



J A D E TM

Replication Framework

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 © 2010 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

JADE Replication Framework	3
Overview	3
Example Schemas.....	5
How to Run the Example	7
Starting	7
Server	7
Clients.....	9
Re-running the Examples	12
Description of Framework Activity in the Example Schemas.....	12
Common.....	12
Server	13
Clients.....	16
Message Flow for Replication	17
Queue Processing – Server	19
Queue Processing – Client	19
Queue Inquiry	19
Data Load	19
General Features.....	20
Transaction Capture	20
Object Equality	21
Conclusion	22

JADE Replication Framework

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 and to synchronize 'disconnected' JADE systems. It handles the activity common to all such activity, with user hooks for application-specific processing.

The base functionality provides:

- Persistent automatic capture of create, update, and delete object activity
- Selective class and property replication
- Replication via user mapping of activity that can be between physically different but logically similar JADE systems
- Replication between different JADE platforms; that is, ANSI, Unicode, or Compact JADE
- Conflict detection and handling on application of the captured activity
- Simple user Application Programming Interface (API)

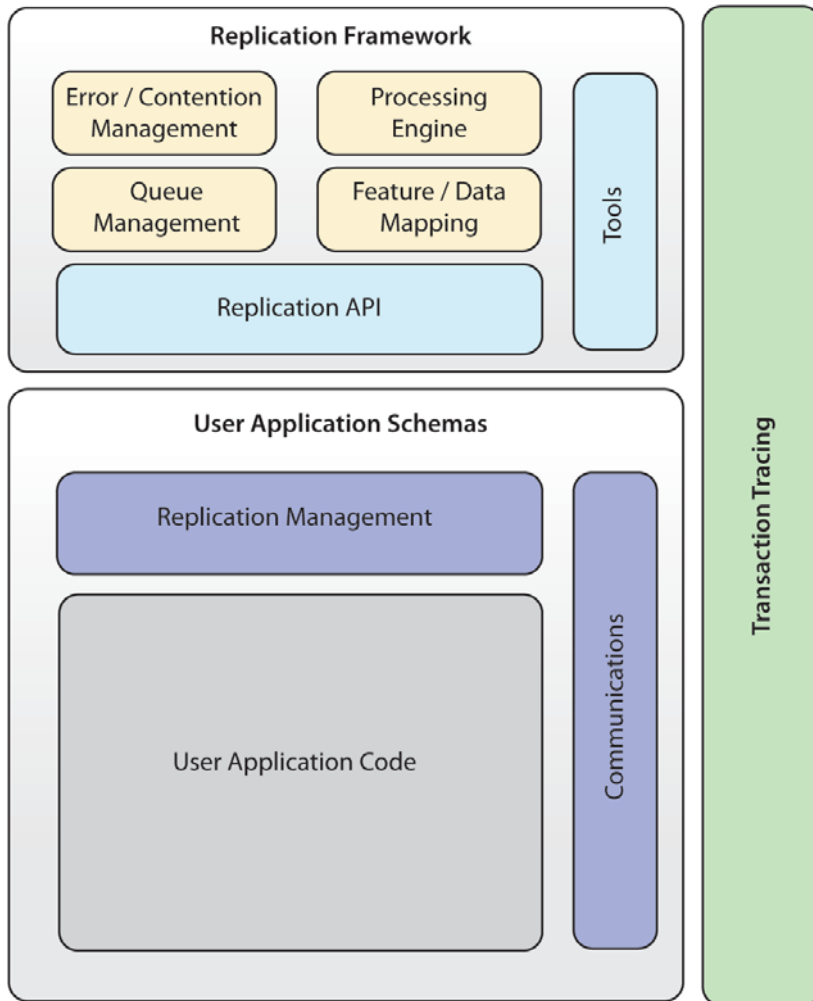
For details about the Replication Framework, see the *JADE Replication Framework User's Guide*, at <http://www.jade.co.nz/downloads/jade/manuals/Replication.pdf>.

Chapter 1 covers components of the framework, Chapter 2 the framework API, Chapter 3 the optional communications package API, and Chapter 4 some coding examples.

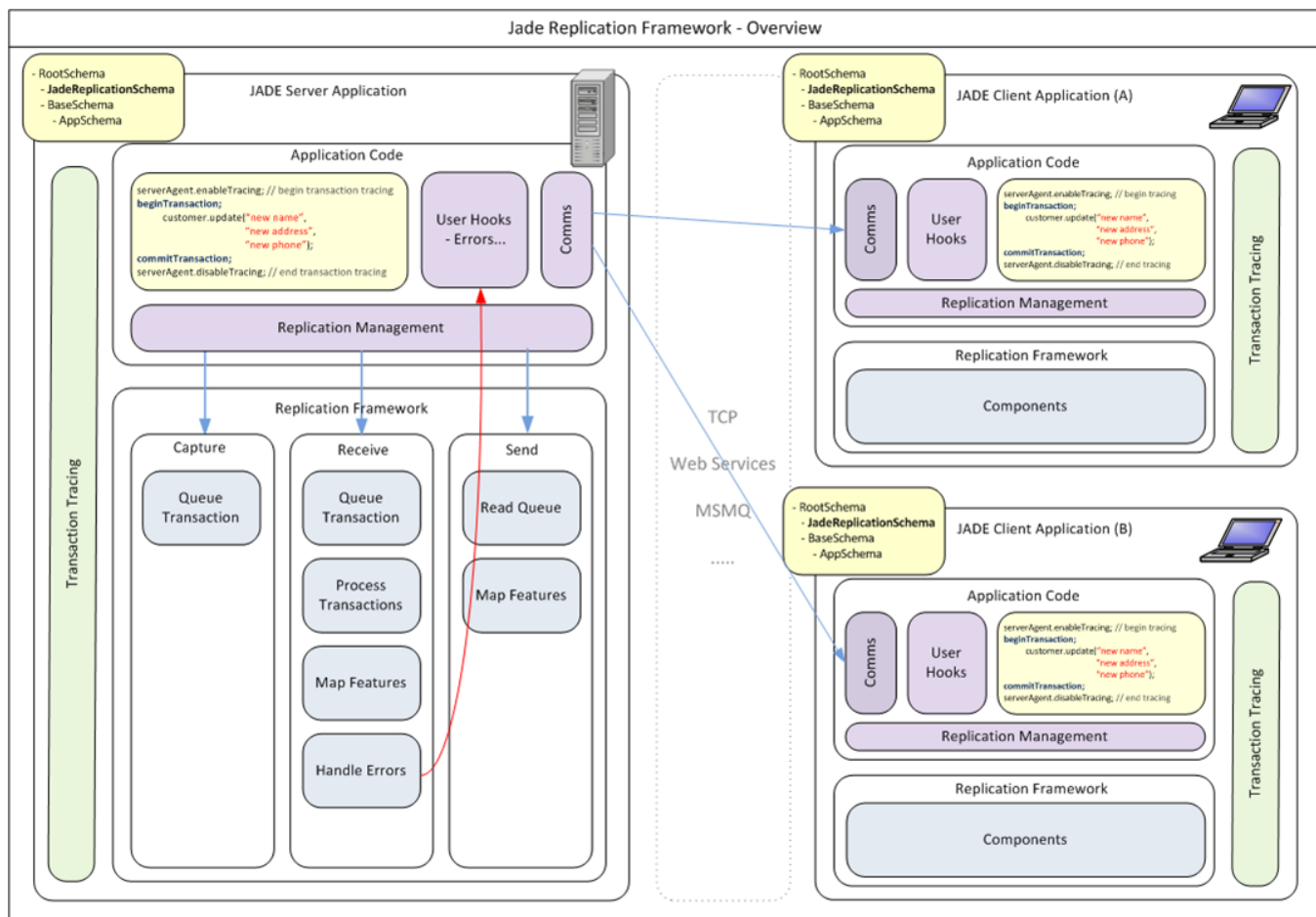
You should read this white paper in conjunction with the *JADE Replication Framework User's Guide*.

