



# Jini™ Transaction Specification

This document specifies the Jini™ two-phase commit protocol, allowing objects using that protocol to enter into distributed transactions. This specification defines the interfaces used by clients, participants, and managers of the protocol. Participants using the protocol may provide any service; they are not limited to databases or other persistent storage services. The default transaction semantics for services is also defined, along with associated semantics classes and interfaces.



Version 1.1Alpha  
November 1999

Copyright © 1999 Sun Microsystems, Inc.  
901 San Antonio Road, Palo Alto, CA 94303 USA.  
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights (“Sun IPR”) relating to implementations of the technology described in this publication (“the Technology”). In particular, and without limitation, Sun IPR may include one or more patents or patent applications in the U.S. or other countries. Your limited right to use this publication does not grant you any right or license to Sun IPR nor any right or license to implement the Technology. Sun may, in its sole discretion, make available a limited license to Sun IPR and/or to the Technology under a separate license agreement. Please visit <http://www.sun.com/software/communitysource/>.

Sun, the Sun logo, Sun Microsystems, Jini, the Jini logo, JavaSpaces, Java, JavaBeans, Solaris, NFS, PC-NFS, EmbeddedJava, PersonalJava, and Solstice are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

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

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE SPECIFICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN ANY TECHNOLOGY, PRODUCT, OR PROGRAM DESCRIBED IN THIS SPECIFICATION AT ANY TIME.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Model and Terms	2
1.2	Distributed Transactions and ACID Properties	4
1.3	Requirements	5
1.4	Dependencies	6
1.5	Comments	6
<b>2</b>	<b>The Two-Phase Commit Protocol</b>	<b>7</b>
2.1	Starting a Transaction	8
2.2	Starting a Nested Transaction	9
2.3	Joining a Transaction	11
2.4	Transaction States	12
2.5	Completing a Transaction: The Client's View	13
2.6	Completing a Transaction: A Participant's View	15
2.7	Completing a Transaction: The Manager's View	18
2.8	Crash Recovery	20
2.8.1	The Roll Decision	21
2.9	Durability	21
<b>3</b>	<b>Default Transaction Semantics</b>	<b>23</b>
3.1	Transaction and NestableTransaction Interfaces	23
3.2	TransactionFactory Class	25
3.3	ServerTransaction and NestableServerTransaction Classes	26
3.4	CannotNestException Class	28
3.5	Semantics	28
3.6	Serialized Forms	30



# TX

---

## The Jini™ Transaction Specification

### TX.1 Introduction

**T**RANSACTIONS are a fundamental tool for many kinds of computing. A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. Transactional behaviors are especially important in distributed computing, where they provide a means for enforcing consistency over a set of operations on one or more remote participants. If all the participants are members of a transaction, one response to a remote failure is to abort the transaction, thereby ensuring that no partial results are written.

Traditional transaction systems often center around transaction processing monitors that ensure that the correct implementation of transactional semantics is provided by all of the participants in a transaction. Our approach to transactional semantics is somewhat different. Within our system we leave it to the individual objects that take part in a transaction to implement the transactional semantics in the way that is best for that kind of object. What the system primarily provides is the coordination mechanism that those objects can use to communicate the information necessary for the set of objects to agree on the transaction. The goal of this system is to provide the *minimal* set of protocols and interfaces that *allow* objects to implement transaction semantics rather than the *maximal* set of interfaces, protocols, and policies that *ensure* the correctness of any possible transaction semantics. So the completion protocol is separate from the semantics of particular transactions.

This document presents this completion protocol, which consists of a two-phase commit protocol for distributed transactions. The two-phase commit proto-

col defines the communication patterns that allow distributed objects and resources to wrap a set of operations in such a way that they appear to be a single operation. The protocol requires a manager that will enable consistent resolution of the operations by a guarantee that all participants will eventually know whether they should commit the operations (roll forward) or abort them (roll backward). A participant can be any object that supports the participant contract by implementing the appropriate interface. Participants are not limited to databases or other persistent storage services.

Clients and servers will also need to depend on specific transaction semantics. The default transaction semantics for participants is also defined in this document.

The two-phase commit protocol presented here, while common in many traditional transaction systems, has the potential to be used in more than just traditional transaction processing applications. Since the semantics of the individual operations and the mechanisms that are used to ensure various properties of the meta-operation joined by the protocol are left up to the individual objects, variations of the usual properties required by transaction processing systems are possible using this protocol, as long as those variances can be resolved by this protocol. A group of objects could use the protocol, for example, as part of a process allowing synchronization of data that have been allowed to drift for efficiency reasons. While this use is not generally considered to be a classical use of transactions, the protocol defined here could be used for this purpose. Some variations will not be possible under these protocols, requiring subinterfaces and subclasses of the ones provided or entirely new interfaces and classes.

Because of the possibility of application to situations that are beyond the usual use of transactions, calling the two-phase commit protocol a transaction mechanism is somewhat misleading. However, since the most common use of such a protocol is in a transactional setting, and because we do define a particular set of default transaction semantics, we will follow the usual naming conventions used in such systems rather than attempting to invent a new, parallel vocabulary.

The classes and interfaces defined by this specification are in the packages `net.jini.core.transaction` and `net.jini.core.transaction.server`. In this document you will usually see these types used without a package prefix; as each type is defined, the package it is in is specified.

### TX.1.1 Model and Terms

A transaction is created and overseen by a *manager*. Each manager implements the interface `TransactionManager`. Each *transaction* is represented by a long identifier that is unique with respect to the transaction's manager.

Semantics are represented by *semantic* transaction objects, such as the ones that represent the default semantics for services. Even though the manager needs to know only how to complete transactions, clients and participants need to share a common view of the semantics of the transaction. Therefore clients typically create, pass, and operate on semantic objects that contain the transaction identifier instead of using the transaction's identifier directly, and transactable services typically accept parameters of a particular semantic type, such as the Transaction interface used for the default semantics.

As shown in Figure TX.1.1, a *client* creates a transaction by a request to the manager, typically by using a semantic factory class such as TransactionFactory to create a semantic object. The semantic object created is then passed as a parameter when performing operations on a service. If the service is to accept this transaction and govern its operations thereby, it must *join* the transaction as a *participant*. Participants in a transaction must implement the interface TransactionParticipant. Particular operations associated with a given transaction are said to be *performed under* that transaction. The client that created the transaction might or might not be a participant in the transaction.

