

Atomikos TransactionsEssentials™

API Specification Guide



Atomikos TransactionsEssentials API Specification Guide

Copyright © 2006 Atomikos

Table of Contents

1. Preface	1
1.1. Who Should Read This Guide	1
1.2. Contents	1
1.3. Standards Compliance	1
1.4. Acknowledgements	1
2. Atomikos TransactionsEssentials API Overview	3
2.1. Configuring and Starting/Stopping the Transaction Service	3
2.2. Creating Transactions	3
2.2.1. CompositeTransactionManager	4
2.2.2. ExportingTransactionManager	5
2.2.3. ImportingTransactionManager	5
2.2.4. Propagation	6
2.2.5. Extent	6
2.3. Atomikos' CompositeTransaction Model	7
2.3.1. CompositeTransaction	7
2.3.2. CompositeCoordinator	9
2.3.3. RecoveryCoordinator	10
2.3.4. HeuristicMessage	10
2.3.5. SubTxThread (in package com.atomikos.icatch.system)	10
2.3.6. Waiter (in package com.atomikos.icatch.system)	11
2.4. Plug-In Interfaces	11
2.4.1. TSLListener	11
2.4.2. SubTxCode	11
2.4.3. Synchronization	11
2.4.4. SubTxAwareParticipant	12
2.4.5. Participant	12
2.4.6. RecoverableResource	13
2.5. Exceptions	13
2.5.1. Checked Exceptions	14
2.5.2. Runtime Exceptions	14
3. Programming with the Atomikos TransactionsEssentials API	16
3.1. Basic Level	16
3.1.1. Importing an existing transaction	18
3.2. Intermediate Level	20
3.2.1. Subtransaction Commit	20
3.2.2. Subtransaction Rollback	21
3.2.3. Implementing Parallel Subtransactions	22
3.2.4. Adding Synchronization Implementations	23
3.2.5. Adding SubTxAware Implementations	24
3.2.6. Adding Participant Implementations	24
3.3. Advanced Level	24
3.3.1. The Big Picture	24
3.3.2. Implementing Lock Inheritance	25
3.3.3. Implementing a Participant	27
3.3.4. Implementing a RecoverableResource	28
4. The Administration API	30
4.1. Administration API Overview	30
4.1.1. LogControl	30
4.1.2. LogAdministrator	30
4.1.3. AdminTransaction	30
4.2. Interactions with the Transaction Service	31

A. References	33
---------------------	----

List of Tables

3.1. Subtransaction commit 20

List of Examples

3.1. Typical export code fragment	16
3.2. Typical import code sample	18

Chapter 1. Preface

1.1. Who Should Read This Guide

You should read this guide if:

- You want to use Atomikos TransactionsEssentials for explicitly importing or exporting transactions to other server VMs.
- You want to use the Atomikos TransactionsEssentials API to make your own applications transactional and two-phase commit aware.
- You want to use nested transactions.
- You want to use different threads for one transaction, or want to make parallel remote calls that belong to the same transaction.
- You want to implement a data source that can take part in nested transactions and that offers lock inheritance.
- You want to understand the design of the Atomikos TransactionsEssentials core.
- You want to offer customized administration tools for the transaction engine.

You do NOT have to read this guide if you just want to use JTA transactions *within one VM*.

1.2. Contents

This user guide explains how to use *Atomikos TransactionsEssentials API*. This guide is a *supplement* to the basic user guide in the particular release you are using. As such, whenever this guide refers to the basic guide it is meant the basic user guide for your particular release. Although the different product releases each have their own particularities, the Atomikos TransactionsEssentials API is a shared feature for all of them.

1.3. Standards Compliance

The core Atomikos TransactionsEssentials API is *not a vendor-independent standard*. It is a core API for the Atomikos TransactionsEssentials system and as such it has been designed to fit exactly with Atomikos' vision on transactions and transactional functionality. This does not mean that we do not like standards - on the contrary: we offer standards-compliant products that essentially wrap the internal core API to match standards such as JTA, WS-Transactions and others.

The core API has been designed to make it as easy as possible to adapt to new standards for transaction processing. This is because we focused on the bare essentials of transaction management: the functionality and information that no transaction system can do without.

