The range of transaction identifiers and sequences in the queue
getQueueName	The internal name of the queue
getQueueType	The type of the queue; that is, current or split
getTransactions	An array of ReplicateQueueTransaction instances for the transactions in the queue

getIds

Signature getIds(firstTransactionId: Integer64 output;
 firstSequence: Integer output;
 lastTransactionId: Integer64 output;
 lastSequence: Integer output);

The **getIds** method of the **ReplicateQueue** class returns the range of transaction identifiers and sequences in the queue for the first and last transactions in the queue.

getQueueName

Signature getQueueName(): String;

The **getQueueName** method of the **ReplicateQueue** class returns the internal name of the queue.

getQueueType

Signature `getQueueType(): Integer;`

The **getQueueType** method of the **ReplicateQueue** class returns the type of the queue, which is zero (0) for the current queue and one (1) for a split queue. You can split a queue, by calling the **ReplicateServerAgent** class **splitMessageQueue** method.

getTransactions

Signature `getTransactions(): ReplicateQueueTransactionArray updating;`

The **getTransactions** method of the **ReplicateQueue** class returns a reference to an array of **ReplicateQueueTransaction** transient instances, one for each transaction in the queue.

ReplicateQueueEntry Class

The **ReplicateQueueEntry** class is a proxy for the internal client and server object-level or property-level changes within a transaction in the current queue.

Multiple property changes to the same object are captured as a single entry. An array of instances of this class is returned by the **getEntries** method of the **ReplicateQueueTransaction** class. For details about the methods defined in the **ReplicateQueueEntry** class, see “**ReplicateQueueEntry Methods**”, in the following subsection.

Inherits From: JRFUserClasses

Inherited By: None

ReplicateQueueEntry Methods

The methods defined in the **ReplicateQueueEntry** class are summarized in the following table.

Method	Returns ...
getClassname	Local name of the class for the object change
getLocalId	Local identifier (object identifier or user-defined unique identifier) for the object change
getOperation	Operation code for the object change
getPropertyCount	Number of properties with changed values in the entry
getPropertyNameAt	Local name of the property for the specified property change
getPropertyOperationAt	Type of change for the specified property change
getPropertyValueAt	Value to which the property was changed for the specified property change

getClassname

Signature `getClassname(): String;`

The **getClassname** method of the **ReplicateQueueEntry** class returns the local name of the class for the object change that has been captured.

getLocalId

Signature `getLocalId(): String;`

The **getLocalId** method of the **ReplicateQueueEntry** class returns the local identifier (the object identifier or the user-defined unique identifier) for the object whose change has been captured.

getOperation

Signature `getOperation(): Integer;`

The **getOperation** method of the **ReplicateQueueEntry** class returns an internal operation code for the type of change captured.

The returned integer value corresponds to one of the **ReplicateAgent** class constants listed in the following table.

Constant	Integer Value
OperationCollAdd	9
OperationCollRemove	11
OperationCreate	4
OperationDelete	6
OperationLoadObject	98
OperationLoadProperty	99
OperationLocalOnlyDelete	100
OperationSetProperty	7
OperationUpdate	3

getPropertyCount

Signature `getPropertyCount(): Integer;`

The **getPropertyCount** method of the **ReplicateQueueEntry** class returns the number of properties in the entry with changed values.

getPropertyNameAt

Signature `getPropertyNameAt(indx: Integer): String;`

The **getPropertyNameAt** method of the **ReplicateQueueEntry** class returns the local name of the property for the change identified by the value of the **indx** parameter, which must be in the range one (1) through the value returned by the **getPropertyCount** method.

getPropertyOperationAt

Signature `getPropertyOperationAt(indx: Integer): Integer;`

The **getPropertyOperationAt** method of the **ReplicateQueueEntry** class returns an internal code for the type of property change identified by the value of the **indx** parameter, which must be in the range one (1) through the value returned by the **getPropertyCount** method.

The returned integer corresponds to one of the [ReplicateAgent](#) class constants listed in the following table.

Constant	Integer Value
OperationCollAdd	9
OperationCollRemove	11
OperationCreate	4
OperationDelete	6
OperationLoadObject	98
OperationLoadProperty	99
OperationLocalOnlyDelete	100
OperationSetProperty	7
OperationUpdate	3

getPropertyValueAt

Signature `getPropertyValueAt(indx: Integer): String;`

The [getPropertyValueAt](#) method of the [ReplicateQueueEntry](#) class returns a [String](#) representation of the value to which the property was changed for the change identified by the value of the **indx** parameter, which must be in the range one (1) through the value returned by the [getPropertyCount](#) method.

For [Binary](#) properties, it is in **base64** format. For other primitive type properties, a return value of “**null**” indicates a **null** value for that type. For references, the return value is the local identifier (object identifier or user-defined unique identifier) of the reference.

ReplicateQueueTransaction Class

The [ReplicateQueueTransaction](#) class is a proxy for the internal client and server transactions for a queue.

An array of instances of this class are returned by the [ReplicateQueue](#) class [getTransactions](#) method.

For details about the methods defined in the [ReplicateQueueTransaction](#) class, see “[ReplicateQueueTransaction Methods](#)”, in the following subsection.

Inherits From: [JRFUserClasses](#)

Inherited By: None

ReplicateQueueTransaction Methods

The methods defined in the [ReplicateQueueTransaction](#) class are summarized in the following table.

Method	Returns ...
getEntries	Array of ReplicateQueueEntry instances representing changes within the transaction
getOidsForClient	List of object identifiers within the transaction that are targeted for replication on the specified client

Method	Returns ...
getOriginatorId	Client or server identifier where the transaction originally occurred
getSequence	Sequence number of the replication transaction within a JADE transaction
getTargetClientIds	List of client identifiers for which the transaction is targeted
getTransactionId	JADE transaction identifier of the original captured transaction
getTransactionStartTime	Date and time when the captured transaction was committed
hasTransactionTargetClients	true if the setTransactionTargetClients method was called to identify clients explicitly

getEntries

Signature `getEntries(): ReplicateQueueEntryArray;`

The **getEntries** method of the [ReplicateQueueTransaction](#) class returns a reference to an array of transient [ReplicateQueueEntry](#) instances, each representing one or more property changes to a single object within the transaction.

getOidsForClient

Signature `getOidsForClient(clientId: String;
oids: StringArray input);`

The **getOidsForClient** method of the [ReplicateQueueTransaction](#) class populates the **oids** string array parameter with a list of object identifiers within the transaction that are targeted for replication on the client specified by the **clientId** parameter.

If the agent [setTransactionTargetClients](#) method is called within a captured transaction, the list of object identifiers is empty but changes in the transaction are targeted to all clients returned by the [getTargetClientIds](#) method.

getOriginatorId

Signature `getOriginatorId(): String;`

The **getOriginatorId** method of the [ReplicateQueueTransaction](#) class returns the identifier of the client or server where the transaction originally occurred.

getSequence

Signature `getSequence(): Integer;`

The **getSequence** method of the [ReplicateQueueTransaction](#) class returns the sequence number of the transaction. For a captured transaction, this value is always one (1). For entries captured using the [ReplicateServerAgent](#) class [replicateLoad](#) method, each replicate load transaction started with the [replicateLoadTransaction](#) method created inside a single JADE transaction has a sequence number, with the first load transaction having sequence number one (1), the second two (2), and so on.

getTargetClientIds

Signature `getTargetClientIds(clients: StringArray input);`

The **getTargetClientIds** method of the [ReplicateQueueTransaction](#) class populates the string array specified by the **clients** parameter with a list of client identifiers for which the transaction is targeted.

getTransactionId

Signature `getTransactionId(): Integer64;`

The `getTransactionId` method of the `ReplicateQueueTransaction` class returns the JADE transaction identifier of the original captured transaction.

getTransactionStartTime

Signature `getTransactionStartTime(): TimeStamp;`

The `getTransactionStartTime` method of the `ReplicateQueueTransaction` class returns the date and time at which the captured transaction was committed.

hasTransactionTargetClients

Signature `hasTransactionTargetClients(): Boolean;`

The `hasTransactionTargetClients` method of the `ReplicateQueueTransaction` class returns `true` if the agent `setTransactionTargetClients` method was called within the captured transaction, to identify target clients for replication explicitly.