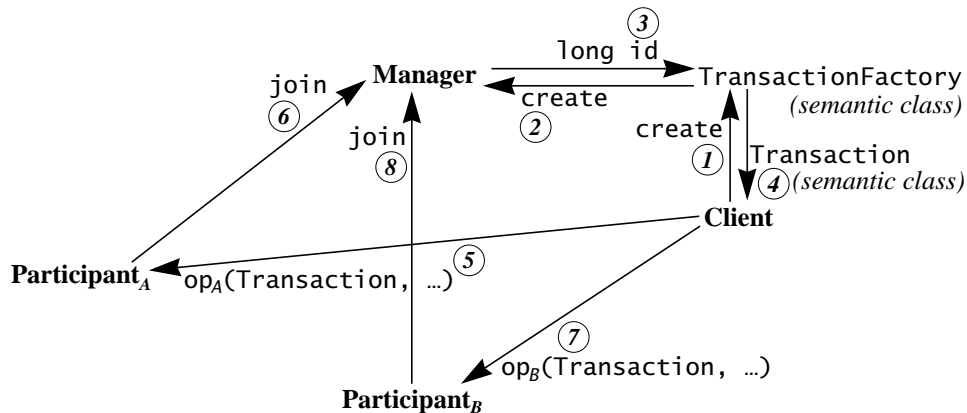


FIGURE TX.1.1: *Transaction Creation and Use*

A transaction *completes* when any entity either *commits* or *aborts* the transaction. If a transaction commits successfully, then all operations performed under that transaction will complete. Aborting a transaction means that all operations performed under that transaction will appear never to have happened.

Committing a transaction requires each participant to *vote*, where a vote is either *prepared* (ready to commit), *not changed* (read-only), or *aborted* (the transaction should be aborted). If all participants vote “prepared” or “not changed,” the

transaction manager will tell each “prepared” participant to *roll forward*, thus committing the changes. Participants that voted “not changed” need do nothing more. If the transaction is ever aborted, the participants are told to *roll back* any changes made under the transaction.

### TX.1.2 Distributed Transactions and ACID Properties

The two-phase commit protocol is designed to enable objects to provide ACID properties. The default transaction semantics define one way to preserve these properties. The ACID properties are:

- ◆ *Atomicity*: All the operations grouped under a transaction occur or none of them do. The protocol allows participants to discover which of these alternatives is expected by the other participants in the protocol. However, it is up to the individual object to determine whether it wishes to operate in concert with the other participants.
- ◆ *Consistency*: The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the realm of the transaction itself—a transaction is a tool to allow consistency guarantees and not itself a guarantor of consistency.
- ◆ *Isolation*: Ongoing transactions should not affect each other. Participants in a transaction should see only intermediate states resulting from the operations of their own transaction, not the intermediate states of other transactions. The protocol allows participating objects to know what operations are being done within the scope of a transaction. However, it is up to the individual object to determine if such operations are to be reflected only within the scope of the transaction or can be seen by others who are not participating in the transaction.
- ◆ *Durability*: The results of a transaction should be as persistent as the entity on which the transaction commits. However, such guarantees are up to the implementation of the object.

The dependency on the participant’s implementation for the ACID properties is the greatest difference between this two-phase commit protocol and more traditional transaction processing systems. Such systems attempt to ensure that the ACID properties are met and go to considerable trouble to ensure that no participant can violate any of the properties.



This approach differs for both philosophical and practical reasons. The philosophical reason is centered on a basic tenet of object-oriented programming, which is that the implementation of an object should be hidden from any part of the system outside the object. Ensuring the ACID properties generally requires that an object's implementation correspond to certain patterns. We believe that if these properties are needed, the object (or, more precisely, the programmer implementing the object) will know best how to guarantee the properties. For this reason, the manager is solely concerned with completing transactions properly. Clients and participants must agree on semantics separately.

The practical reason for leaving the ACID properties up to the object is that there are situations in which only some of the ACID properties make sense, but that can still make use of the two-phase commit protocol. A group of transient objects might wish to group a set of operations in such a way that they appear atomic; in such a situation it makes little sense to require that the operations be durable. An object might want to enable the monitoring of the state of some long-running transactions; such monitoring would violate the isolation requirement of the ACID properties. Binding the two-phase commit protocol to all of these properties limits the use of such a protocol.

We also know that particular semantics are needed for particular services. The default transaction semantics provide useful general-purpose semantics built on the two-phase commit completion protocol.

Distributed transactions differ from single-system transactions in the same way that distributed computing differs from single-system computing. The clearest difference is that a single system can have a single view of the state of several services. It is possible in a single system to make it appear to any observer that all operations performed under a transaction have occurred or none have, thereby achieving isolation. In other words, no observer will ever see only part of the changes made under the transaction. In a distributed system it is possible for a client using two servers to see the committed state of a transaction in one server and the pre-committed state of the same transaction in another server. This can be prevented only by coordination with the transaction manager or the client that committed the transaction. Coordination between clients is outside the scope of this specification.

### **TX.1.3 Requirements**

The transaction system has the following requirements:

- ◆ Define types and contracts that allow the two-phase commit protocol to govern operations on multiple servers of differing types or implementations.

- ◆ Allow participation in the two-phase commit protocol by any object in the Java™ programming language, where “participation” means to perform operations on that object under a given transaction.
- ◆ Each participant may provide ACID properties with respect to that participant to observers operating under a given transaction.
- ◆ Use standard Java programming language techniques and tools to accomplish these goals. Specifically, transactions will rely upon Java Remote Method Invocation (RMI) to communicate between participants.
- ◆ Define specific default transaction semantics for use by services.

### **TX.1.4 Dependencies**

This document relies upon the following other specifications:

- ◆ *Java™ Remote Method Invocation Specification*
- ◆ *Jini™ Distributed Leasing Specification*

### **TX.1.5 Comments**

Please direct comments to `jini-comments@java.sun.com`.