1.4. Acknowledgements

The Atomikos TransactionsEssentials API has been designed around a lot of past work, by Atomikos and by others. A lot of the design is similar to *CORBA's OTS*, with major differences in the explicit support for distributed

transactions, heuristic terminations and the non-classical interpretation of rollback/commit. Also, the Atomikos API was designed to be pure Java without the need for IDL data types.

The basis of this API goes back to ETH Zurich, Switzerland: the basic research and prototypes that formed the starting point of our design were realized in the group of *Prof. Gustavo Alonso*. Back in the year 2000, we implemented a webservice transaction system *avant la lettre*. Not surprisingly, this is one of the major strengths of Atomikos API right now.

Chapter 2. Atomikos TransactionsEssentials API Overview

The overview of the API is structured into the following main parts:

- Configuring and Starting/Stopping the Transaction Service
- Creating Transactions
- Atomikos CompositeTransaction Model
- Plug-In Interfaces
- Exceptions

The first part deals with startup and configuration of the transaction service. The second part is about creating, importing or exporting transactions. The third part explains the structure of the core API's transaction model. The fourth part explains what external 'hooks' there are to the APIs. The interfaces in this part are those that users can implement to extend the transactional functionality of Atomikos TransactionsEssentials. The last part explains the exceptions defined and used in the API.

Unless explicitly state otherwise, all API components are in the package *com.atomikos.icatch*.

2.1. Configuring and Starting/Stopping the Transaction Service

The configuration and startup/shutdown of the transaction service core can be done via the classes and interfaces in package *com.atomikos.icatch.config*. Even if you start the transaction service via other ways such as JTA, the classes in this package are still involved underneath.

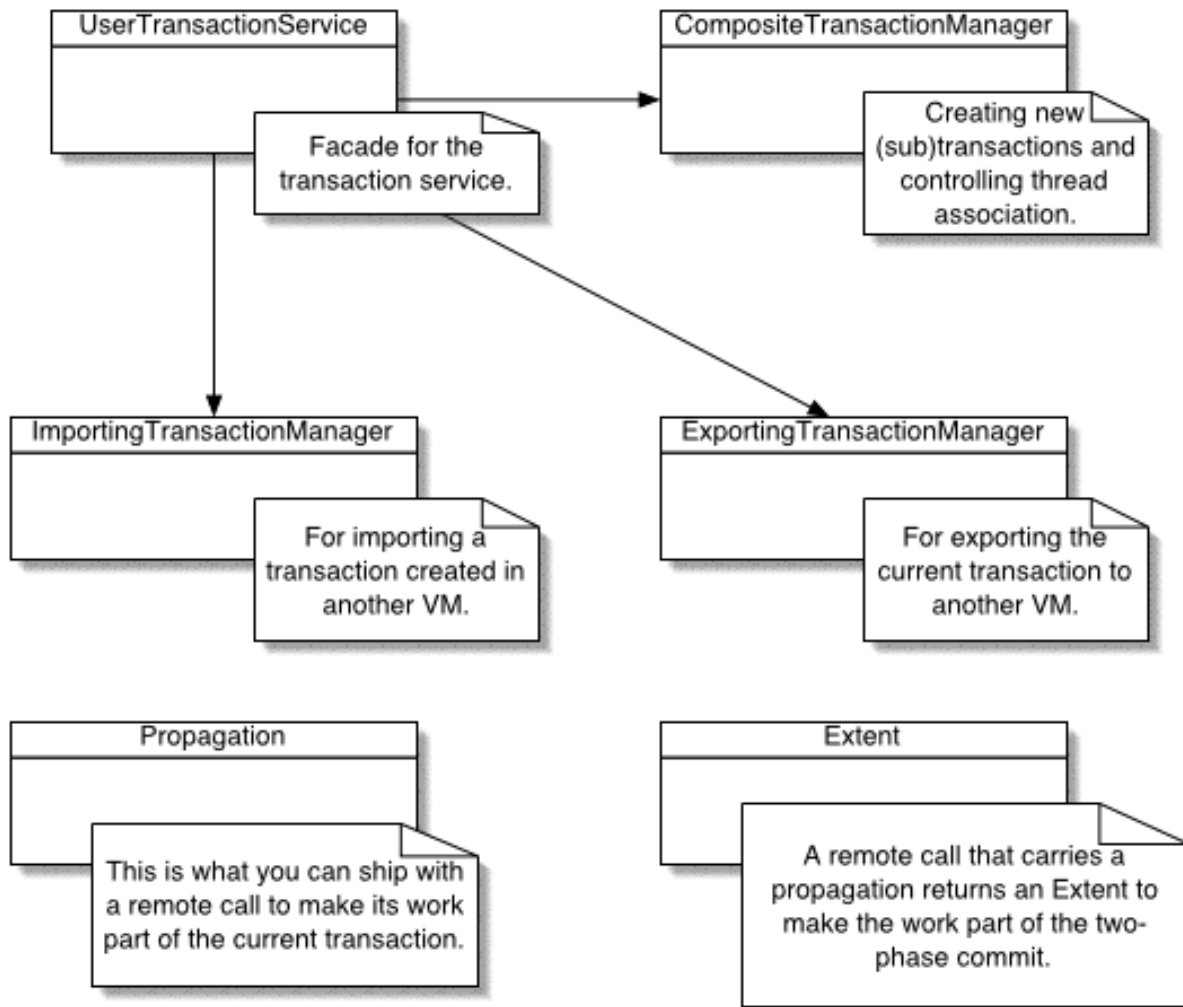
The configuration API allows you to programmatically set transaction service properties (overriding any equivalent properties that may be found in a properties file). This can be done on an instance of *com.atomikos.icatch.config.TSInitInfo* (obtained via an instance of the *com.atomikos.icatch.config.UserTransactionService*).