The Replication Framework is implemented by a package in a separate **JadeReplicationSchema** schema.

The package is imported into the participating user schemas and provides the functionality for each schema, as shown in the following diagram.



The JADE transaction tracing feature is used by the framework to capture user transactions for subsequent replication, as shown in the following diagram.



This diagram shows the interaction between the functionality of the framework within a server application and the participating client applications, of which two are shown. Captured and queued transactions are exchanged between the participating systems using an appropriate messaging system such as the suggested Transmission Control Protocol/Internet Protocol (TCP/IP), Web services, or Microsoft Message Queuing (MSMQ). This messaging needs to be written as part of the user systems, although there is a package available in the framework schema that implements a basic TCP/IP communications mechanism.

This white paper explains typical usage of the framework using sample schemas, including both user-written TCP/IP messaging and use of the framework communications package.

Example Schemas

Four schemas are provided, which represent two versions of a server and two remote client systems.

The schemas implement a simple application to control reading and repair of meters.

- The server schemas create new reading and repair/replace jobs for client schemas that can be thought of as running in separate mobile devices out in the field.

- The client schemas allow receipt of new jobs from the server. The status of these jobs is updated on the client device once they have been carried out and sent back to the server that then responds with any new jobs.

This client-server synchronizing is done on demand from each client.

The schemas use a TCP/IP connection to transfer replication changes between them, not a Web service, although a Web service could also be used just as easily. The TCP/IP connection is used for the purposes of the example, because it is more self-contained and the example applications can be run without the additional set-up that would be required for a Web service.

The **JobManagementMeterMaintClient** and **JobManagementServer** schemas both use the replication framework TCP/IP connection provided via the exposed **ReplicationBasicCommsPackage**. This requires the following JADE initialization file settings, with example parameter values that set the TCP/IP host and port used by both applications.

```
[JRFCComms]
HostName=localhost
Port=21000
ReadTimeout=30
```

These values default to **localhost**, **9999**, and **30** (seconds), respectively, if they are not specified.

The **JobManagementMeterReaderClient** and **JobManagementServerUserComms** schemas both use explicit TCP/IP connections instead of the built-in framework. However, this implementation formats the messages between server and client in the same way that the framework TCP/IP package does, so the schemas all interoperate. This requires the following JADE initialization file settings, with example parameter values that set the TCP host and port used by both applications.

```
[UserTCP]
HostName=localhost
Port=21000
ReadTimeout=30
```

These values default to **localhost**, **9999**, and **30** (seconds), respectively, if they are not specified.

To run all four schemas together (as shown in the example in the following section), ensure that both sets of JADE initialization file settings use the same host name and port numbers.

Two versions of the server schema are provided to show the alternative messaging transport mechanisms.

The **JobManagementServer** and **JobManagementMeterMaintClient** schemas use the exposed **ReplicationBasicCommsPackage**, which provides built-in sending and receiving of replication messages using TCP/IP. This communications package is easy to use but provides only the basic functionality required for replication.

The main functionality not directly supported is security, although there is provision for adding or removing headers to or from each message and encrypting or decrypting it in user code from the **processInput** and **processOutput** call-back methods.

The **JobManagementServerUserComms** and **JobManagementMeterReaderClient** schemas have a user-coded transport mechanism, which happens to use TCP/IP as well. However a Web service could also have been used, in which case the server **UserTCPServer::processInput** method would be replaced by separate Web service provider methods for each message type, each with explicit method parameters in place of the token decoding code in this method.

The client **UserTCPClient** implementations of the **JITransfer** interface methods would call these Web service methods passing the parameters explicitly instead of packaging them into the message string.

The provision for user-coded transport mechanisms is to allow for cases where there is a requirement for additional functionality over that provided in the **ReplicationBasicCommsPackage** and also where there is already a data transport mechanism between the server and clients which the Replication Framework is to use.

How to Run the Example

For convenience, the applications are all run from the one JADE node but their behavior is identical to that running in separate JADE nodes or host machines.

Starting

Run the **StartExample** application of the **JobManagementServer** or the **JobManagementServerUserComms** schema.