ReplicateServerAgent Class

The `ReplicateServerAgent` class represents a system that is participating in replication as the *server*.

The server JADE system has a super-set of the data stored on any client. Replication of changes can occur in both directions. The server is considered to have the master copy of data shared between clients and the server.

Create a `ReplicateServerAgent` instance by calling the `ReplicateFramework` class `registerAsServer` method.

Some methods have the same name as methods in the `ReplicateClientAgent` class but have a different method signature, typically an additional `clientId` parameter. For details about the methods defined in the `ReplicateServerAgent` class, see “[ReplicateServerAgent Methods](#)”, in the following subsection.

Inherits From: `ReplicateAgent`

Inherited By: None

ReplicateServerAgent Methods

The methods defined in the `ReplicateServerAgent` class are summarized in the following table.

Method	Description
<code>addOidForClient</code>	Manually adds the object identifier mapping for an object that exists on the client and the server
<code>beginReplicateLoad</code>	Specifies the target client for subsequent <code>replicateLoad</code> method calls
<code>clientHasReplicableObject</code>	Returns <code>true</code> if the specified object exists in the internal oid map for the specified client
<code>deregisterClient</code>	Removes the specified client from the list of clients registered with the server
<code>endReplicateLoad</code>	Signals the end of the load process
<code>getCallReplToNominatedClients</code>	Returns <code>true</code> if the <code>replicateToNominatedClients</code> method is called on replicated objects

Method	Description
getClientIds	Returns the identifiers of all registered clients
getDeferredInputQueue	Sets the queue parameter as a proxy for the internal queue of deferred transactions from the specified client
getInitialClientIds	Returns the transaction identifier and sequence at which replication starts for the specified client
getInitialOidEqualityMode	Returns true if client environments are set up as copies of the server
getInlineQueueRemovalMode	Returns true if entries in the server queue of captured transactions are automatically removed after they are sent to all registered clients
getLastProcessedClientIds	Returns the transaction identifier and sequence from which replication was last carried out for the specified client
getLastTransferredClientIds	Returns the transaction identifier and sequence of the last replication transaction transferred to the specified client
getLoadQueue	Sets the queue parameter as a proxy for the internal queue of replicateLoad transactions waiting to be sent to the specified client
getMapClass	Returns a ReplicateMapClass instance for the specified local class, which maps class and property names between server and client
getQueues	Returns an array of all queues on the server
getServerId	Returns the server identifier
getTotalReplicateQueueSize	Returns the total number of transactions in the current server queue and all split queues
processQueuedTransactions	Applies previously queued input replication changes from the specified client
purgeDetailsForClient	Deletes saved object identifier mappings for the specified client
registerClient	Adds the specified client and system name to the list of clients registered with the server
registerLoadClass	Returns a ReplicateLoadClass instance for the specified local class so that instances of the class can be loaded to clients
removeQueueEntry	Removes the specified captured replicable transaction entry from the current queue
removeSentQueueEntries	Removes captured replicable transaction entries that have been sent to clients
replicateLoad	Performs an initial data load to a client
replicateLoadTransaction	Starts a new transaction when load objects are applied on the client
replicateServerFromClient	Processes replication information received from the client
replicateServerToClient	Returns replication information which is sent to the specified client
setCallReplToNominatedClients	Specifies whether the replicateToNominatedClients method is called instead of the replicateToClient method for each replicated object
setInitialClientIds	Sets the transaction identifier and sequence for replication to a specified new client
setInitialOidEqualityMode	Specifies whether client environments are set up as copies of the server
setInlineQueueRemovalMode	Specifies whether entries in the server queue of captured transactions are automatically removed after they are sent to all registered clients
setSyncPointForClient	Sets or clears the end-point in the replication queue for the specified client
splitMessageQueue	Splits the current replicate queue on the server at the last entry

getInlineQueueRemovalMode

Signature `getInlineQueueRemovalMode(): Boolean;`

The `getInlineQueueRemovalMode` method of the `ReplicateServerAgent` class returns `true` if entries in the server queue of captured replicable transactions are automatically removed when they have been sent to all registered clients.

The `setInlineQueueRemovalMode` method sets this value for the JADE Replication Framework.

getLastProcessedClientIds

Signature `getLastProcessedClientIds(clientId: String;
transactionId: Integer64 output;
sequence: Integer output);`

The `getLastProcessedClientIds` method of the `ReplicateServerAgent` class updates the `transactionId` and `sequence` output parameters with values from which replication was last carried out for the client specified by the `clientId` parameter. All entries prior to these identifiers have been sent and processed.

getLastTransferredClientIds

Signature `getLastTransferredClientIds(clientId: String;
transactionId: Integer64 output;
sequence: Integer output);`

The `getLastTransferredClientIds` method of the `ReplicateServerAgent` class updates the `transactionId` and `sequence` output parameters with values from the last replication transaction that was transferred to the client specified by the `clientId` parameter.

getLoadQueue

Signature `getLoadQueue(clientId: String;
queue: ReplicateQueue input);`

The `getLoadQueue` method of the `ReplicateServerAgent` class sets the transient `queue` parameter instance as a proxy for the internal queue of `replicateLoad` transactions waiting to be sent to the client specified by the `clientId` parameter.

getMapClass

Signature `getMapClass(class: Class;
systemName: String;
mappedClassName: String): ReplicateMapClass;`

The `getMapClass` method of the `ReplicateServerAgent` class returns a transient instance of the `ReplicateMapClass` class for the local class specified by the `class` parameter. This instance is used to map class and property names between the server and client.

The `systemName` parameter is used on the server to enable client targets to have different class names. When the `ReplicateServerAgent` instance is deleted, all associated `ReplicateMapClass` instances are automatically deleted with it.

Mapping of class and property names is done on the server and is not required on the client.

getQueues

Signature `getQueues(): ObjectArray;`

The **getQueues** method of the **ReplicateServerAgent** class returns an object array containing instances of the **ReplicateQueue** class for all queues on the server.

getServerId

Signature `getServerId(): String;`

The **getServerId** method of the **ReplicateServerAgent** class returns the server identifier that is set by calling the **registerAsServer** method of the **ReplicateFramework** class.

getTotalReplicateQueueSize

Signature `getTotalReplicateQueueSize(): Integer;`

The **getTotalReplicateQueueSize** method of the **ReplicateServerAgent** class returns the total number of message transactions contained in the current server queue and all split queues.

processQueuedTransactions

Signature `processQueuedTransactions(clientId: String);`

The **processQueuedTransactions** method of the **ReplicateServerAgent** class method applies previously queued input replication changes from the client specified by the **clientId** parameter to the server. The client changes were queued by calling the **replicateServerFromClient** method with the value of the **directApply** parameter set to **false**.

purgeDetailsForClient

Signature `purgeDetailsForClient(clientId: String);`

The **purgeDetailsForClient** method of the **ReplicateServerAgent** class deletes the saved object identifier (oid) mappings for the client specified by the **clientId** parameter.

Use this method to remove all data for a client before reloading it from the server using the **replicateLoad** method.