Further configuration is possible via the *com.atomikos.icatch.config.UserTransactionService* interface (with default implementation in class *com.atomikos.icatch.config.UserTransactionServiceImp*): here you can add transaction-aware resources, transaction service listeners and log administrators (all these are explained in more detail later in the guide).

The *UserTransactionService* also gives you access to the rest of the core API needed to do the things explained in this manual.

2.2. Creating Transactions

Before any transaction exists, it has to be created or imported. An overview of the API is shown in the following picture:



2.2.1. CompositeTransactionManager

The transaction manager is where you can create new transactions and associate a thread with a transaction. It also allows your application to retrieve the current transaction, after you have created one. Note that this is thread-safe: if you have multiple threads running concurrently, then each thread can create its own transaction and will be able to retrieve only that transaction which it created. The transaction manager will behave as a 'private' manager for each thread of your application.

The CompositeTransactionManager is a core interface that is used by the JTA transaction manager implementation of Atomikos. Whenever you begin a JTA transaction through the JTA interfaces, this will also cause a corresponding thread association for the underlying CompositeTransactionManager. In this way, these two transaction manager instances each provide a different API view on the same underlying transaction.

The following methods are provided:

- *createCompositeTransaction*: this method creates a transaction for the application. When it returns, you will be able to retrieve the transaction object through *getCompositeTransaction* within the current thread. Atomikos TransactionsEssentials supports *nested transactions*, meaning that a transaction can be created within another one. This means that calling this method twice in the same thread (without commit/rollback in between) will create a nested transaction, whose final commit will coincide with the commit of the first transaction you created. The thread association is terminated by calling either suspend, or commit/rollback for the composite transaction (as explained below).
- *getCompositeTransaction*: if called without arguments, then this method returns the transaction object for the calling thread, or null if there is no active transaction. The transaction object is needed in order to add work to it: all the work that needs to be part of this transaction must be explicitly added to it.

If called with a string argument, then this method returns the transaction with the given ID (if any). This variant is useful for retrieving a suspended transaction that needs to be resumed.

- *suspend*: this method is useful if an active transaction exists, but you need to start a new transaction that is independent. By suspending the current transaction, you dissociate it from the current thread and are free to begin a new one, whose commit or rollback will not affect the current transaction. If you want to have another thread continue the current transaction then this method can be used (in combination with resume) to 'pass on' the transaction to another thread.
- *resume*: this method (re-)associates the calling thread with an *existing* transaction (typically one that was suspended first). If you continue a transaction in a different thread, then that thread should call this method with the transaction as an argument. If you have done some intermediate work in a *different* transaction, then this method can be called to resume the original transaction.