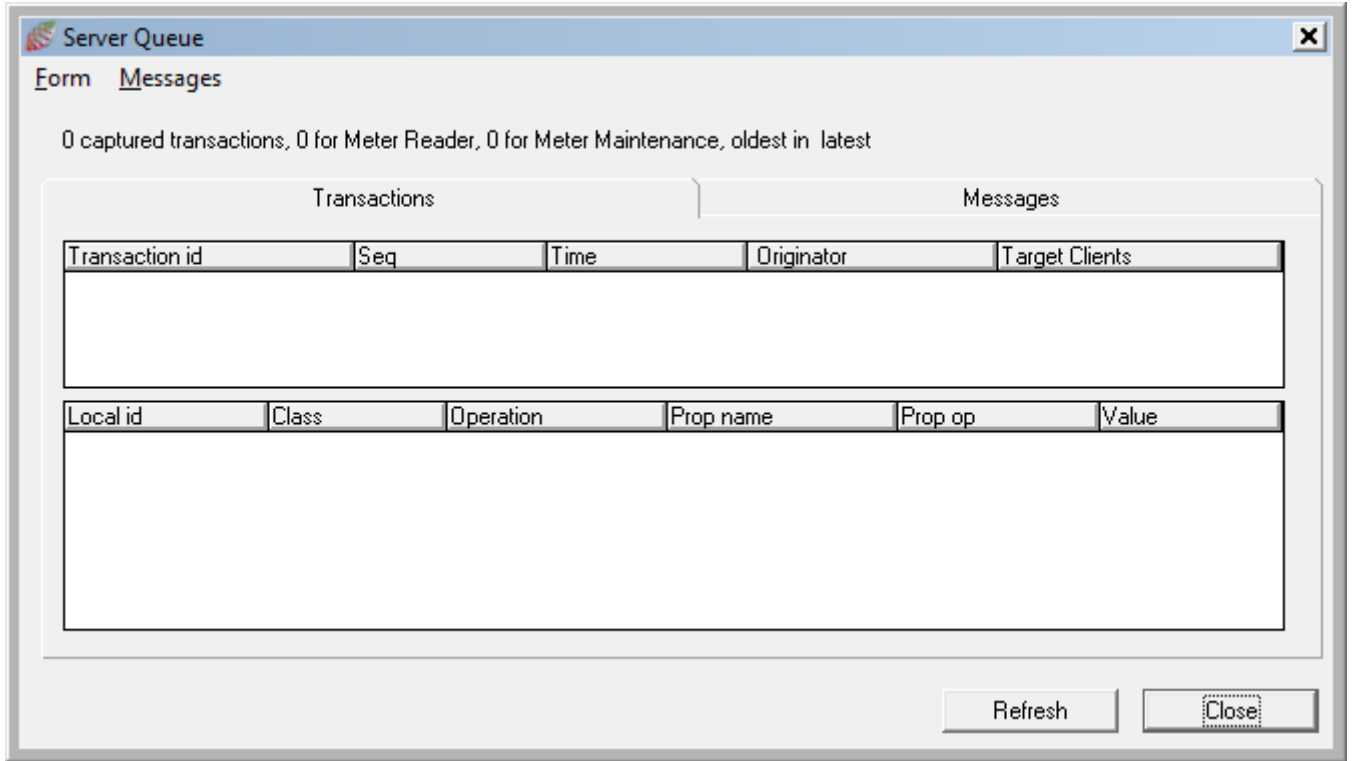
This uses the AppInit form to run the **ServerQueue** and **ServerJobAdministration** server applications, the client **MeterReader** application of the **JobManagementMeterReaderClient** schema, and the **MeterMaintenance** application of the **JobManagementMeterMaintClient** schema.

If the applications fail to start for any reason, typically because the schemas are not present, the AppInit form stays loaded with details of the runs; otherwise the **StartExample** application terminates and the forms for each application are displayed.

Server

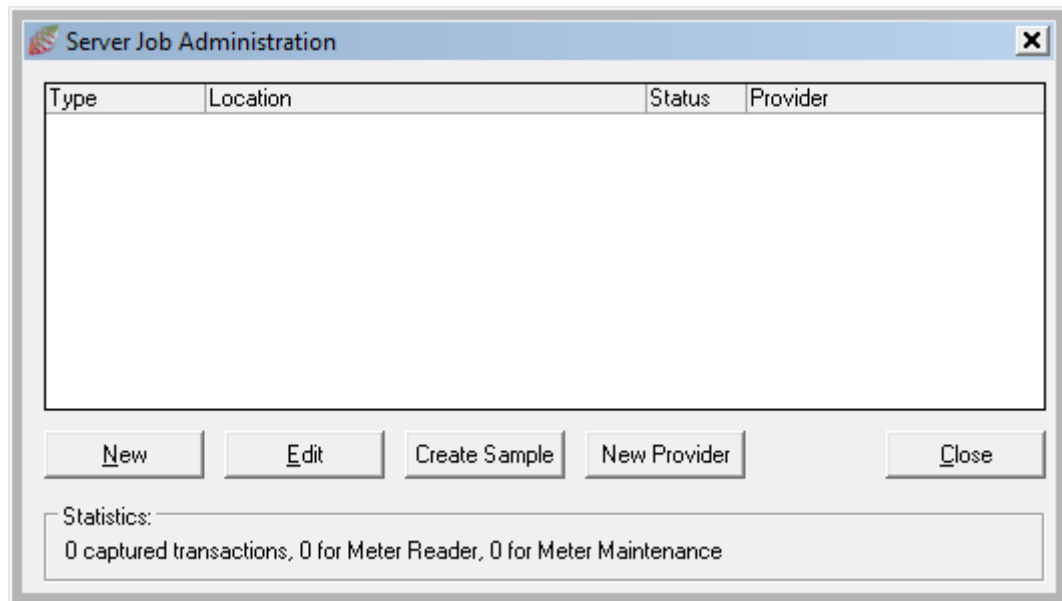
Two server forms are displayed.

The first server form is the Server Queue form, shown in the following diagram, which displays current replication queue entries on the **Transactions** tab and TCP/IP messages sent between the clients and server on the **Messages** tab.