Call the **deregisterClient** method, to automatically delete the saved object identifier (oid) mappings.

registerClient

Signature `registerClient(clientId: String;
clientSystemName: String): Boolean;`

The **registerClient** method of the **ReplicateServerAgent** class adds an entry for the client and system specified by the **clientId** and **clientSystemName** parameters to the list of valid clients of the server, which is held persistently.

The client system name is saved with the value of the **clientId** parameter for use in the class mapping. An attempt to register the same client and system name is ignored.

The JADE Replication Framework accepts input from registered clients only. Input from unregistered clients raises an exception.

registerLoadClass

Signature `registerLoadClass(class: Class): ReplicateLoadClass;`

The **registerLoadClass** method of the **ReplicateServerAgent** class returns a transient instance of the **ReplicateLoadClass** class for the local class specified by the **class** parameter so that instances of the class can subsequently be loaded to clients, by calling the **replicateLoad** method.

removeQueueEntry

Signature `removeQueueEntry(tranId: Integer64;
 sequence: Integer);`

The **removeQueueEntry** method of the **ReplicateServerAgent** class removes the entry with the transaction identifier and sequence number specified by the **tranId** and **sequence** parameters from the current queue.

removeSentQueueEntries

Signature `removeSentQueueEntries();`

The **removeSentQueueEntries** method of the **ReplicateServerAgent** class removes entries from the server queue of captured replicable transactions that have been sent to clients.

Call the **setInlineQueueRemovalMode** method with the **mode** parameter set to **true**, to automatically remove entries.

Tip Call the **removeSentQueueEntries** method when replicable transactions are not being captured and replication to clients is not taking place. This method locks the queue at the start and exits immediately if the lock cannot be applied.

replicateLoad

Signature `replicateLoad(obj: Object) updating;`

The **replicateLoad** method of the **ReplicateServerAgent** class is used for initial data loading to the client, where the data is to be fully replicated between client and server. This method adds an entry for an existing object to the server replication queue for the target client only, which is replicated on the client as an add operation if the object does not yet exist, or an update operation if it does exist.

As the client load processing resolves references automatically, the order of loading objects does not depend on the relationships between them. The client load does this by creating an object from a reference if the object does not yet exist so that the subsequent load entry for this object populates the object.

Call this method when in transaction state.

replicateLoadTransaction

Signature `replicateLoadTransaction() updating;`

The **replicateLoadTransaction** method of the **ReplicateServerAgent** class is called optionally within a block of **replicateLoad** method calls, to cause a new transaction when load objects are applied if the load is to be applied as a number of transactions.

Call this method when in transaction state.

This chapter covers the following topics.

- [Overview](#)
- [ReplicationBasicCommsPackage Interfaces](#)
- [ReplicationBasicCommsPackage Classes](#)
- [JRFCComms Section of the JADE Initialization File](#)

Overview

The **ReplicationBasicCommsPackage** package exported from the **JadeReplicationSchema** schema into your user-defined schema enables replication messages to be sent between participating systems using TCP/IP. This package provides an optional communications layer to connect client and server systems and to simplify initial development using the JADE Replication Framework.

The core replication functionality in the JADE Replication Framework is contained in the **ReplicationPackage** package exported from the **JadeReplicationSchema** schema. For details, see Chapter 2, “[Replication Package](#)”.

This chapter is a reference to the API provided by the **ReplicationBasicCommsPackage** package. For examples of the use of the classes, methods, and interfaces in this package, see Chapter 4, “[Using the JADE Replication Framework](#)”.

Note The entities documented in this chapter are those that are available when you import the **ReplicationBasicCommsPackage** package into a user-defined schema (that is, they do *not* include all entities in the **JadeReplicationSchema** itself, many of which are system-specific).

This package, which makes use of the **JITransfer** interface, is an optional communications mechanism. You can write alternative communications mechanisms, by using Web services or the TCP/IP communications protocol and the **JITransfer** interface.

Replication Basic Comms Package Interfaces

The **ReplicationBasicCommsPackage** package provides the following interfaces.

Interface	Enables a ...
JITCPClientCallback	Client to process information appended or prepended to replication data
JITCPServerCallback	Server to process information appended or prepended to replication data and to perform actions requested by clients

JITCPClientCallback Interface

The **JITCPClientCallback** interface is an optional interface that you require only if additional user information is to be added to transferred messages or if progress is to be reported.

The **JRFTCPClient** class invokes methods for an instance of a class implementing this interface that is registered as the TCP/IP client callback receiver.

The TCP/IP client is an instance of the **JRFTCPClient** class and can register callback receivers using its **initialize** method or **setTCPClientCallback** method. For details about the methods defined in the **JITCPClientCallback** interface, see "[JITCPClientCallback Methods](#)", in the following subsection.

JITCPClientCallback Methods

The methods defined in the **JITCPClientCallback** interface are summarized in the following table.

Method	Callback method invoked ...
processInput	When a replication message is received
processOutput	Before a replication message is sent

processInput

Signature `processInput(message: Binary io);`

The **processInput** method of the **JITCPClientCallback** interface is called before a message containing serialized replication data is processed. The message content received by the JADE Replication Framework is contained in the **message** parameter.

Your implementation of this method can strip off and process any user-added prefix or suffix that is added to this message by the server. It should not change the core content, as this would invalidate replication processing.

processOutput

Signature `processOutput(message: Binary io);`

The **processOutput** method of the **JITCPClientCallback** interface is called before a message containing serialized replication data is sent by the JADE Replication Framework. The message content generated by the JADE Replication Framework is contained in the **message** parameter.

Your implementation of this method can add a prefix or suffix to this message. It should not change the core content, as this would invalidate replication processing.

JITCPServerCallback Interface

The **JITCPServerCallback** interface is required to support the **actionRequested** callback.

The JADE Replication Framework invokes methods for an instance of a class implementing this interface that is registered as the TCP/IP server callback receiver. The TCP/IP server is an instance of the **JRFTCPClient** class and can register callback receivers using its **initialize** method or **setTCPServerCallback** method.

For details about the methods defined in the **JITCPServerCallback** interface, see “**JITCPServerCallback Methods**”, in the following subsection.

JITCPServerCallback Methods

The methods defined in the **JITCPServerCallback** interface are summarized in the following table.

Method	Callback method that ...
actionRequested	Carries out a request for a specified client
handleUserMessage	Handles a non-replication message from a client and sends a response
processInput	Processes a replication message when it is received
processOutput	Processes a replication message before it is sent

actionRequested

Signature `actionRequested(clientId: String;
 actionType: String);`

The **actionRequested** method of the **JITCPServerCallback** interface actions the request specified by the **actionType** parameter received from the client specified by the **clientId** parameter.

The request originates on the client with the **requestAction** method being called on an instance of the **JRFTCPClient** class (the method is in fact a **JITTransfer** interface method), typically to request the loading of objects when the next replication connection is made.

The **actionRequested** method does not return data at the time of the method call.

handleUserMessage

Signature `handleUserMessage(message: String): String;`

The **handleUserMessage** method of the **JITCPServerCallback** interface handles a non-replication message from a client and sends a response. The message content is contained in the **message** parameter. The message originates on the client with the **sendUserMessage** method being called on an instance of the **JRFTCPClient** class.

The return value of the **handleUserMessage** method is the response to the message and it is sent back to the client to become the return value from the client **sendUserMessage** method.

processInput

Signature `processInput(message: Binary io);`

The **processInput** method of the [JITCPServerCallback](#) interface is called before a message containing serialized replication data is processed. The message content received by the JADE Replication Framework is contained in the **message** parameter.

Your implementation of the method can strip off and process a prefix or suffix added to this message by the server. It should not change the core content, as this would invalidate replication processing.

processOutput

Signature `processOutput(message: Binary io);`

The **processOutput** method of the [JITCPServerCallback](#) interface is called before a message containing serialized replication data is sent by the framework. The message content generated by the framework is contained in the **message** parameter.