2.2.2. ExportingTransactionManager

The ExportingTransactionManager is where a client that runs Atomikos TransactionsEssentials can get a Propagation instance for adding it to a remote call. It is also the place where any returned Extent can be added to the client's transaction.

Even if you created a JTA transaction through the JTA TransactionManager in the AtomikosJTA API, then you can still export that transaction by using this interface. This is because all transaction manager classes merely represent different views on the same underlying transaction object.

- *getPropagation*: this method returns a propagation for the transaction that is associated with the current thread. The Propagation can be shipped to remote servers because it is Serializable.
- *addExtent*: call this method for a returned Extent, but only if your application wants the remote work to be committed in the end. If you ignore the returned extent of a remote call then the corresponding work will eventually be rolled back by the remote transaction service (due to timeout).

2.2.3. ImportingTransactionManager

The ImportingTransactionManager is the interface that a server uses when it wants to become part of a remote client-side transaction (whose Propagation was received as part of an incoming invocation). It is also where an Extent can be retrieved in order to return that Extent to a remote client (as part of the return data of a finished call).

Even if you are using the regular JTA API in the rest of your application, you can still use this interface to import a Propagation received from a remote VM. This will create a local transaction

that will immediately associate the thread with a JTA Transaction instance as well! This instance will then be accessible through the `javax.transaction.TransactionManager` available via the `com.atomikos.icatch.config.UserTransactionService`. Of course, if you prefer to use the Atomikos TransactionsEssentials API then the `CompositeTransactionManager` will also work.

- *importTransaction*: if your application receives a `Propagation` as part of an incoming request's parameters, then you can use this method to associate the current thread with the transaction at the client side. This association is done in the form of a subtransaction that is local to the server, and a descendant of the client's transaction. Any resources accessed can later become subject to commitment of the root transaction (if the `Extent` is received by the client after the call is done). As part of the import, you also have to specify two boolean parameters that are not part of the client's request. The first one is *orphancheck*: if you want to use Atomikos' built-in support for detection of pending (lost) transactions then you should specify *true* here. The second parameter is *heur_commit*; this specifies your preference when the transactional work for this import remains indoubt for too long. If you specify a value of *true* then a possible indoubt will be resolved with a heuristic commit by the transaction service. If you specify *false* the a heuristic rollback will be done instead. Heuristic decisions are always made after timeout of an *indoubt* transaction.
- *terminated*: call this method to indicate that an imported transaction's thread is about to be finished, and wants to perform subtransaction commit or rollback. This method will dissociate the subtransaction from the current thread and return the `Extent` that should be shipped to the client.

2.2.4. Propagation

The `Propagation` interface (which extends `java.io.Serializable`) is meant for propagation of transactions from one server VM to another (assuming that both server VMs have Atomikos TransactionsEssentials running in them). The propagation contains the necessary information to create (at the receiver) a subtransaction of the sender's transaction and also propagates the settings for timeout and concurrency of subtransactions. It is unlikely that your application will ever need to manipulate this interface; all that typically needs to be done is retrieving it at the sender and registering it with the transaction service of the receiver. This usually means that it has to be passed as a parameter to any remote calls that you make.

The propagation mechanism is subject to several Atomikos patents and pending patents not included in the license for this product.