---

## TX.2 The Two-Phase Commit Protocol

**T**HE two-phase commit protocol is defined using three primary types:

- ◆ **TransactionManager**: A transaction manager creates new transactions and coordinates the activities of the participants.
- ◆ **NestableTransactionManager**: Some transaction managers are capable of supporting nested transactions.
- ◆ **TransactionParticipant**: When an operation is performed under a transaction, the participant must join the transaction, providing the manager with a reference to a **TransactionParticipant** object that will be asked to vote, roll forward, or roll back.

The following types are imported from other packages and are referenced in unqualified form in the rest of this specification:

```
java.rmi.Remote  
java.rmi.RemoteException  
java.rmi.NoSuchObjectException  
java.io.Serializable  
net.jini.core.lease.LeaseDeniedException  
net.jini.core.lease.Lease
```

All the methods defined to throw **RemoteException** will do so in the circumstances described by the RMI specification.

Each type is defined where it is first described. Each method is described where it occurs in the lifecycle of the two-phase commit protocol. All methods, fields, and exceptions that can occur during the lifecycle of the protocol will be specified. The section in which each method or field is specified is shown in a comment, using the § abbreviation for the word “section.”

## TX.2.1 Starting a Transaction

The `TransactionManager` interface is implemented by servers that manage the two-phase commit protocol:

```
package net.jini.core.transaction.server;

public interface TransactionManager
    extends Remote, TransactionConstants // §TX.2.4
{
    public static class Created implements Serializable {
        public final long id;
        public final Lease lease;
        public Created(long id, Lease lease) {...}
    }
    Created create(long leaseFor) // §TX.2.1
        throws LeaseDeniedException, RemoteException;
    void join(long id, TransactionParticipant part,
        long crashCount) // §TX.2.3
        throws UnknownTransactionException,
            CannotJoinException, CrashCountException,
            RemoteException;
    int getState(long id) // §TX.2.7
        throws UnknownTransactionException, RemoteException;
    void commit(long id) // §TX.2.5
        throws UnknownTransactionException,
            CannotCommitException,
            RemoteException;
    void commit(long id, long waitFor) // §TX.2.5
        throws UnknownTransactionException,
            CannotCommitException,
            TimeoutExpiredException, RemoteException;
    void abort(long id) // §TX.2.5
        throws UnknownTransactionException,
            CannotAbortException,
            RemoteException;
    void abort(long id, long waitFor) // §TX.2.5
        throws UnknownTransactionException,
            CannotAbortException,
            TimeoutExpiredException, RemoteException;
}
```

A client obtains a reference to a `TransactionManager` object via a lookup service or some other means. The details of obtaining such a reference are outside the scope of this specification.

A client creates a new transaction by invoking the manager's `create` method, providing a desired `leaseFor` time in milliseconds. This invocation is typically indirect via creating a semantic object. The time is the client's expectation of how long the transaction will last before it completes. The manager may grant a shorter lease or may deny the request by throwing `LeaseDeniedException`. If the granted lease expires or is cancelled before the transaction manager receives a `commit` or `abort` of the transaction, the manager will abort the transaction.

The purpose of the `Created` nested class is to allow the `create` method to return two values: the transaction identifier and the granted lease. The constructor simply sets the two fields from its parameters.

## TX.2.2 Starting a Nested Transaction

The `TransactionManager.create` method returns a new *top-level* transaction. Managers that implement just the `TransactionManager` interface support only top-level transactions. *Nested* transactions, also known as *subtransactions*, can be created using managers that implement the `NestableTransactionManager` interface:

```
package net.jini.core.transaction.server;

public interface NestableTransactionManager
    extends TransactionManager
{
    TransactionManager.Created
        create(NestableTransactionManager parentMgr,
              long parentID, long leaseFor) // §TX.2.2
        throws UnknownTransactionException,
               CannotJoinException, LeaseDeniedException,
               RemoteException;
    void promote(long id, TransactionParticipant[] parts,
                 long[] crashCounts,
                 TransactionParticipant drop)
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException; // §TX.2.7
}
```

The `create` method takes a *parent* transaction—represented by the manager for the parent transaction and the identifier for that transaction—and a desired lease time in milliseconds, and returns a new *nested* transaction that is *enclosed* by the specified parent along with the granted lease.

When you use a nested transaction you allow changes to a set of objects to abort without forcing an abort of the parent transaction, and you allow the commit of those changes to still be conditional on the commit of the parent transaction.

When a nested transaction is created, its manager joins the parent transaction. When the two managers are different, this is done explicitly via `join` (§TX.2.3). When the two managers are the same, this may be done in a manager-specific fashion.

The `create` method throws `UnknownTransactionException` if the parent transaction is unknown to the parent transaction manager, either because the transaction ID is incorrect or because the transaction is no longer active and its state has been discarded by the manager.

```
package net.jini.core.transaction;

public class UnknownTransactionException
    extends TransactionException
{
    public UnknownTransactionException() {...}
    public UnknownTransactionException(String desc) {...}
}

public class TransactionException extends Exception {
    public TransactionException() {...}
    public TransactionException(String desc) {...}
}
```

The `create` method throws `CannotJoinException` if the parent transaction is known to the manager but is no longer active.

```
package net.jini.core.transaction;

public class CannotJoinException extends TransactionException
{
    public CannotJoinException() {...}
    public CannotJoinException(String desc) {...}
}
```

### TX.2.3 Joining a Transaction

The first time a client tells a participant to perform an operation under a given transaction, the participant must invoke the transaction manager's `join` method with an object that implements the `TransactionParticipant` interface. This object will be used by the manager to communicate with the participant about the transaction.

```
package net.jini.core.transaction.server;

public interface TransactionParticipant
    extends Remote, TransactionConstants // §TX.2.4
{
    int prepare(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    void commit(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    void abort(TransactionManager mgr, long id) // §TX.2.6
        throws UnknownTransactionException, RemoteException;
    int prepareAndCommit(TransactionManager mgr, long id)
        // §TX.2.7
        throws UnknownTransactionException, RemoteException;
}
```

If the participant's invocation of the `join` method throws `RemoteException`, the participant should not perform the operation requested by the client and should rethrow the exception or otherwise signal failure to the client.