Your implementation of the method can add a prefix or suffix to this message. It should not change the core content, as this would invalidate replication processing.

ReplicationBasicCommsPackage Classes

The ReplicationPackage provides the following classes.

Class	Enables a ...
JRFTCPClient	Client to connect using TCP/IP, with the server to carry out replication
JRFTCPServer	Server to connect using TCP/IP, with clients to carry out replication

JRFTCPClient Class

The **JRFTCPClient** class implements the JADE Replication Framework [JITransfer](#) interface so that messages are sent to and received from the server automatically when the client agent [replicateClientFromServer](#) and [replicateClientToServer](#) methods are called.

For details about the methods defined in the **JRFTCPClient** class, see “[JRFTCPClient Methods](#)”, in the following subsection.

Inherits From: JRFTCPTransport

Inherited By: None

JRFTCPClient Methods

The methods defined in the **JRFTCPClient** class are summarized in the following table.

Method	Description
finalize	Closes the TCP/IP connection with the server
initialize	Specifies the client agent for the TCP/IP connection with the server
open	Opens the TCP/IP connection with the server

Method	Description
sendUserMessage	Sends non-replication messages across an open TCP/IP connection with the server
setOptions	Sets TCP/IP options to connect to the server
setTCPClientCallback	Sets the object to perform additional processing on replication messages that are sent and received

finalize

Signature `finalize() updating;`

The **finalize** method of the [JRFTCPClient](#) class closes the TCP/IP connection when no further communication with the server is required.

The same [JRFTCPClient](#) instance can be reused as the method receiver for the [open](#) method, to reestablish communication with the server.

initialize

Signature `initialize(clientAgent: ReplicateClientAgent;
 tcpClientCallback: JITCPClientCallback) updating;`

The **initialize** method of the [JRFTCPClient](#) class sets the client agent specified in the **clientAgent** parameter for the TCP/IP connection.

If you require additional processing of the sent and received messages, use the **tcpClientCallback** parameter to specify an instance of a class that implements the [JITCPClientCallback](#) interface. If you do not require this, you can pass a **null** reference as the **tcpClientCallback** parameter.

You can subsequently change this call-back instance, by calling the [setTCPClientCallback](#) method.

open

Signature `open() updating;`

The **open** method of the [JRFTCPClient](#) class opens the TCP/IP connection with the server. For the connection to be successful, the connection must have initialized using the [initialize](#) method and the server must be listening on the correct port.

sendUserMessage

Signature `sendUserMessage(message: String): String;`

The **sendUserMessage** method of the [JRFTCPClient](#) class sends non-replication messages (for example, initial sign on or handshaking messages) across an established TCP/IP connection with the server.

These messages are processed by the [handleUserMessage](#) callback method of the implementer of the [JITCPServerCallback](#) interface on the server and are otherwise ignored by the JADE Replication Framework. The return value of the **sendUserMessage** method is the value returned by the [handleUserMessage](#) method and sent back from the server.

setOptions

Signature `setOptions(hostName: String;
 portNumber: Integer;
 timeout: Integer;
 logFileName: String) updating;`

The **setOptions** method of the **JRFTCPClient** class sets TCP/IP options for a client connecting to the replication server. The options are used when the **open** method is called.

The parameters for this method and their default values are listed in the following table.

Parameter	Description	Default
hostName	Host name or IP address of the server	"localhost"
portNumber	Port number on which the server listens	9999
timeout	Time in milliseconds before a read or write operation times out	600000
logFileName	File containing connection error messages	""

The default values in this table apply if a value is not specified in the initialization file or by calling the **setOptions** method.

The parameter values specified in the **setOptions** method override the corresponding parameter values in the [JRFCComms] section of the JADE initialization file. For details, see “[[JRFCComms Section of the JADE Initialization File](#)]”, later in this chapter.

Note The **timeout** parameter for the **setOptions** method specifies the number of milliseconds whereas the **timeout** parameter in the [JRFCComms] section of the JADE initialization file specifies the number of seconds.

setTCPClientCallback

Signature `setTCPClientCallback(tcpClientCallback: JITCPClientCallback) updating;`

The **setTCPClientCallback** method of the **JRFTCPClient** class specifies the **tcpClientCallback** parameter as the method receiver for calls to perform additional processing on replication messages that are sent and received.

The value of the **tcpClientCallback** parameter must be an instance of a class that implements the **JITCPClientCallback** interface.

JRFTCPServer Class

The **JRFTCPServer** class implements server communications with clients using the TCP/IP communications protocol to facilitate the processing of replication changes.

For details about the methods defined in the **JRFTCPServer** class, see “[[JRFTCPServer Methods](#)]”, in the following subsection.

Inherits From: JRFTCPTransport

Inherited By: None

JRFTCPServer Methods

The methods defined in the **JRFTCPServer** class are summarized in the following table.

Method	Description
finalize	Closes the TCP/IP connection with the clients
getDirectApply	Returns true if replication changes are applied immediately and false if they are saved to the persistent input queue
initialize	Specifies the server agent for the TCP/IP connection with the server
open	Listens for TCP/IP connections with clients
setDirectApply	Sets whether replication changes are applied immediately or saved to the persistent input queue
setOptions	Sets TCP/IP options to listen for connections from clients
setTCPServerCallback	Sets the object to perform additional processing on messages and actions that are sent and received

finalize

Signature `finalize() updating;`

The **finalize** method of the **JRFTCPServer** class closes the TCP/IP connection when no further communication with clients is required.

You can reuse the same **JRFTCPServer** instance as the method receiver for the **open** method to listen for connections from clients.

getDirectApply

Signature `getDirectApply(): Boolean;`

The **getDirectApply** method of the **JRFTCPServer** class returns **true** if replication changes are applied immediately and it returns **false** if they are saved to the persistent input queue and applied when the server agent method **processQueuedTransactions** is called.

initialize

Signature `initialize(serverAgent: ReplicateServerAgent;
 tcpServerCallback: JITCPServerCallback) updating;`

The **initialize** method of the **JRFTCPServer** class sets the server agent and instance specified by the respective **serverAgent** and **tcpServerCallback** parameters that implements the **JITCPServerCallback** interface.

You can subsequently change the **tcpServerCallback** callback instance, by calling the **setTCPServerCallback** method.

open

Signature `open() updating;`

The **open** method of the **JRFTCPServer** class causes the server to listen for client TCP/IP connections.

The connection must have initialized using the **initialize** method.

setDirectApply

Signature setDirectApply(option: Boolean) updating;

The **setDirectApply** method of the **JRFTCPServer** class specifies that replication changes are to be applied immediately, if the value of the **option** parameter is **true**.

If the value of the **option** parameter is **false**, replication changes are saved to the persistent input queue and are applied when the server agent method **processQueuedTransactions** is called.

If the **setDirectApply** method is never called, replication changes are applied immediately.

setOptions

Signature setOptions(hostName: String;
 portNumber: Integer;
 timeout: Integer;
 minimumWorkers: Integer;
 maximumWorkers: Integer;
 logFileName: String) updating;

The **setOptions** method of the **JRFTCPServer** class sets TCP/IP options for the replication server, which listens for connections from clients. The options are used when the **open** method is called.

The parameters and their default values for this method are listed in the following table.

Parameter	Description	Default
hostName	Host name or IP address of the server	"localhost"
portNumber	Port number of the server	9999
timeout	Time in milliseconds before a read or write operation times out	600000
minimumWorkers	Minimum number of server processes to handle incoming client TCP/IP connections	1
maximumWorkers	Maximum number of server processes to handle incoming client TCP/IP connections	1
logFileName	File containing connection error messages	""

The default values in this table apply if a value is not specified in the initialization file or by calling the **setOptions** method.