On first startup, both client applications request an initial data load of the static data in the **JobStatusCode**, **JobTypeCode**, **ServiceProvider**, and **Site** classes. This activity can be seen in the **Messages** tab of this form.

The second server form is the Server Job Administration form, shown in the following diagram, which represents activity done on the server that is to be replicated using the Replication Framework to one or more client systems.



This activity is the creation of job requests for the clients of either meter reading or meter maintenance.

Server Activity

Clicking the **New** button on the Server Job Administration form brings up a Job Maintenance form. The type of activity that you can select is Read Meter, Repair Meter, or Replace Meter.

Activities of the first type are automatically queued for the Meter Reader client and those of the other two types are automatically queued for the Meter Maintenance client. Select the appropriate values for the other fields and then click the **OK** button, to create a new job request.

Alternatively, click the **Create Sample** button to create six new requests: four read, one repair, and one replace.

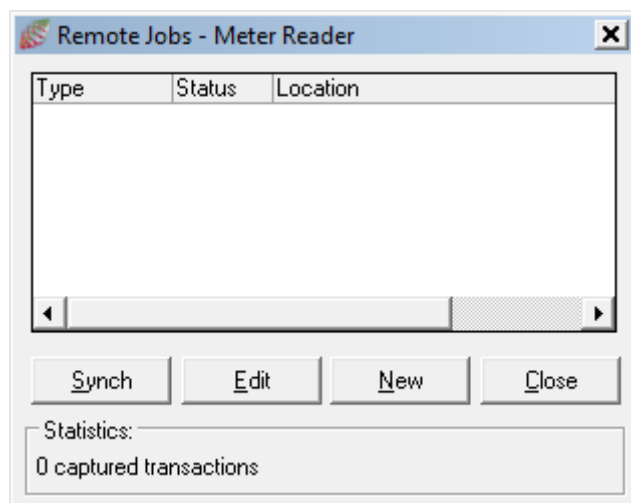
As jobs are created, the Server Job Administration form is refreshed to show the details.

These requests have also been automatically captured by the Replication Framework and can be seen by clicking the **Refresh** button on the Server Queue form. Selecting a line in the table at the top of the **Transaction** tab shows the individual per-property changes in the table at the bottom of this tab. These changes are what will be sent to the clients by the framework.

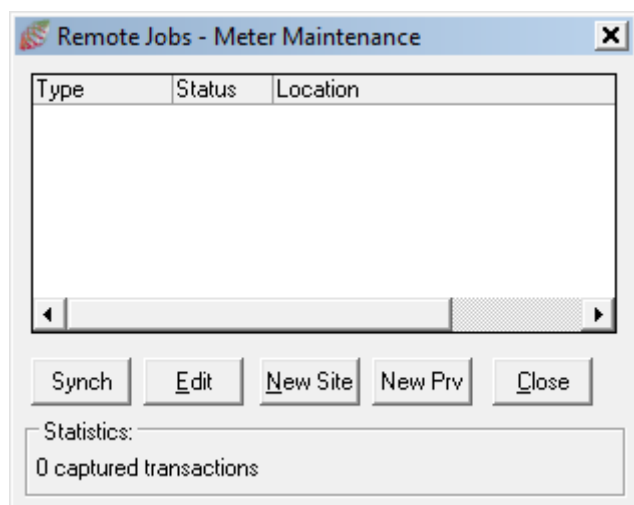
Clients

Two client forms are displayed.

The first client form is the Remote Jobs - Meter Reader form, shown in the following diagram.



The second client form is the Remote Jobs – Meter Maintenance form, shown in the following diagram.



Client Initial Synchronization

After creating some activity using the Server Job Administration form, click the **Synch** button on each of the client forms.

A Progress form is briefly displayed as the transfer between each client and server is in progress. The **Messages** tab of the Server Queue form is also updated to show the message traffic each way.

Each client form then shows the transferred server activity appropriate to it. For example, if the **Create Sample** button had been clicked on the Server Job Administration form, the Meter Reader client form will show four Read Meter entries and the Meter Maintenance client form will show one each for Repair Meter and Replace Meter.

Client Updates

To carry out the client activity:

1. Click on an entry and then the **Edit** button, or just double-click the entry. The maintenance form is then displayed.
2. Update the status to Complete.
3. Write some notes for each client.
4. Click the **OK** button to update the job request.

The Replication Framework will automatically capture these updates and queue them to be sent to the server on request.

5. Click the **Synch** button on each client.

In addition, you may first want to add more jobs on the server.

Each client first sends its changes to the server and then receives any server changes for it. The client forms are updated to drop sent jobs and to add new ones. The Server Job Administration form is also updated with the activity received from the clients.

To see the changes, click on an entry and then the **Edit** button, or just double-click an entry.

Client-to-Client Activity

Jobs can also be created on the Reader client for the Maintenance client. To do this:

1. Click the **New** button on the Remote Jobs – Meter Reader form.

Note If you select an existing entry first, the new job is based on this entry; for example, a reading that could not be done because the meter was faulty.

2. Add any details and then click the **OK** button, to create the new job.
3. Click the **Synch** button to add the job to the list on the server.
4. Click the **Synch** button on the Remote Jobs – Meter Maintenance form, to transfer the job to this client.

Note This activity is completely automatic when the job has been created on the reader client; there is no additional user application code required to make this happen.

The Maintenance client also includes actions that are handled on the server as user-coded updates from the captured and replicated client transactions, and that can cause conflicts on the server when replicated from this client.

Server User Updates

To cause the user-coded updates:

1. Click the **New Site** button on the Remote Jobs – Meter Maintenance form. The **New Site** form is then displayed.
2. Enter an address and then click the **OK** button. The Meter Maintenance form is refreshed to show a new entry with type **New Client** and the specified address.
3. Click the **Synch** button, to send this to the server.

On the server, the implementation of the preApply call-back creates a new **Site** instance from this action.

4. To see the new entry in the **Site** combo box on the Job Maintenance form, click the **New** button on the Server Job Administration form.
5. Click the **Cancel** button, to close this form.

Server Conflict

To illustrate the conflict-detection mechanism, cause a conflict with the server, as follows.