The `join` method's third parameter is a *crash count* that uniquely defines the version of the participant's storage that holds the state of the transaction. Each time the participant loses the state of that storage (because of a system crash if the storage is volatile, for example) it must change this count. For example, the participant could store the crash count in stable storage.

When a manager receives a `join` request, it checks to see if the participant has already joined the transaction. If it has, and the crash count is the same as the one specified in the original `join`, the `join` is accepted but is otherwise ignored. If the crash count is different, the manager throws `CrashCountException` and forces the transaction to abort.

```
package net.jini.core.transaction.server;

public class CrashCountException extends TransactionException
{
    ...
}
```

```
public CrashCountException() {...}  
public CrashCountException(String desc) {...}  
}
```

The participant should reflect this exception back to the client. This check makes join idempotent when it should be, but forces an abort for a second join of a transaction by a participant that has no knowledge of the first join and hence has lost whatever changes were made after the first join.

An invocation of join can throw `UnknownTransactionException`, which means the transaction is unknown to the manager, either because the transaction ID was incorrect, or because the transaction is no longer active and its state has been discarded by the manager. The join method throws `CannotJoinException` if the transaction is known to the manager but is no longer active. In either case the join has failed, and the method that was attempted under the transaction should reflect the exception back to the client. This is also the proper response if join throws a `NoSuchObjectException`.

## TX.2.4 Transaction States

The `TransactionConstants` interface defines constants used in the communication between managers and participants.

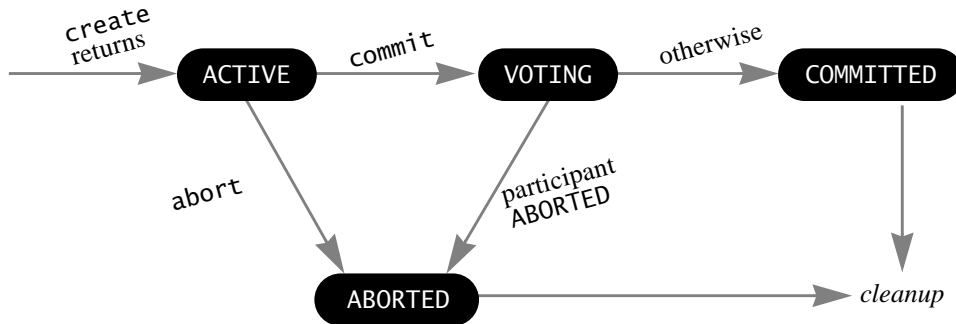
```
package net.jini.core.transaction.server;  
  
public interface TransactionConstants {  
    int ACTIVE = 1;  
    int VOTING = 2;  
    int PREPARED = 3;  
    int NOTCHANGED = 4;  
    int COMMITTED = 5;  
    int ABORTED = 6;  
}
```

These correspond to the states and votes that participants and managers go through during the lifecycle of a given transaction.



## TX.2.5 Completing a Transaction: The Client's View

In the client's view, a transaction goes through the following states:



For the client, the transaction starts out ACTIVE as soon as `create` returns. The client drives the transaction to completion by invoking `commit` or `abort` on the transaction manager, or by cancelling the lease or letting the lease expire (both of which are equivalent to an abort).

The one-parameter `commit` method returns as soon as the transaction successfully reaches the COMMITTED state, or if the transaction is known to have previously reached that state due to an earlier commit. If the transaction reaches the ABORTED state, or is known to have previously reached that state due to an earlier commit or abort, then `commit` throws `CannotCommitException`.

```

package net.jini.core.transaction;

public class CannotCommitException
    extends TransactionException
{
    public CannotCommitException() {...}
    public CannotCommitException(String desc) {...}
}

```

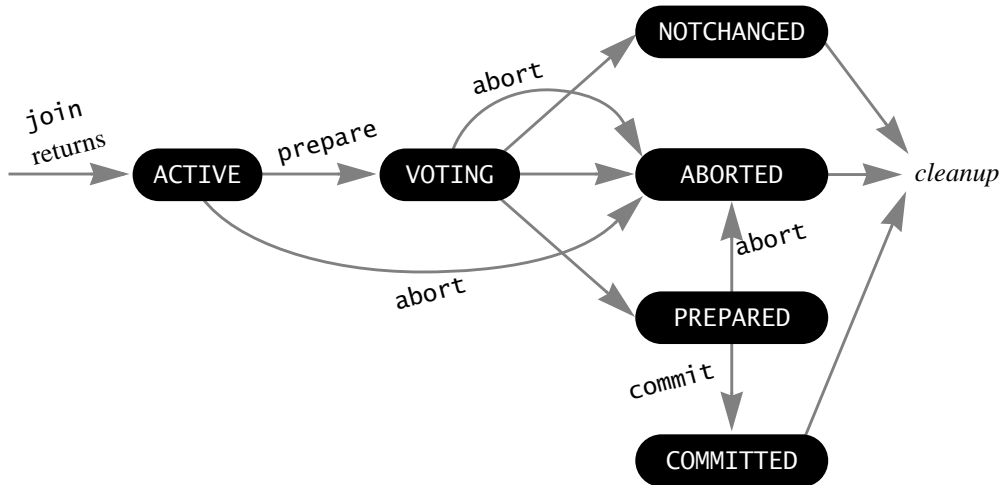
The one-parameter `abort` method returns as soon as the transaction successfully reaches the ABORTED state, or if the transaction is known to have previously reached that state due to an earlier `commit` or `abort`. If the transaction is known to have previously reached the COMMITTED state due to an earlier `commit`, then `abort` throws `CannotAbortException`.

Both `commit` and `abort` can throw `UnknownTransactionException`, which means the transaction is unknown to the manager. This may be because the transaction ID was incorrect, or because the transaction has proceeded to *cleanup* due to an earlier commit or abort, and has been forgotten.

[illegible]

## TX.2.6 Completing a Transaction: A Participant's View

In a participant's view, a transaction goes through the following states:



For the participant, the transaction starts out **ACTIVE** as soon as **join** returns. Any operations attempted under a transaction are valid only if the participant has the transaction in the **ACTIVE** state. In any other state, a request to perform an operation under the transaction should fail, signaling the invoker appropriately.