The parameter values specified in the **setOptions** method override the corresponding parameter values in the [JRFCOMMS] section of the JADE initialization file. For details, see “[JRFCOMMS] Section of the JADE Initialization File”, in the following section.

Note The **timeout** parameter for the **setOptions** method is in milliseconds, whereas the **timeout** parameter in the [JRFCOMMS] section of the JADE initialization file is in seconds.

setTCPServerCallback

Signature setTCPServerCallback(tcpServerCallback: JITCPServerCallback) updating;

The **tcpServerCallback** parameter of the **JRFTCPServer** class **setTCPServerCallback** method specifies the callback instance that overrides the value specified in the **initialize** method, if required.

The value of the **tcpClientCallback** parameter must be an instance of a class that implements the **JITCPServerCallback** interface.

[JRFCOMMS] Section of the JADE Initialization File

The [JRFCOMMS] section of the JADE initialization file contains parameters that configure a TCP/IP connection between a replication server and its replication clients when server and clients use the **ReplicationBasicCommsPackage** exported from the **JadeReplicationSchema**.

The parameters are read by the server when the **JRFTCPServer** instance is created and by the clients when the **JRFTCPClient** instance is created.

The [JRFCOMMS] section can contain the following parameters.

HostName

Value Type String **Default** localhost

Purpose

The **HostName** parameter specifies the host name or IP address of the replication server.

Port

Value Type Integer **Default** 9999

Purpose

The **Port** parameter specifies the port number on which the replication server listens for connections from replication clients.

Timeout

Value Type Integer **Default** 600

Purpose

The **Timeout** parameter specifies the time in seconds before a read or write operation times out.

MinimumInUse

Value Type Integer **Default** 1

Purpose

The **MinimumInUse** parameter specifies the minimum number of server processes to handle incoming client TCP/IP connections.

MaximumInUse

Value Type Integer **Default** 1

Purpose

The **MaximumInUse** parameter specifies the maximum number of server processes to handle incoming client TCP/IP connections.

This chapter covers the following topics.

- **Overview**
- **Coding Examples Using the JADE Replication Framework**
 - [Registering a Client](#)
 - [Registering the Server](#)
 - [Registering Classes and Properties for Replication of Transactions on the Server](#)
 - [Registering Classes and Properties for Replication of Transactions on a Client](#)
 - [Capturing a Transaction for Replication](#)
 - [Registering a Transfer Receiver on a Client](#)
 - [Replicating Transactions from the Server to a Client](#)
 - [Replicating Transactions from a Client to the Server](#)
 - [Registering Classes on the Server for an Initial Load of Objects to a Client](#)
 - [Requesting an Initial Load of Objects from the Server to a Client](#)

Overview

You can automatically replicate user logic changes to an object by using the JADE Replication Framework.

Any class whose objects are to be replicated between systems must implement the **JIREplicable** interface on the client and the server. Before any replicable transaction activity, you must call the client and server agent **replicateProperty** or **replicateAllProperties** methods for each property whose changes are to be replicated.

For individual property changes, replication activity is per-property not per-object.

If one or more clients have different class or property names for corresponding server items, before any replicated queues are sent to the client or applied on the server, you must call the server agent **getMapClass** method and the **ReplicateMapClass** class **mapProperty** method for each replicable class and property whose names are different, to register these names with the framework.

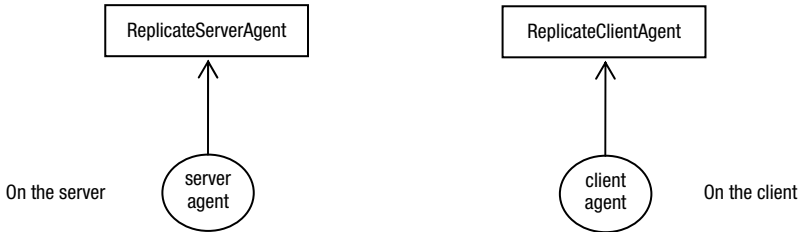
All transactions to be replicated must have tracing enabled, by using the client and server agent `enableTracing` method before the `beginTransaction` instruction is executed.

If the `replicateToClient` method is called on the server for a replicable class, it must return `true` for target clients. If the `replicateToNominatedClients` method is called, it must return `true` for all clients to be targeted or it must return `false`, in which case the client identifiers and group identifiers added to the `ids` parameter are targeted.

Coding Examples Using the JADE Replication Framework

The following subsections provide code fragments for typical activities within the JADE Replication Framework.

The important object on the server that drives replication activities is the *server agent*, an instance of the `ReplicateServerAgent` class, on the server. The corresponding object on the client is the client agent, an instance of the `ReplicateClientAgent` class.



In a simple design, you could access the client agent or server agent instance from a reference on the `app` environmental object.

In a more-sophisticated design, you could define a `ReplicationManager` class to encapsulate access to the agent instance.

Registering a Client

When the client application initializes, a transient instance of the JADE Replication Framework is created and the `registerAsClient` method called to create the client agent instance that drives replication activities.

In the following code fragment, the system in which the server and client participate for replication is called **replication system**, the client identifier of the client is **client**, and the server identifier of the server is **server**.

```
// replicationFramework is an instance of ReplicateFramework
create replicationFramework transient;
// clientAgent is an instance of ReplicateClientAgent
clientAgent := replicationFramework.registerAsClient("replication system",
                                                    "client",
                                                    "server",
                                                    currentSchema,
                                                    appContext);
```

Registering the Server

When the server application initializes, a transient instance of the JADE Replication Framework is created and the `registerAsServer` method called to create the server agent that drives replication activities.

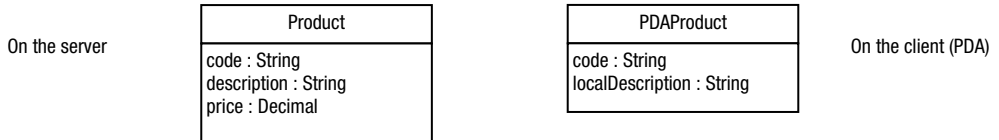
The last line of code in the following fragment uses the `registerClient` method to register the client with the server.

```
// replicationFramework is an instance of ReplicateFramework
create replicationFramework transient;
// serverAgent is an instance of ReplicateServerAgent
serverAgent := replicationFramework.registerAsServer("replication system",
                                                    "server",
                                                    currentSchema,
                                                    appContext);
serverAgent.registerClient("client", "replication system");
```

Registering Classes and Properties for Replication of Transactions on the Server

The server can register individual properties for replication by using the `replicateProperty` method or all the properties in a class by using the `replicateAllProperties` method.

The following diagram shows the **Product** class on the server and the corresponding **PDAProduct** class on the client. The classes and properties have different names on the client and the server.



In the following code fragment, two properties of the **Product** class are registered for replication.

```
serverAgent.replicateProperty(Product::description);
serverAgent.replicateProperty(Product::code);
```

An instance of the `ReplicateMapClass` establishes mappings between entities that are named differently on the client and server, as shown in the following code fragment.

```
replicateMapClass := serverAgent.getMapClass(Product,
                                             "replication system",
                                             "PDAProduct");
replicateMapClass.mapProperty(Product::description, "localDescription");
```

Registering Classes and Properties for Replication of Transactions on a Client

Classes and properties are registered for replication on the client in the same way as on the server.