1. Click the **New Provider** button on the Server Job Administration form. The New Provider form is then displayed.
2. Enter a name and then click the **OK** button.
3. On the Remote Jobs – Meter Maintenance client form, click the **New Prv** button, to bring up the New Provider form.
4. Enter the same name, click the **OK** button, and then click the **Synch** button, to send this to the server.

The application of this action causes a duplicate key exception, as there is already a **Site** instance with this name just created on the server.

The conflict call-back method displays a message box showing the following.

```
Duplicate ServiceProvider from client apply. Select 'Yes' to keep
the server instance, 'No' to keep the client instance or 'Cancel'
to skip the transaction.
```

Clicking the **Yes** button causes the framework to skip the addition of the client, clicking the **No** button deletes the server **Site** instance and tells the framework to re-try the client transaction that then succeeds, and clicking the **Cancel** button causes the framework to skip the whole transaction. Note that in this case, the **Yes** and **Cancel** actions are in fact the same, as the transaction contains only the single **Site** addition.

Closing Down the Examples

Close all applications by simply clicking the **Close** button on each form.

Re-running the Examples

You can stop the applications and run them again at any point, and you can apply any captured but un-transferred transactions by clicking the **Synch** button of each client form.

To reset both the application data and Replication Framework status back to the initial state, run the **JadeScript::initialize** method of each schema.

Note Although two server schemas are supplied, run only one with the client applications. If both are run, the internal framework object-equality mapping of equivalent server and client objects will get out of step and replication of transactions will not work properly.

Description of Framework Activity in the Example Schemas

This section provides a description of Replication Framework activity in the example schemas.

Common

Both client and server schemas have a similar class structure. Both have a number of control classes under **ApplicationModel** and a number of persistent data classes under **DomainModel**.

The **DomainModel** classes should be largely self-explanatory. The only change required by the Replication Framework is the requirement to implement the **JIREplicable** interface in all classes whose persistent instances are to have activity automatically captured by the framework. This interface is implemented by the top-level **DomainClass**, which has default implementations.

Of the three methods in this interface, **getUniqueId** is reserved for future use. The **replicateToClient** and **replicateToNominatedClients** methods are alternative implementations of the same functionality. For details, see "Server", in the following section. Note that the client schemas require this interface to be used, but the methods are not called. The interface is used as a mechanism to mark classes as having their instance changes replicated. On the client, all changes are replicated to the server for the registered properties.

The **ApplicationModel** classes include **CodeManager** and **JobApplicationManager**, which each have a single persistent instance functioning as root objects to hold collections of the other persistent application instances.

The **ReplicationManager** class encapsulates the Replication Framework and has a singleton transient instance accessible on the **Application** class. All of the code to interact with the framework is in this class. Many of the methods just call into the framework via the defined **ReplicateServerAgent** or **ReplicateClientAgent**, as appropriate, and others are called from application initialization to do the initial set up. These are controlled from the **ReplicationManager::initialize** method. For details, see the following sections.

The **abortTracedTransaction**, **beginTracedTransaction**, and **commitTracedTransaction** methods are convenience methods to set the tracing option, begin the transaction, and reset the tracing option after an abort transaction or commit transaction, respectively. Transactions are not automatically captured by the Replication Framework, as not all transactions within an application may be relevant. Transaction capture is off by default, is enabled after a **ReplicateAgent::enableTracing** method call, and disabled after a **ReplicateAgent::disableTracing** method call.

Server

The following subsections describe Replication Framework server activity.

Initialize

The **ReplicationManager::initialize** method starts with the creation of a **ReplicationFramework** instance and a **registerAsServer** method call to get a **ReplicateServerAgent** instance.

The **specifyReplicableProperties** method is then called, to register all properties of all classes containing persistent data whose updates are to be replicated, using the agent **replicateProperty** method.

```
serverAgent.replicateProperty(JobStatusCode::codeValue);
serverAgent.replicateProperty(JobStatusCode::displayValue);
```

Updates are tracked per property by the framework. Object creates and deletes are tracked per instance.

If a property is not registered via the agent **replicateProperty** method before a user **commitTransaction** instruction of a change to a class instance with that property, the change will not be captured by the framework. This is something to watch when adding new properties to an existing class and using the Replication Framework. Note that you can use the agent **replicateAllProperties** method if updates to all properties are to be replicated.

The **defineMapping** method then registers mappings of class and property name differences between clients and the server, using the **ReplicateMapClass** class for class name mapping and the map class **mapProperty** method for individual property name mapping, as shown in the following code fragment.

```
mapClass := serverAgent.getMapClass(RemoteJob,
    "JobManagementMeterReaderClient", "ReadJob");
mapClass.mapProperty(RemoteJob::myJobStatus, "jobStatus");
mapClass.mapProperty(RemoteJob::myJobType, "jobType");
```

Mapping of client and server name differences is required only on the server. The example Meter Reader client schema has artificially different names to illustrate use of this, but in practice it may be useful to have differences in the names.

Another point to watch is ensuring that all classes and properties involved in replication have mappings registered where the names are different. On application of a replicated action on both client and server, if the class or property name associated with that action is not found in the current schema, the action is ignored and not currently logged.

The **defineReceivers** method registers some call-back receivers, as shown in the following code fragment.

```
create replicationReceiver transient;  
serverAgent.registerPreapplyReceiver(replicationReceiver);  
serverAgent.registerPostapplyReceiver(replicationReceiver);
```

These must be transient instances of a class that implements the **JIReplicateControl** interface. The call-backs are all optional and the interface methods are called only if the specific type of activity is registered.

In this case, receivers are registered for pre- and post-apply, conflicts, and transaction control in the **JobManagementServer** schema, and pre-apply and conflict control in the **JobManagementServerUserComms** schema.

Pre-apply, post-apply and conflict receivers are discussed later in this document. For the transaction control receiver, the **abortDBTransaction** method implementation must call **abortTransaction**, the **beginDBTransaction** method implementation must call **beginTransaction**, and the **commitDBTransaction** method implementation must call **commitTransaction**. This receiver would be used when the normal user transactions are done using methods that call **beginTransaction**, **commitTransaction**, and so on, but also do other processing, to ensure that this processing is also done when applying replicated actions.