When the manager asks the participant to **prepare**, the participant is **VOTING** until it decides what to return. There are three possible return values for **prepare**:

- ◆ The participant had no changes to its state made under the transaction—that is, for the participant the transaction was read-only. It should release any internal state associated with the transaction. It must signal this with a return of **NOTCHANGED**, effectively entering the **NOTCHANGED** state. As noted below, a well-behaved participant should stay in the **NOTCHANGED** state for some time to allow idempotency for **prepare**.
- ◆ The participant had its state changed by operations performed under the transaction. It must attempt to **prepare** to roll those changes forward in the event of a future incoming **commit** invocation. When the participant has successfully prepared itself to roll forward (§TX.2.8), it must return **PREPARED**, thereby entering the **PREPARED** state.
- ◆ The participant had its state changed by operations performed under the transaction but is unable to guarantee a future successful roll forward. It

must signal this with a return of `ABORTED`, effectively entering the `ABORTED` state.

For top-level transactions, when a participant returns `PREPARED` it is stating that it is ready to roll the changes forward by saving the necessary record of the operations for a future `commit` call. The record of changes must be at least as durable as the overall state of the participant. The record must also be examined during recovery (§TX.2.8) to ensure that the participant rolls forward or rolls back as the manager dictates. The participant stays in the `PREPARED` state until it is told to `commit` or `abort`. It cannot, having returned `PREPARED`, drop the record except by following the “roll decision” described for crash recovery (§TX.2.8.1).

For nested transactions, when a participant returns `PREPARED` it is stating that it is ready to roll the changes forward into the parent transaction. The record of changes must be as durable as the record of changes for the parent transaction.

If a participant is currently executing an operation under a transaction when `prepare` is invoked for that transaction, the participant must either: wait until that operation is complete before returning from `prepare`; know that the operation is guaranteed to be read-only, and so will not affect its ability to `prepare`; or `abort` the transaction.

If a participant has not received any communication on or about a transaction over an extended period, it may choose to invoke `getState` on the manager. If `getState` throws `UnknownTransactionException` or `NoSuchObjectException`, the participant may safely infer that the transaction has been aborted. If `getState` throws a `RemoteException` the participant may choose to believe that the manager has crashed and abort its state in the transaction—this is not to be done lightly, since the manager may save state across crashes, and transient network failures could cause a participant to drop out of an otherwise valid and committable transaction. A participant should drop out of a transaction only if the manager is unreachable over an extended period. However, in no case should a participant drop out of a transaction it has `PREPARED` but not yet rolled forward.

If a participant has joined a nested transaction and it receives a `prepare` call for an enclosing transaction, the participant must complete the nested transaction, using `getState` on the manager to determine the proper type of completion.

If a participant receives a `prepare` call for a transaction that is already in a post-VOTING state, the participant should simply respond with that state.

If a participant receives a `prepare` call for a transaction that is unknown to it, it should throw `UnknownTransactionException`. This may happen if the participant has crashed and lost the state of a previously active transaction, or if a previous `NOTCHANGED` or `ABORTED` response was not received by the manager and the participant has since forgotten the transaction.

Note that a return value of NOTCHANGED may not be idempotent. Should the participant return NOTCHANGED it may proceed directly to clean up its state. If the manager receives a `RemoteException` because of network failure, the manager will likely retry the prepare. At this point a participant that has dropped the information about the transaction will throw `UnknownTransactionException`, and the manager will be forced to abort. A well-behaved participant should stay in the NOTCHANGED state for a while to allow a retry of prepare to again return NOTCHANGED, thus keeping the transaction alive, although this is not strictly required. No matter what it voted, a well-behaved participant should also avoid exiting for a similar period of time in case the manager needs to re-invoke prepare.

If a participant receives an abort call for a transaction, whether in the ACTIVE, VOTING, or PREPARED state, it should move to the ABORTED state and roll back all changes made under the transaction.

If a participant receives a commit call for a PREPARED transaction, it should move to the COMMITTED state and roll forward all changes made under the transaction.

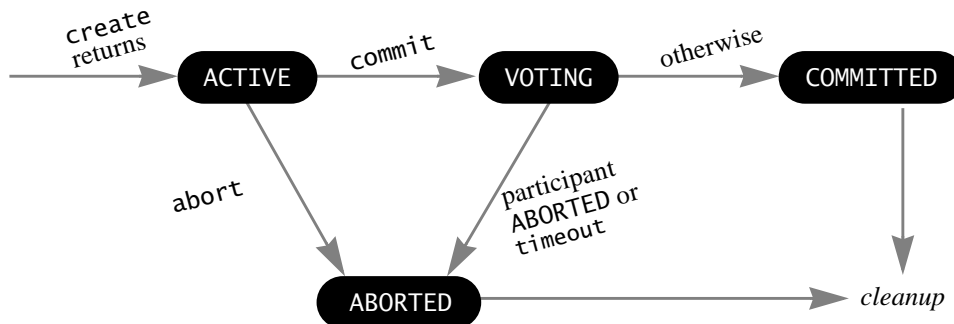
The participant's implementation of `prepareAndCommit` must be equivalent to the following:

```
public int prepareAndCommit(TransactionManager mgr, long id)
    throws UnknownTransactionException, RemoteException
{
    int result = prepare(mgr, id);
    if (result == PREPARED) {
        commit(mgr, id);
        result = COMMITTED;
    }
    return result;
}
```

The participant can often implement `prepareAndCommit` much more efficiently than shown, but it must preserve the above semantics. The manager's use of this method is described in the next section.

## TX.2.7 Completing a Transaction: The Manager's View

In the manager's view, a transaction goes through the following states:



When a transaction is created using `create`, the transaction is **ACTIVE**. This is the only state in which participants may join the transaction. Attempting to join the transaction in any other state throws a `CannotJoinException`.

Invoking the manager's `commit` method causes the manager to move to the **VOTING** state, in which it attempts to complete the transaction by rolling forward. Each participant that has joined the transaction has its `prepare` method invoked to vote on the outcome of the transaction. The participant may return one of three votes: `NOTCHANGED`, `ABORTED`, or `COMMITTED`.