You do not have to specify the mapping of differently named entities. as this mapping is required on the server only.

```
clientAgent.replicateAllProperties(PDAProduct, PDAProduct);
```

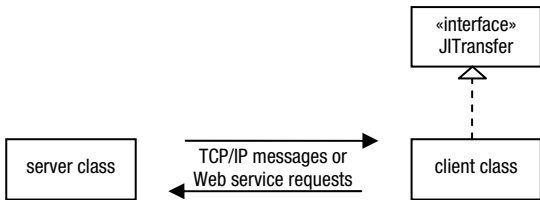
Capturing a Transaction for Replication

For a transaction to be captured, you must turn tracing on. In the following code fragment, the `code` property of a product is changed.

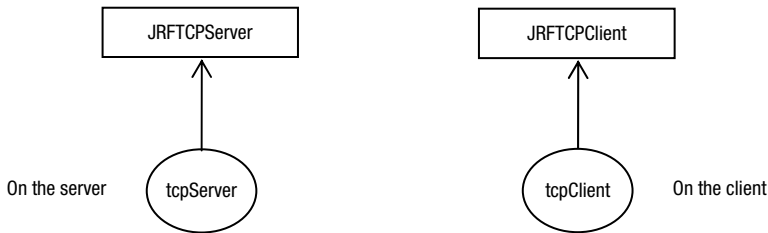
```
serverAgent.enableTracing; // or clientAgent on the client
beginTransaction;
product.code := "XX123";
commitTransaction;
serverAgent.disableTracing; // or clientAgent on the client
```

Registering a Transfer Receiver on a Client

When replication activities occur, messages are passed between the client and server using TCP/IP, Web services, or some other mechanism.



An instance of a class on the client that implements the **JITransfer** interface must be created and registered to receive transfer callback messages. The **ReplicationBasicCommsPackage** provides the **JRFTCPClient** and **JRFTCPServer** classes, which are ready-made communication classes using the TCP/IP communications protocol.



The following code fragments initialize and open a connection between the server and client.

```
// On the server
// tcpServer is an instance of JRFTCPServer
create tcpServer transient;
tcpServer.initialize(serverAgent, self);
tcpServer.open;

// On the client
// tcpClient is an instance of JRFTCPClient
create tcpClient transient;
tcpClient.initialize(clientAgent, self);
tcpClient.open;
// Register the callback receiver for replication data
clientAgent.registerTransferReceiver(tcpClient.JITransfer);
```

Replicating Transactions from the Server to a Client

The following code fragment on the client initiates replication.

```
clientAgent.replicateClientFromServer;
```

This code fragment calls back to the registered **JITransfer** instance **receiveFromServer** method, which uses a communication mechanism such as Web service consumer or the TCP/IP communications protocol to pass the following **message** parameter to server code.

```
response := serverAgent.replicateServerToClient(message);
```

After executing this code fragment on the server, the response is returned to the client by using the communications mechanism and a response is returned from the **JITransfer** instance **sendToServer** method.

The **replicateClientFromServer** method also calls back to the registered **JITransfer** instance **sendResponse** method, which uses the communication mechanism to pass the following **message** parameter to server code.

```
response := serverAgent.processResponse(message);
```

After executing this code fragment on the server, the response is returned to the client by using the communication mechanism and returned from the **JITransfer** instance **sendResponse** method.

Replicating Transactions from a Client to the Server

The following code fragment on the client initiates replication.

```
clientAgent.replicateClientToServer;
```

This code fragment calls back to the registered **JITransfer** instance **sendToServer** method, which uses the communication mechanism to pass the following **message** parameter to server code.

```
response := serverAgent.replicateServerFromClient(message);
```

After executing this code fragment on the server, the response is returned to the client using the communications mechanism and a response is returned from the **JITransfer** instance **sendToServer** method.

Registering Classes on the Server for an Initial Load of Objects to a Client

The registration of classes for an initial load of objects is separate from the registration for replication of captured transactions.

The following code fragment registers the **Product** class for loading.

```
replicateLoadClass := serverAgent.registerLoadClass(Product);  
replicateLoadClass.registerAllPropertiesUpTo(Product);
```

Requesting an Initial Load of Objects from the Server to a Client

The client requests the load action with the following code fragment.

```
tcpClient.JITransfer.requestAction("client", "0");  
clientAgent.replicateClientFromServer(true);
```

This invokes the following code on the server, which calls the `replicateLoad` method for each server object to be loaded to the client and the `replicateLoadTransaction` method to split the load into a number of *transactions*.

```
serverAgent.beginReplicateLoad("client");
beginTransaction;
foreach obj in <collection of objects to be loaded> do
    serverAgent.replicateLoad(obj);
    serverAgent.replicateLoadTransaction;
endforeach;
commitTransaction;
serverAgent.endReplicateLoad;
```

Index

A

- abortDBTransaction method, 23
- actionRequested method, 74
- addClientGroup method, 30
- addClientToGroup method, 30
- additional processing during replication callback, 23-26, 36-37
- addOidForClient method, 65
- addOidMap method, 31
- addToUniqueIds method, 23
- addUserMessage method, 31
- agent
 - client replication, 11, 28
 - server replication, 11, 28
- applications
 - developing Replication Framework, 19
- appliedTransaction method, 20
- applyAction property, 52
- applying transactions
 - captured on a client, 16
 - captured on the server, 16
 - conflicts when, 16, 31-32, 35
- applyTransaction property, 52

B

- beginDBTransaction method, 24
- beginReplicateLoad method, 65

C

- callback
 - additional processing during replication, 23-26, 36-37
 - postApply processing, 23-24, 33, 36-38
 - preApply processing, 23-25, 33, 36, 39
 - replication progress, 20
 - replication target, 21-22, 65
 - replication transfer, 26-27
 - resolve conflict, 23-25
- captured transactions
 - applying, 16
 - queue, 15-16, 33-35
 - replication of, 10-14, 85
- class
 - JRFTCPClient, 75-77
 - JRFTCPServer, 78

class (*continued*)

- ReplicateAgent, 28-29
- ReplicateAgent, 28
- ReplicateApplyObject, 41-42
- ReplicateApplyTransaction, 45-46
- ReplicateClientAgent, 46-47, 56
- ReplicateConflictEntry, 48
- ReplicateConflictObject, 50-51
- ReplicateFramework, 54
- ReplicateLoadClass, 56
- ReplicateMapClass, 57
- ReplicateQueue, 58
- ReplicateQueueEntry, 59
- ReplicateQueueTransaction, 61
- ReplicateServerAgent, 63

class names

- mapping, 11-12

class structure

- flattening the, 13

classes

- mapping names of, 11-12
- registering for replication, 84

className property, 52

clearConflicts method, 31

client

- registering for replication, 83
- targeted for replication, 11

client agent, 11, 28

client for replication, 11

client groups, 13, 30-32, 37

clientHasReplicableObject method, 65

clientId property, 52

commitDBTransaction method, 24

concepts of the Replication Framework, 10

configuring the JADE Replication Framework, 80

conflicts when applying transactions, 16

conflictType property, 53

considerations for the Replication Framework, 82

constants

- ReplicateAgent class, 28

- ReplicateConflictObject class, 51

correctedObject property, 53

D

deferred input queue, 15-16

deletions that are not replicated, 15
deregisterClient method, 65
deserializedTransaction method, 20
disableTracing method, 31
doFrameworkLocks method, 32

E

enableTracing method, 32
endReplicateLoad method, 66
exceptionObject property, 53

F

files
 jade.ini, 80
finalize method, 76-78
flattening the class structure, 13
Framework
 concepts of the Replication, 10
 developing applications for the Replication, 19
 overview of the Replication, 10
 runtime considerations for the Replication, 82