Communications

The **JobManagementServer** schema uses the framework TCP/IP connectivity and does this via a local intermediate **TCPServerManager** class, which has an instance variable of type **ReplicationBasicCommsPackage::JRFTCPServer**, and the call-back methods implemented for the **ReplicationBasicCommsPackage::JITCPServerCallback** interface. The **initialize** method of this class does the set-up required to use this functionality.

The **JobManagementServerUserComms** schema uses its own TCP/IP connectivity and also does this via a local intermediate **UserTCPServerManager** class. This class uses an additional **UserTCPServer** class to handle receiving and sending TCP/IP messages to and from the client.

The **UserTCPServer** class uses **JadeMultiWorkerTcpConnection** as the connection mechanism, to obtain the additional functionality that this provides.

Replication

Replication is initiated from each client. The server needs to have an application running to receive client messages, typically via TCP/IP or a Web service.

In this example, the **ServerQueue** application of both the **JobManagementServer** and the **JobManagementServerUserComms** schemas do this, using the **TCPServerManager::initialize** method.

Client Targeting

The Server Job Administration forms of both server schemas illustrate transaction capture and client targeting using the three different available mechanisms.

In the **JobManagementServer** schema, the **ServerJobForm::handleCreateSample** method creates six remote jobs in one traced transaction.

In the commit processing by the framework, the **RemoteJob::replicateToClient** method is called for each new **RemoteJob** instance and then for each registered client within each object. The implementation returns **true** or **false**, based on the combination of **myJobType** and client identifier (id) so that meter read jobs are sent to the meter reader client and maintenance jobs are sent to the meter maintenance client.

In the following code fragment, objects within a single transaction can be split among several different clients so that each client gets part of the content of the transaction.

```

if myJobType.codeValue = "Read Meter" then
  if clientId = ReaderClientId then
    return true;
  elseif clientId = MaintenanceClientId then
    return false;
  endif;
else
  if clientId = ReaderClientId then
    return false;
  elseif clientId = MaintenanceClientId then
    return true;
  endif;
endif;
return false;

```

In the **JobManagementServerUserComms** schema, the **ServerJobForm::handleCreateSample** method creates six remote jobs in one traced transaction. Because the **ReplicationManager::initialize** method has called **serverAgent::setCallReplToNominatedClients(true)**, in the commit processing by the framework, the **RemoteJob::replicateToNominatedClients** method is called for each new **RemoteJob** instance. The implementation returns **false** to indicate that replication of each object is selective, and updates the id's **StringArray** parameter with the relevant client ids based on the **myJobType** so that meter read jobs are sent to the meter reader client and maintenance jobs are sent to the meter maintenance client.

In the following code fragment, objects within a single transaction can be split among several different clients so that each client gets part of the content of the transaction.

```

if myJobType.codeValue = "Read Meter" then
  ids.add(ReaderClientId);
else
  ids.add(MaintenanceClientId);
endif;

return false;

```

In this schema, the **ServerJobForm::handleCreateSample2** method also creates six remote jobs in two traced transactions. Each transaction is targeted to a list of client ids by the **ReplicateServerAgent::setTransactionTargetClients** (called via the same method in the **ReplicationManager** class).

The first transaction has only meter read jobs and is targeted at the Meter Reader client, and the second transaction has only meter maintenance jobs and is targeted at the Meter Maintenance client. For both transactions, neither the **JIREplicable** interface **replicateToClient** nor **replicateToNominatedClients** method is called on the **RemoteJob** instance from the commit processing.

Where the nature of the application is such that client targeting is at transaction level, the **ServerAgent::setTransactionTargetClients** method is a more-efficient mechanism for client targeting than the **JIREplicable** interface method call-backs.

Note also that it does not matter where in the transaction this method is called as long as it is before the **commitTransaction**, as it sets up the targeting information used when the transaction is committed. This also means that where there are multiple calls to this method in a transaction, the last one executed will be actioned, each overwriting the effect of the previous one. As the stored targeting information is transient and is cleared after each **commitTransaction**, it does not carry over into following transactions. This allows the **setTransactionTargetClients** mechanism to be inter-mixed with the **JIREplicable** interface call-back mechanism.

Clients

The following subsections describe Replication Framework client activity.

Initialize

The **ReplicationManager::initialize** method starts with the creation of a transient **ReplicateFramework** instance and a **registerAsClient** method call to get a **ReplicateClientAgent** instance.

The **specifyReplicableProperties** method is then called to register all properties of all classes containing persistent data whose changes are to be replicated, using the agent **replicateProperty** method. (For more details, see "Initialize" under "Server", earlier in this document.)

The **registerReceivers** method is called to register any call-back receivers. These must be transient instances of a class that implements the **JIREplicateControl** interface.

The **JobManagementMeterMaintClient** schema uses the framework TCP/IP connectivity via a local intermediate **TCPCClientManager** class, which has an instance variable of type **ReplicationBasicCommsPackage::JRFTCPClient** and the call-back methods implemented for the **ReplicationBasicCommsPackage JITCPClientCallback** interface. The **initialize** method of this class does the setup required to use this functionality.

The **JobManagementMeterReaderClient** schema uses its own TCP/IP connectivity via a local intermediate **UserTCPCClientManager** class. This class uses an additional **UserTCPCClient** class to handle receiving and sending TCP/IP messages to and from the server. The **UserTCPCClient** class uses **JadeInternetTCPIPConnection** as the connection mechanism.

Replication

Replication is initiated from each client, using the **ReplicationManager** class **requestSynchronize** method. Captured and queued client updates are sent first, by the **ReplicateClientAgent::replicateClientToServer** method. This uses the declared transport mechanism to send serialized captured changes to the server.

Note There is no inherent need for the client changes to be sent first; it depends on the specific requirements of the application. The assumption is that the client is given work to do by the server, does it, sends it back, and then gets the next lot of work from the server.