If a participant votes `ABORTED`, the manager must abort the transaction. If `prepare` throws `UnknownTransactionException` or `NoSuchObjectException`, the participant has lost its state of the transaction, and the manager must abort the transaction. If `prepare` throws `RemoteException`, the manager may retry as long as it wishes until it decides to abort the transaction.

To abort the transaction, the manager moves to the **ABORTED** state. In the **ABORTED** state, the manager should invoke `abort` on all participants that have voted `PREPARED`. The manager should also attempt to invoke `abort` on all participants on which it has not yet invoked `prepare`. These notifications are not strictly necessary for the one-parameter forms of `commit` and `abort`, since the participants will eventually abort the transaction either by timing out or by asking the manager for the state of the transaction. However, informing the participants of the abort can speed up the release of resources in these participants, and so attempting the notification is strongly encouraged.

If a participant votes `NOTCHANGED`, it is dropped from the list of participants, and no further communication will ensue. If all participants vote `NOTCHANGED` then the entire transaction was read-only and no participant has any changes to roll forward. The transaction moves to the **COMMITTED** state and then can immediately

move to *cleanup*, in which resources in the manager are cleaned up. There is no behavioral difference to a participant between a NOTCHANGED transaction and one that has completed the notification phase of the COMMITTED state.

If no participant votes ABORTED and at least one participant votes PREPARED, the transaction also moves to the COMMITTED state. In the COMMITTED state the manager must notify each participant that returned PREPARED to roll forward by invoking the participant's `commit` method. When the participant's `commit` method returns normally, the participant has rolled forward successfully and the manager need not invoke `commit` on it again. As long as there exists at least one participant that has not rolled forward successfully, the manager must preserve the state of the transaction and repeat attempts to invoke `commit` at reasonable intervals. If a participant's `commit` method throws `UnknownTransactionException`, this means that the participant has already successfully rolled the transaction forward even though the manager did not receive the notification, either due to a network failure on a previous invocation that was actually successful or because the participant called `getState` directly.

If the transaction is a nested one and the manager is prepared to roll the transaction forward, the members of the nested transaction must become members of the parent transaction. This *promotion* of participants into the parent manager must be atomic—all must be promoted simultaneously, or none must be. The multi-participant `promote` method is designed for this use in the case in which the parent and nested transactions have different managers.

The `promote` method takes arrays of participants and crash counts, where `crashCounts[i]` is the crash count for `parts[i]`. If any crash count is different from a crash count that is already known to the parent transaction manager, the parent manager throws `CrashCountException` and the parent transaction must abort. The `drop` parameter allows the nested transaction manager to drop itself out of the parent transaction as it promotes its participants into the parent transaction if it no longer has any need to be a participant itself.

The manager for the nested transaction should remain available until it has successfully driven each participant to completion and promoted its participants into the parent transaction. If the nested transaction's manager disappears before a participant is positively informed of the transaction's completion, that participant will not know whether to roll forward or back, forcing it to vote ABORTED in the parent transaction. The manager may cease `commit` invocations on its participants if any parent transaction is aborted. Aborting any transaction implicitly aborts any uncommitted nested transactions. Additionally, since any committed nested transaction will also have its results dropped, any actions taken on behalf of that transaction can be abandoned.

Invoking the manager's `abort` method, cancelling the transaction's lease, or allowing the lease to expire also moves the transaction to the ABORTED state as

described above. Any transactions nested inside that transaction are also moved directly to the ABORTED state.

The manager may optimize the VOTING state by invoking a participant's `prepareAndCommit` method if the transaction has only one participant that has not yet been asked to vote and all previous participants have returned NOTCHANGED. (Note that this includes the special case in which the transaction has exactly one participant.) If the manager receives an ABORTED result from `prepareAndCommit`, it proceeds to the ABORTED state. In effect, a `prepareAndCommit` moves through the VOTING state straight to operating on the results.

A `getState` call on the manager can return any of ACTIVE, VOTING, ABORTED, NOTCHANGED, or COMMITTED. A manager is permitted, but not required, to return NOTCHANGED if it is in the COMMITTED state and all participants voted NOTCHANGED.

## TX.2.8 Crash Recovery

Crash recovery ensures that a top-level transaction will consistently abort or roll forward in the face of a system crash. Nested transactions are not involved.

The manager has one *commit point*, where it must save state in a durable fashion. This is when it enters the COMMITTED state with at least one PREPARED participant. The manager must, at this point, commit the list of PREPARED participants into durable storage. This storage must persist until all PREPARED participants successfully roll forward. A manager may choose to also store the list of PREPARED participants that have already successfully rolled forward or to rewrite the list of PREPARED participants as it shrinks, but this optimization is not required (although it is recommended as good citizenship). In the event of a manager crash, the list of participants must be recovered, and the manager must continue acting in the COMMITTED state until it can successfully notify all PREPARED participants.

The participant also has one commit point, which is prior to voting PREPARED. When it votes PREPARED, the participant must have durably recorded the record of changes necessary to successfully roll forward in the event of a future invocation of `commit` by the manager. It can remove this record when it is prepared to successfully return from `commit`.

Because of these commitments, manager and participant implementations should use durable forms of RMI references, such as the `Activatable` references introduced in the Java™ 2 platform. An unreachable manager causes much havoc and should be avoided as much as possible. A vanished PREPARED participant puts a transaction in an untenable permanent state in which some, but not all, of the participants have rolled forward.



### TX.2.8.1 The Roll Decision

If a participant votes PREPARED for a top-level transaction, it must guarantee that it will execute a recovery process if it crashes between completing its durable record and receiving a `commit` notification from the manager. This recovery process must read the record of the crashed participant and make a *roll decision*—whether to roll the recorded changes forward or roll them back.

To make this decision, it invokes the `getState` method on the transaction manager. This can have the following results:

- ◆ `getState` returns `COMMITTED`: The recovery should move the participant to the `COMMITTED` state.
- ◆ `getState` throws either an `UnknownTransactionException` or a `NoSuchObjectException`: The recovery should move the participant to the `ABORTED` state.
- ◆ `getState` throws `RemoteException`: The recovery should repeat the attempt after a pause.