G

getActivityDetails method, 49
getCallReplToNominatedClients method, 66
getClassName method, 42, 59
getClientGroupDescription method, 32
getClientGroupIds method, 32
getClientId method, 47
getClientIds method, 66
getClientsInGroup method, 32
getConflicts method, 32
getCurrentQueue method, 33
getCurrentReplicateQueueSize method, 33
getDeferredInputQueue method, 47, 66
getDirectApply method, 78
getEntries method, 46, 62
getExceptionDetails method, 49
getIds method, 50, 58
getInitialClientIds method, 66
getInitialOidEqualityMode method, 66
getInlineQueueRemovalMode method, 67
getInstance method, 42
getLastProcessedClientIds method, 67
getLastTransferredClientIds method, 67
getLoadQueue method, 67
getLocalId method, 42, 60
getMapClass method, 67
getMaximumTransferSize method, 33
getObjectForId method, 24
getOidsForClient method, 62

getOperation method, 43, 60
getOperationForProperty method, 43
getOperationForPropertyAt method, 43
getOriginatorId method, 62
getPostapplyMode method, 33
getPreapplyMode method, 33
getPropertyCount method, 60
getPropertyNameAt method, 60
getPropertyNames method, 44
getPropertyOperationAt method, 60
getPropertyValueAt method, 61
getQueueName method, 58
getQueues method, 68
getQueueType method, 59
getReplicateQueueSize method, 47
getSequence method, 46, 62
getServerId method, 68
getStartTime method, 46
getTargetClientIds method, 62
getTotalReplicateQueueSize method, 68
getTracingMode method, 34
getTransactionId method, 46, 63
getTransactions method, 59
getTransactionStartTime method, 63
getTransferPerTransaction method, 34
getUniqueId method, 22
getUserClass method, 44
getUserLogLevel method, 34
getUserText method, 50
getValueCountForProperty method, 44
getValueForProperty method, 44
getValueForPropertyAt method, 45

H

handleUserMessage method, 74
hasChangeForProperty method, 45
hasTransactionTargetClients method, 63
HostName parameter, 80

I

initialize method, 76-78
initializeFramework method, 54
initializing JADE, 80
interface
 JIPProgressCallback, 20
 JIReplicable, 12, 21-22
 JIReplicateControl, 23
 JITCPClientCallback, 73
 JITCPServerCallback, 74
 JITTransfer, 26-27
invalidObject property, 53
isInReplication method, 35

J

JADE

- configuring, 80
- initializing, 80

JADE Replication Framework

- concepts of the, 10
- developing applications for the, 19
- overview of the, 10
- runtime considerations for the, 82

jade.ini file, 80

JadeReplicationSchema schema, 10, 19, 72

JIProgressCallback interface

- methods, 20
- overview, 20

JIReplicable interface

- methods, 22
- overview, 21

JIReplicateControl interface

- methods, 23
- overview, 23

JITCPClientCallback interface

- methods, 73
- overview, 73

JITCPServerCallback interface

- methods, 74
- overview, 74

JITransfer interface

- methods, 27
- overview, 26

JRFTCPClient class

- methods, 75
- overview, 75-77

JRFTCPServer class methods, 78

L

limitations of the Replication Framework, 17

load objects queue, 15

load of objects, 10-11, 15, 86

local deletes, 15

LogFileName parameter, 81

M

mapping

- class names, 11-12
- objects on client and server, 14, 31, 37
- property names, 11-12

mapProperty method, 57

MaximumInUse parameter, 80

message

- non-replication, 17, 31, 37

method

- abortDBTransaction, 23
- actionRequested, 74
- addClientGroup, 30
- addClientToGroup, 30
- addOidForClient, 65
- addOidMap, 31
- addToUniqueIds, 23
- addUserMessage, 31
- appliedTransaction, 20
- beginDBTransaction, 74
- beginReplicateLoad, 65
- clearConflicts, 31
- clientHasReplicableObject, 65
- commitDBTransaction, 24
- deregisterClient, 65
- deserializedTransaction, 20
- disableTracing, 31
- doFrameworkLocks, 32
- enableTracing, 32
- endReplicateLoad, 66
- finalize, 76-78
- getActivityDetails, 49
- getCallReplToNominatedClients, 66
- getClassName, 42, 59
- getClientGroupDescription, 32
- getClientGroupIds, 32
- getClientId, 47
- getClientIds, 66
- getClientsInGroup, 32
- getConflicts, 32
- getCurrentQueue, 33
- getCurrentReplicateQueueSize, 33
- getDeferredInputQueue, 47, 66
- getDirectApply, 78
- getEntries, 46, 62
- getExceptionDetails, 49
- getIds, 50, 58
- getInitialClientIds, 66
- getInitialOidEqualityMode, 66
- getInlineQueueRemovalMode, 67
- getInstance, 42
- getLastProcessedClientIds, 67
- getLastTransferredClientIds, 67
- getLoadQueue, 67
- getLocalId, 42, 60
- getMapClass, 67
- getMaximumTransferSize, 33
- getObjectForId, 24
- getOidsForClient, 62
- getOperation, 43, 60
- getOperationForProperty, 43

method (*continued*)

- getOperationForPropertyAt, 43
- getOriginatorId, 62
- getPostapplyMode, 33
- getPreapplyMode, 33
- getPropertyCount, 60
- getPropertyNameAt, 60
- getPropertyNames, 44
- getPropertyOperationAt, 60
- getPropertyValueAt, 61
- getQueueName, 58
- getQueues, 68
- getQueueType, 59
- getReplicateQueueSize, 47
- getSequence, 46, 62
- getServerId, 68
- getStartTime, 46
- getTargetClientIds, 62
- getTotalReplicateQueueSize, 68
- getTracingMode, 34
- getTransactionId, 46, 63
- getTransactions, 59
- getTransactionStartTime, 63
- getTransferPerTransaction, 34
- getUniqueId, 22
- getUserClass, 44
- getUserLogLevel, 34
- getUserText, 50
- getValueCountForProperty, 44
- getValueForProperty, 44
- getValueForPropertyAt, 45
- handleUserMessage, 74
- hasChangeForProperty, 45
- hasTransactionTargetClients, 63
- initialize, 76-78
- initializeFramework, 54
- isInReplication, 35
- mapProperty, 57
- open, 76-78
- postApply, 24
- preApply, 25
- processInput, 73-75
- processOutput, 73-75
- processQueuedTransactions, 47, 68
- processResponse, 35
- purgeCurrentQueue, 35
- purgeDetailsForClient, 68
- receiveFromServer, 27
- registerAllPropertiesUpTo, 56
- registerAsClient, 55
- registerAsServer, 55
- registerClient, 68
- registerConflictReceiver, 35

method (*continued*)

- registerIdController, 36
- registerLoadClass, 69
- registerPostapplyReceiver, 36
- registerPreapplyReceiver, 36
- registerProgressReceiver, 36
- registerProperty, 57
- registerTransactionReceiver, 36
- registerTransferReceiver, 48
- registerUserLogReceiver, 37
- registerUserMessageReceiver, 37
- removeClientFromGroup, 37
- removeClientGroup, 37
- removeOidMap, 37
- removeQueueEntry, 69
- removeSentQueueEntries, 69
- removeSystem, 56
- replicateAllProperties, 38
- replicateClientFromServer, 48
- replicateClientToServer, 48
- replicateLoad, 69
- replicateLoadTransaction, 69
- replicateProperty, 38
- replicateServerFromClient, 70
- replicateServerToClient, 70
- replicateToClient, 22, 65
- replicateToNominatedClients, 22
- requestAction, 27
- resolveConflict, 25
- sendResponse, 27
- sendToServer, 27
- sendUserMessage, 76
- serializedTransaction, 21
- setCallReplToNominatedClients, 70
- setDirectApply, 79
- setInitialClientIds, 70
- setInitialOidEqualityMode, 70
- setInlineQueueRemovalMode, 71
- setMaximumTransferSize, 38
- setOptions, 77-79
- setPostapplyMode, 38
- setPreapplyMode, 39
- setSyncPointForClient, 71
- setTCPClientCallback, 77
- setTCPServerCallback, 79
- setTracingMode, 39
- setTransactionTargetClients, 40
- setTransferPerTransaction, 40
- setUserErrorMessage, 40
- setUserLogLevel, 40
- splitMessageQueue, 59, 71
- unregisterProperty, 57
- unreplicateProperty, 41

method (*continued*)