The client then requests captured and queued server updates by the **ReplicateClientAgent::replicateClientFromServer** method. The method parameter determines whether the changes are applied as they are received (for a parameter value of **true**) or they are queued persistently and applied by a subsequent call to the **ReplicateClientAgent::processQueuedTransactions** method (for a parameter value **false**).

Queuing transactions may be a desirable option if connection cost or time between server and mobile client are to be minimized for each replication.

Message Flow for Replication

Replication is initiated from the client and causes processing on the server. Methods marked with **(a)** are called if the framework communication package is used on the server, and methods marked with **(b)** are called if a user-implemented communications mechanism is implemented on the server.

Method calls that are conditional on call-back receivers having been registered or on a particular condition occurring are shown in parentheses.

The **ReplicateClientAgent::replicateClientToServer** method call results in an internal loop of:

Client	Server
(JIProgressCallback::serializedTransaction)	
JITransfer::sendToServer	
(ITCPClientCallback::processOutput)	
	(a) JRTCPServer internal message receipt method
	(b) Server message receipt method, which must call ReplicateServerAgent::replicateServerFromClient
	JITCPServerCallback::processInput
	(JIReplicateControl::preApply)
	(JIReplicateControl::resolveConflict)
	(JIReplicateControl::userMessage)
	(JIReplicateControl::postApply)
	(JIProgressCallback::appliedTransaction)
	(JIProgressCallback::serializedTransaction)
	JITCPServerCallback::processOutput
(JITCPClientCallback::processInput)	

This loop continues until all of the queued client transactions have been sent to the server. This is controlled by two client settings: the transfer-per-transaction option set by the **ReplicateClientAgent::setTransferPerTransaction** and the maximum transfer size, set by the **ReplicateClientAgent::setMaximumTransferSize** method. The default values for these are **false** and **500,000** bytes, respectively.

If the transfer-per-transaction option is set, each captured transaction is sent via a separate call to the **JITransfer::sendToServer** method; otherwise captured transactions are serialized until their total size equals or exceeds the maximum transfer size and then each such block is sent. Note that transactions are never split but are always transferred in their entirety.

The **ReplicateClientAgent::replicateClientFromServer** method results in an internal loop of:

Client	Server
JITransfer::receiveFromServer (JITCPClientCallback::processOutput)	(a) JRTCPServer internal message receipt method (b) Server message receipt method which must call ReplicateServerAgent class replicateServerToClient method
(JITCPClientCallback::processInput)	JITCPServerCallback::processInput
(JIProgressCallback::deserializedTransaction)	(JIProgressCallback::serializedTransaction)
(JIReplicateControl::preApply)	JITCPServerCallback::processOutput
(JIReplicateControl::resolveConflict)	
(JIReplicateControl::userMessage)	
(JIReplicateControl::postApply)	
(JIProgressCallback::appliedTransaction)	
JITransfer::sendResponse	
(JITCPClientCallback::processOutput)	
(JITCPClientCallback::processInput)	

This loop continues until all of the queued transactions on the server for this client have been sent. This is controlled by two server settings: the transfer-per-transaction option set by the **ReplicateServerAgent::setTransferPerTransaction** method and the maximum transfer size, set by the **ReplicateServerAgent::setMaximumTransferSize** method. The default values and processing of these options is the same as those of the **replicateClientToServer** method.

For methods on JADE interfaces, the actual method called will be the local method mapping for the class of the registered call-back receiver that implements the interface. Receiver registration methods are covered in the *JADE Replication Framework User's Guide*; for example, **JITransfer** receivers are registered by the **ReplicateClientAgent** class **registerTransferReceiver** method.

Queue Processing – Server

All captured user transactions go into a single server queue. Each transaction is identified by its transaction id (returned by the **Process::getTransactionId** method) and has a list of object identifiers (oids) to be sent to each target client. Each registered client on the server has an internal last transaction id sent, maintained by the framework.

When a **ReplicateServerAgent::replicateServerToClient** method is executed, the transaction id of the current end of the server queue is saved then the transactions in this server queue are serialized, starting with the first entry past the last transaction sent to this client and ending with the saved end transaction id.

Each transaction in the queue is checked to see if it has at least one action for the current client. If not, it is skipped; otherwise the actions for the client are serialized. At this point, the server oids are replaced with client oids from the client-server oid maps, and server class and property names are replaced by client names if they are different from any server agent **getMapClass** and **ReplicateMapClass::mapProperty** definitions.

Queue Processing – Client

When a **ReplicateClientAgent::replicateClientToServer** method is called, all of the captured entries in the client queue after the last send transaction id are serialized and sent to the server where they are applied.

When acknowledgement of the server application is received, the last sent transaction ids are updated. This allows automatic restart of the replication process. If it fails for some reason (for example, a program exception or communications failure), re-executing the **replicateClientToServer** method automatically restarts from the last send queued transaction.

Queue Inquiry

The **ServerQueueForm** form of both server schemas shows details of the captured transactions on the server. This uses the framework **ReplicateQueueTransactions** and **ReplicateQueueEntry** classes to obtain details of the content of the captured and queued transactions.

You can use these classes on both the client and server to inspect the details of the queues.

Data Load

The framework captures, transfers, and applies user transactions automatically (described elsewhere in this document) and maintains an oid map on the server for the corresponding client objects.

However, replication will often also need to apply object changes that have references to existing objects for which the framework does not have an oid map. Examples of this in the sample schemas are the references to **JobStatusCode** and **JobTypeCode** on the **RemoteJob** instances being transferred between server and client.

The **JobStatusCode** instances on the client are not automatically transferred by the framework as they already exist on the server when replication begins, yet the server needs to know the oids of the corresponding client instances on capture of the **RemoteJob** creates.

The main way of handling this is to use the **ReplicateServerAgent::replicateLoad** method to transfer existing server objects to a given client and create them as new instances. The oids of the new instances plus the original server oids are then sent back to the server in the response to the transfer, and added to the server client oid map.

Subsequent replication of creates or changes, including references to the objects from the server, are transferred using the client oids for the equivalent referenced instances on the client.

An example in both client schemas is the **ReplicationManager::requestLoad** method call from the **Application::initialize** method. The **requestLoad** method uses the TCP/IP connection to send an action request to the server.