## TX.2.9 Durability

Durability is a commitment, but it is not a guarantee. It is impossible to guarantee that any given piece of stable storage can *never* be lost; one can only achieve decreasing probabilities of loss. Data that is force-written to a disk may be considered durable, but it is less durable than data committed to two or more separate, redundant disks. When we speak of “durability” in this system it is always used relative to the expectations of the human who decided which entities to use for communication.

With multi-participant transactions it is entirely possible that different participants have different durability levels. The manager may be on a tightly replicated system with its durable storage duplicated on several host systems, giving a high degree of durability, while a participant may be using only one disk. Or a participant may always store its data in memory, expecting to lose it in a system crash (a database of people currently logged into the host, for example, need not survive a system crash). When humans make a decision to use a particular manager and set of participants for a transaction they must take into account these differences and be aware of the ramifications of committing changes that may be more durable on one participant than another. Determining, or even defining and exposing, varying levels of durability is outside the scope of this specification.



---

## TX.3 Default Transaction Semantics

**T**HE two-phase commit protocol defines how a transaction is created and later driven to completion by either committing or aborting. It is neutral with respect to the semantics of locking under the transaction or other behaviors that impart semantics to the use of the transaction. Specific clients and servers, however, must be written to expect specific transaction semantics. This model is to separate the completion protocol from transaction semantics, where transaction semantics are represented in the parameters and return values of methods by which clients and participants interact.

This chapter defines the default transaction semantics of services. These semantics preserve the traditional ACID properties (you will find a brief description of the ACID properties in §TX.1.2). The semantics are represented by the `Transaction` and `NestableTransaction` interfaces and their implementation classes `ServerTransaction` and `NestableServerTransaction`. Any participant that accepts as a parameter or returns any of these types is promising to abide by the following definition of semantics for any activities performed under that transaction.

### TX.3.1 Transaction and NestableTransaction Interfaces

The client's view of transactions is through two interfaces: `Transaction` for top-level transactions and `NestableTransaction` for transactions under which nested transactions can be created. First, the `Transaction` interface:

```
package net.jini.core.transaction;

public interface Transaction {
    public static class Created implements Serializable {
        public final Transaction transaction;
        public final Lease lease;
        Created(Transaction transaction, Lease lease) {...}
    }
}
```

```

void commit() // §TX.2.5
    throws UnknownTransactionException,
           CannotCommitException,
           RemoteException;
void commit(long waitFor) // §TX.2.5
    throws UnknownTransactionException,
           CannotCommitException,
           TimeoutExpiredException, RemoteException;
void abort() // §TX.2.5
    throws UnknownTransactionException,
           CannotAbortException,
           RemoteException;
void abort(long waitFor) // §TX.2.5
    throws UnknownTransactionException,
           CannotAbortException,
           TimeoutExpiredException, RemoteException;
}

```

The Created nested class is used in a factory create method for top-level transactions (defined in the next section) to hold two return values: the newly created Transaction object and the transaction's lease, which is the lease granted by the transaction manager. The commit and abort methods have the same semantics as discussed in §TX.2.5.

Nested transactions are created using NestableTransaction methods:

```

package net.jini.core.transaction;

public interface NestableTransaction extends Transaction {
    public static class Created implements Serializable {
        public final NestableTransaction transaction;
        public final Lease lease;
        Created(NestableTransaction transaction, Lease lease)
            {...}
    }
    Created create(long leaseFor) // §TX.2.2
        throws UnknownTransactionException,
               CannotJoinException, LeaseDeniedException,
               RemoteException;
    Created create(NestableTransactionManager mgr,
                  long leaseFor) // §TX.2.2
        throws UnknownTransactionException,

```

```

        CannotJoinException, LeaseDeniedException,
        RemoteException;
    }

```

The Created nested class is used to hold two return values: the newly created Transaction object and the transaction's lease, which is the lease granted by the transaction manager. In both create methods, leaseFor is the requested lease time in milliseconds. In the one-parameter create method the nested transaction is created with the same transaction manager as the transaction on which the method is invoked. The other create method can be used to specify a different transaction manager to use for the nested transaction.

### TX.3.2 TransactionFactory Class

The TransactionFactory class is used to create top-level transactions.

```

package net.jini.core.transaction;

public class TransactionFactory {
    public static Transaction.Created
        create(TransactionManager mgr, long leaseFor)
                                                    // §TX.2.1
        throws LeaseDeniedException, RemoteException {...}
    public static NestableTransaction.Created
        create(NestableTransactionManager mgr, long leaseFor)
                                                    // §TX.2.2
        throws LeaseDeniedException, RemoteException {...}
}

```

The first create method is usually used when nested transactions are not required. However, if the manager that is passed to this method is in fact a NestableTransactionManager, then the returned Transaction can in fact be cast to a NestableTransaction. The second create method is used when it is known that nested transactions need to be created. In both cases, a Created instance is used to hold two return values: the newly created transaction object and the granted lease.

### TX.3.3 ServerTransaction and NestableServerTransaction Classes

The ServerTransaction class exposes functionality necessary for writing participants that support top-level transactions. Participants can cast a Transaction to a ServerTransaction to obtain access to this functionality.

```
public class ServerTransaction
    implements Transaction, Serializable
{
    public final TransactionManager mgr;
    public final long id;
    public ServerTransaction(TransactionManager mgr, long id)
        {...}
    public void join(TransactionParticipant part,
                     long crashCount) // §TX.2.3
        throws UnknownTransactionException,
               CannotJoinException, CrashCountException,
               RemoteException {...}
    public int getState() // §TX.2.7
        throws UnknownTransactionException, RemoteException
        {...}
    public boolean isNested() {...} // §TX.3.3
}
```

The `mgr` field is a reference to the transaction manager that created the transaction. The `id` field is the transaction identifier returned by the transaction manager's `create` method.

The constructor should not be used directly; it is intended for use by the `TransactionFactory` implementation.

The methods `join`, `commit`, `abort`, and `getState` invoke the corresponding methods on the manager, passing the transaction identifier. They are provided as a convenience to the programmer, primarily to eliminate the possibility of passing an identifier to the wrong manager. For example, given a `ServerTransaction` object `tr`, the invocation

```
tr.join(participant, crashCount);
```

is equivalent to

```
tr.mgr.join(tr.id, participant, crashCount);
```

The `isNested` method returns `true` if the transaction is a nested transaction (that is, if it is a `NestableServerTransaction` with a non-null parent) and

false otherwise. It is provided as a method on `ServerTransaction` for the convenience of participants that do not support nested transactions.