- userLog, 26
- userMessage, 26

methods

- JJProgressCallback interface, 20
 - JJReplicable interface, 22
 - JJReplicateControl interface, 23
 - JITCPClientCallback interface, 73
 - JITCPServerCallback interface, 74
 - JITTransfer interface, 27
 - JRFTCPClient class, 75
 - JRFTCPServer class, 78
 - ReplicateAgent class, 29
 - ReplicateApplyObject class, 42
 - ReplicateApplyTransaction class, 46
 - ReplicateClientAgent class, 47
 - ReplicateConflictEntry class, 48
 - ReplicateFramework class, 54
 - ReplicateLoadClass class, 56
 - ReplicateMapClass class, 57
 - ReplicateQueue class, 58
 - ReplicateQueueEntry class, 59
 - ReplicateQueueTransaction class, 61
 - ReplicateServerAgent class, 63
- MinimumInUse parameter, 80

N

non-replication message, 17

O

- object identity mapping, 14
- oidValue property, 53
- open method, 76-78
- overview of the Replication Framework, 10

P

- package
 - ReplicationBasicCommsPackage, 10, 19, 72
 - ReplicationPackage, 10, 19, 72
- parameter
 - HostName, 80
 - LogFileName, 81
 - MaximumInUse, 80
 - MinimumInUse, 80
 - Port, 80
 - Timeout, 80
- parentObject property, 54
- Personal Digital Assistant (PDA), 10
- Port parameter, 80
- postApply method, 24
- postApply processing callback, 16, 23-24, 33, 36-38, 41

- preApply method, 25
- preApply processing callback, 16, 23-25, 33, 36, 39-41
- processInput method, 73-75
- processOutput method, 73-75
- processQueuedTransactions method, 47, 68
- processResponse method, 35
- properties
 - registering for replication, 84
 - ReplicateConflictObject class, 51
- property
 - applyAction, 52
 - applyTransaction, 52
 - className, 52
 - clientId, 52
 - conflictType, 53
 - correctedObject, 53
 - exceptionObject, 53
 - invalidObject, 53
 - oidValue, 53
 - parentObject, 54
 - propertyName, 54
- property names mapping, 11-12
- propertyName property, 54
- purgeCurrentQueue method, 35
- purgeDetailsForClient method, 68

Q

- queue
 - captured transactions, 15-16, 33-35
 - load objects, 15
 - removing entries from, 15

R

- receiveFromServer method, 27
- registerAllPropertiesUpTo method, 56
- registerAsClient method, 55
- registerAsServer method, 55
- registerClient method, 68
- registerConflictReceiver method, 35
- registerIdController method, 36
- registering
 - classes for replication, 11-12, 84
 - client for replication, 83
 - properties for replication, 11-12, 41, 84
 - server for replication, 84
 - transfer receiver, 85
- registerLoadClass method, 69
- registerPostapplyReceiver method, 36
- registerPreapplyReceiver method, 36
- registerProgressReceiver method, 36
- registerProperty method, 57
- registerTransactionReceiver method, 36

- registerTransferReceiver method, 48
 - registerUserLogReceiver method, 37
 - registerUserMessageReceiver method, 37
 - removeClientFromGroup method, 37
 - removeClientGroup method, 37
 - removeOidMap method, 37
 - removeQueueEntry method, 69
 - removeSentQueueEntries method, 69
 - removeSystem method, 56
 - removing entries from a queue, 15
 - replicable objects, 12, 21
 - ReplicateAgent class
 - constants, 28
 - methods, 29
 - overview, 28
 - replicateAllProperties method, 38
 - ReplicateApplyObject class
 - methods, 42
 - overview, 41
 - ReplicateApplyTransaction class
 - methods, 46
 - overview, 45
 - ReplicateClientAgent class
 - methods, 47
 - overview, 46, 56
 - replicateClientFromServer method, 48
 - replicateClientToServer method, 48
 - ReplicateConflictEntry class
 - methods, 48
 - overview, 48
 - ReplicateConflictObject class
 - constants, 51
 - overview, 50
 - properties, 51
 - ReplicateFramework class
 - methods, 54
 - overview, 54
 - replicateLoad method, 69
 - ReplicateLoadClass class
 - methods, 56
 - replicateLoadTransaction method, 69
 - ReplicateMapClass class
 - methods, 57
 - overview, 57
 - replicateProperty method, 38
 - ReplicateQueue class
 - methods, 58
 - overview, 58
 - ReplicateQueueEntry class
 - methods, 59
 - overview, 59
 - ReplicateQueueTransaction class
 - methods, 61
 - ReplicateQueueTransaction class (*continued*)
 - overview, 61
 - ReplicateServerAgent class
 - methods, 63
 - overview, 63
 - replicateServerFromClient method, 70
 - replicateServerToClient method, 70
 - replicateToClient method, 22, 65
 - replicateToNominatedClients method, 22
 - replication
 - agent, 11
 - captured transactions, 10-14, 38, 41, 45
 - client, 11
 - errors, 16, 31-32, 35, 40
 - load, 10
 - server, 11
 - system, 11
 - targeted clients, 17, 40
 - transaction, 41
 - Replication Framework
 - concepts of the, 10
 - developing applications for the, 19
 - limitations, 17
 - overview of the, 10
 - runtime considerations for the, 82
 - using the, 82-87
 - replication load, 11, 15
 - replication of transactions, 41
 - replication progress callback, 20
 - replication target callback, 21-22, 65
 - replication transactions from client to server, 86
 - replication transactions from server to client, 86
 - replication transfer callback, 26-27
 - ReplicationBasicCommsPackage package, 10, 19, 72
 - ReplicationPackage package, 10, 19, 72
 - requestAction method, 27
 - resolve conflict callback, 23-25
 - resolveConflict method, 25
 - resolving conflicts during replication, 16, 25, 31-32, 35
 - runtime
 - considerations for the Replication Framework at, 82
- ## S
- schema
 - JadeReplicationSchema, 10, 19, 72
 - sendResponse method, 27
 - sendToServer method, 27
 - sendUserMessage method, 76
 - serializedTransaction method, 21
 - server
 - registering for replication, 84
 - server agent, 11, 28

- server for replication, 11
- setCallReplToNominatedClients method, 70
- setDirectApply method, 79
- setInitialClientIds method, 70
- setInitialOidEqualityMode method, 70
- setInlineQueueRemovalMode method, 71
- setMaximumTransferSize method, 38
- setOptions method, 77-79
- setPostapplyMode method, 38
- setPreapplyMode method, 39
- setSyncPointForClient method, 71
- setTCPClientCallback method, 77
- setTCPServerCallback method, 79
- setTracingMode method, 39
- setTransactionTargetClients method, 40
- setTransferPerTransaction method, 40
- setUserErrorText method, 40
- setUserLogLevel method, 40
- splitMessageQueue method, 59, 71
- system of replication participants, 11

T

- targeting clients for replication, 17, 40
- Timeout parameter, 80
- tracing transactions, 14, 17, 31-32, 39
- transactions
 - capturing for replication, 85
 - replicating from client to server, 86
 - replicating from server to client, 86
 - replication, 10-11
 - tracing, 14, 17, 31-32, 39

U

- unregisterProperty method, 57
- unreplicateProperty method, 41
- user message, 17, 31, 37
- userLog method, 26
- userMessage method, 26