The request is received in the **JobManagementServer** schema by the **TCPServerManager::stub_LoadRequested** method and in the **JobManagementServerUserComms** schema by the **UserTCPServer::processInput** method, which calls the **UserTCPServerManager::stub_LoadRequested** method for this specific message type. Both server **stub_LoadRequested** methods call the **ReplicationManager::requestLoad** method, which registers the load classes and properties, if not already done, and then uses the server agent **replicateLoad** method to copy the **JobStatusCode**, **JobTypeCode**, **ServiceProvider**, and **Site** instances to the client.

The **ReplicationManager::registerReplicateLoadClasses** method, which calls the **ReplicateServerAgent::registerLoadClass** and **ReplicateLoadClass::registerAllPropertiesUpTo** methods, registers the classes and properties to be loaded. You can also use the **ReplicateLoadClass::registerProperty** method to register individual properties.

The server agent **replicateLoad** method replicates only objects for which a **registerLoadClass** method has been called with the class of this object. The **replicateLoad** method replicates values for all properties of this class and all superclasses for which a **ReplicateLoadClass::registerProperty** or **registerAllPropertiesUpTo** method call has been done.

The **ReplicationManager::requestLoad** method performs a server agent **replicateLoadTransaction** method call after each collection. This isn't necessary in this case, but it illustrates a typical usage of the **replicateLoadTransaction** method that breaks up a block of consecutive **replicateLoad** actions into a single logical transaction, which is transferred to the client as a single unit and applied as a single transaction.

General Features

The following subsections described general Replication Framework features.

Transaction Capture

Capturing user object changes (object add, property update, and object delete) for use by the Replication Framework on both client and server is done using the Transaction Tracing feature of JADE. This feature captures object activity as it happens and then passes the activity to a registered method at the start of a **commitTransaction**.

The Replication Framework registers an internal method as a call-back receiver. If replication tracing has been enabled by the **Agent::enableTracing** method, the framework call-back method captures the activity for all classes that implement the **JIREplicible** interface and saves it in a persistent queue.

On the server, as each object activity is processed, the **JIREplicable::replicateToClient** or **replicateToNominatedClients** method is called on each object, depending on which has been enabled (subject to the use of the **setTransactionTargetClients** method, discussed earlier in this document).

On the client, all applicable object activity is captured to a persistent queue; the **JIREplicable** interface methods are not called.

The transaction-tracing feature captures the set property value, regardless of whether the value has changed.

Each object update is captured and replayed in the same sequence as the original action. In particular, multiple settings of the same property are replayed individually; there is no consolidation of such changes. Automatic consequences of explicit property settings (for example, inverse maintenance) are not replayed as such, as they are assumed to be also automatically done (if required) when the activity is applied to the target system by the framework.

Object Equality

Objects created in captured transactions, transferred, and then applied become new object instances in the target system on the server or client. The new instances have oids (unique object ids made up of class number and instance number) that will be completely different from the original oid.

If the original object is subsequently changed in a captured transaction, the framework code applying the transferred change needs to know this new oid so that it can find the object to apply the captured change to.

The framework uses oids to identify objects but does it via internal oid maps on the server.

When an object is created on the server in a captured transaction that is transferred to a client and applied there, the original server oid is transferred as part of the action. The framework saves the server oid plus the new client oid, and returns both in the response to the transfer. The details are saved persistently on the client for recoverability and they are deleted when confirmation of receipt on the server has been received. In the case of deferred application on the client, the oid details are transferred on the following replication with the server. The server updates an oid map for each registered client with matching client and server oids.

Subsequent captured transactions containing changes to this object are sent with the client oid from this map rather than the server oid, so that the object can be found directly. If there is no such map entry, the server oid is sent. The oids are marked as being client or server. The client keeps an interim server/client oid map for objects received from the server but for which the client oids have not yet been sent back to the server.

When an object is created on the client in a captured transaction that is transferred to the server and applied there, it always sends the client oid. The server uses the oid map for this client to find the equivalent server object.

The data load discussion elsewhere in this document covers the case of determining oids for objects that are referenced by added and changed instances but that have not been automatically replicated.

Although the framework has provision for an alternative user-defined object unique identifier mechanism, this has not yet been implemented.

Object Equality Example

1. When the Meter Reader client application first starts, it requests the existing server **JobStatusCode**, **JobTypeCode**, **ServiceProvider**, and **Site** instances using the **ReplicateServerAgent::replicateLoad** method. This captures and queues object details, which include class name and server oids.

The client then requests these to be sent, and applies the activity as object creates. The resulting object instances from an actual run include the following.

Class Name	Server Oid	Client Oid
JobStatusCode	2449.177	2483.229
JobTypeCode	2450.230	2484.290
ServiceProvider	2453.201	2487.230
Site	2454.417	2488.534

After the details are applied on the client, the framework responds with a list of the server oids and the new client oids. These are saved on the server for each client.

2. When the server **Create Sample** button is clicked, new **RemoteJob** objects are created on the server and captured by the transaction tracing feature.

When the client next requests synchronization, these are sent. The server maps both class and property names and server oids to the client equivalents before sending them, so the object creates are received on the client as:

```
Create, class name=ReadJob server oid=2452.297,
property=jobType (server oid 2450.230) client oid=2484.290,
property=jobStatus (server oid 2449.177) client oid=2483.229
and so on
```

On the client, a new **ReadJob** object is created with the **jobType** and **jobStatus** properties set to the client **JobStatusType** and **JobTypeCode** instances given by the client oid values, which match the equivalent server instances. The new **ReadJob** oid is **2091.148**.

After the details are applied, the framework again responds with a list of the server oids and the new client oids. These are saved on the server, so its oid map for this client now includes the following.

Server Oid	Client Oid
2452.297	2091.148

3. If there is a subsequent change to this **RemoteJob** instance on the server, to set the **notes** property, for example, it is sent to the client as:

```
Update, class name=ReadJob client oid=2091.148, property=notes
value="changed details"
```

Conclusion

The basic functionality and usage of the Replication Framework is covered in this while paper. For details, see the *JADE Replication Framework User's Guide*.