The `hashCode` method returns the `id` cast to an `int` XORed with the result of `mgr.hashCode()`. The `equals` method returns true if the specified object is a `ServerTransaction` object with the same manager and transaction identifier as the object on which it is invoked.

The `NestableServerTransaction` class exposes functionality that is necessary for writing participants that support nested transactions. Participants can cast a `NestableTransaction` to a `NestableServerTransaction` to obtain access to this functionality.

```
package net.jini.core.transaction.server;

public class NestableServerTransaction
    extends ServerTransaction implements NestableTransaction
{
    public final NestableServerTransaction parent;
    public NestableServerTransaction(
        NestableTransactionManager mgr, long id,
        NestableServerTransaction parent) {...}
    public void promote(TransactionParticipant[] parts,
        long[] crashCounts,
        TransactionParticipant drop)
        // §TX.2.7
        throws UnknownTransactionException,
            CannotJoinException, CrashCountException,
            RemoteException {...}
    public boolean enclosedBy(NestableTransaction enclosing)
        {...}
}
```

The `parent` field is a reference to the parent transaction if the transaction is nested (§TX.2.2) or null if it is a top-level transaction.

The constructor should not be used directly; it is intended for use by the `TransactionFactory` and `NestableServerTransaction` implementations.

Given a `NestableServerTransaction` object `tr`, the invocation

```
tr.promote(parts, crashCounts, drop)
```

is equivalent to

```
((NestableTransactionManager)tr.mgr).promote(tr.id, parts,
    crashCounts, drop)
```

The `enclosedBy` method returns `true` if the specified transaction is an enclosing transaction (parent, grandparent, etc.) of the transaction on which the method is invoked; otherwise it returns `false`.

### TX.3.4 CannotNestException Class

If a service implements the default transaction semantics but does not support nested transactions, it usually needs to throw an exception if a nested transaction is passed to it. The `CannotNestException` is provided as a convenience for this purpose, although a service is not required to use this specific exception.

```
package net.jini.core.transaction;

public class CannotNestException extends TransactionException
{
    public CannotNestException() {...}
    public CannotNestException(String desc) {...}
}
```

### TX.3.5 Semantics

Activities that are performed as pure transactions (all access to shared mutable state is performed under transactional control) are subject to sequential ordering, meaning the overall effect of executing a set of sibling (all at the same level, whether top-level or nested) pure transactions concurrently is always equivalent to some sequential execution.

Ancestor transactions can execute concurrently with child transactions, subject to the locking rules below.

Transaction semantics for objects are defined in terms of strict two-phase locking. Every transactional operation is described in terms of acquiring locks on objects; these locks are held until the transaction completes. The most typical locks are read and write locks, but others are possible. Whatever the lock types are, conflict rules are defined such that if two operations do not commute, then they acquire conflicting locks. For objects using standard read and write locks, read locks do not conflict with other read locks, but write locks conflict with both read locks and other write locks. A transaction can acquire a lock if the only conflicting locks are those held by ancestor transactions (or itself). If a necessary lock cannot be acquired and the operation is defined to proceed without waiting for that



lock, then serializability might be violated. When a subtransaction commits, its locks are inherited by the parent transaction.

In addition to locks, transactional operations can be defined in terms of object creation and deletion visibility. If an object is defined to be created under a transaction, then the existence of the object is visible only within that transaction and its inferiors, but will disappear if the transaction aborts. If an object is defined to be deleted under a transaction, then the object is not visible to any transaction (including the deleting transaction) but will reappear if the transaction aborts. When a nested transaction commits, visibility state is inherited by the parent transaction.

Once a transaction reaches the VOTING stage, if all execution under the transaction (and its subtransactions) has finished, then the only reasons the transaction can abort are:

- ◆ The manager crashes (or has crashed)
- ◆ One or more participants crash (or have crashed)
- ◆ There is an explicit abort

Transaction deadlocks are not guaranteed to be prevented or even detected, but managers and participants are permitted to break known deadlocks by aborting transactions.

An active transaction is an *orphan* if it or one of its ancestors is guaranteed to abort. This can occur because an ancestor has explicitly aborted or because some participant or manager of the transaction or an ancestor has crashed. Orphans are not guaranteed to be detected by the system, so programmers using transactions must be aware that orphans can see internally inconsistent state and take appropriate action.

Causal ordering information about transactions is not guaranteed to be propagated. First, given two sibling transactions (at any level), it is not possible to tell whether they were created concurrently or sequentially (or in what order). Second, if two transactions are causally ordered and the earlier transaction has completed, the outcome of the earlier transaction is not guaranteed to be known at every participant used by the later transaction, unless the client is successful in using the variant of `commit` or `abort` that takes a timeout parameter. Programmers using non-blocking forms of operations must take this into account.

As long as a transaction persists in attempting to acquire a lock that conflicts with another transaction, the participant will persist in attempting to resolve the outcome of the transaction that holds the conflicting lock. Attempts to acquire a lock include making a blocking call, continuing to make non-blocking calls, and registering for event notification under a transaction.

**TX.3.6 Serialized Forms**

Class	serialVersionUID	Serialized Fields
Transaction.Created	-5199291723008952986L	<i>all public fields</i>
NestableTransaction.Created	-2979247545926318953L	<i>all public fields</i>
TransactionManager.Created	-4233846033773471113L	<i>all public fields</i>
ServerTransaction	4552277137549765374L	<i>all public fields</i>
NestableServerTransaction	-3438419132543972925L	<i>all public fields</i>
TransactionException	-5009935764793203986L	<i>none</i>
CannotAbortException	3597101646737510009L	<i>none</i>
CannotCommitException	-4497341152359563957L	<i>none</i>
CannotJoinException	5568393043937204939L	<i>none</i>
CannotNestException	3409604500491735434L	<i>none</i>
TimeoutExpiredException	3918773760682958000L	<i>all public fields</i>
UnknownTransactionException	443798629936327009L	<i>none</i>
CrashCountException	4299226125245015671L	<i>none</i>