2.2.5. Extent

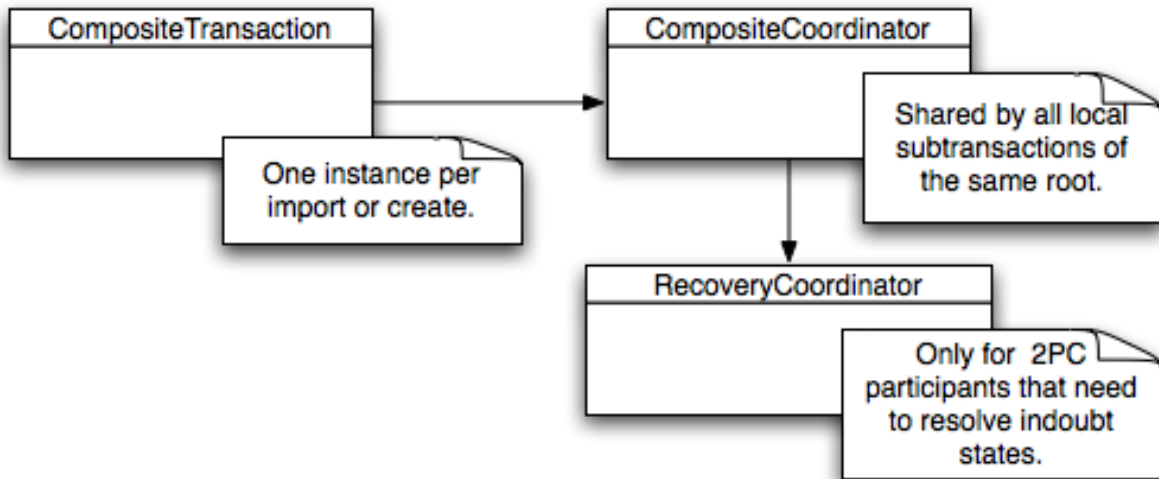
The `Extent` interface is the counterpart of the `Propagation`: it should be returned from a server that has previously imported a propagation. The information it contains is essential for the server's subtransaction work to become part of the two-phase commit termination of the root transaction. Like the `Propagation`, this interface should not be manipulated by your application: retrieving it at the server and registering it at the client's transaction service are all that is needed. This implies that you will have to account for this information to be contained in your remote server's returned data set.

For remote calls that fail, you should not care about any returned `Extent`. Neglecting the `Extent` implies that the work done at the server (if any) will not become part of the root transaction's commit, meaning it will timeout and rollback. Note that this behaviour is consistent with application-level semantics for such scenarios. Using this mechanism ensures that failed remote calls do not affect the persistent state of the remote server (a property that is missing in most regular RPC-style interactions).

The extent mechanism is subject to several Atomikos patents and pending patents not included in the license for this product.

2.3. Atomikos' CompositeTransaction Model

This section discusses the component classes that are related to individual transactions. Instances of these classes are provided by Atomikos. The big picture is shown in the figure below...



2.3.1. CompositeTransaction

The transaction interface allows manipulation of an active transaction. The most important role of this interface is to add work to the scope of the transaction, thereby making the outcome of the work depend on the outcome of the transaction. In addition, this interface allows comparison of different transactions, which can be used by resources to determine if and when different accesses should see each other's updates. The functionality of the transaction interface is discussed below.

The **CompositeTransaction** interface is the transaction kernel's notion of a transaction. It is the underlying transaction model for all other APIs. For instance, if you use our JTA then the *javax.transaction.Transaction* instances that you create will correspond to exactly one **CompositeTransaction** in the core.

- *isRoot*: True if and only if the instance is a root transaction (i.e., a top-level transaction: one that has no parent transaction).
- *getLineage*: This method returns a stack of ancestors, or an empty stack for a root transaction. The stack of ancestors can be useful for implementing lock inheritance among related transactions: a resource can do this by adding itself as a *SubTxAwareParticipant* to the appropriate parent in case related transactions are competing for access to common data.

Whenever a propagation is imported in a VM, the transaction service automatically creates a new subtransaction for the imported transaction. This subtransaction then becomes associated with the

importing thread. Depending on the particular propagation policy, an implementation may limit the lineage information to the *root transaction only*. So you should *not* count on *all ancestor transactions* always being represented in the lineage.

- *getTid*: a getter for the unique ID of the transaction.
- *isAncestorOf*: This method tests if one transaction is an ancestor of another. Resources that want to implement lock inheritance can use this method.
- *isDescendantOf*: This method tests if one transaction is a descendant of another. Resources that want to implement lock inheritance can use this method.
- *isRelatedTransaction*: This method tests if one transaction is a child of the same root as another one. Resources that want to implement lock inheritance can use this method.
- *isSameTransaction*: This method tests if one transaction represents the same work as another one. Resources can use this test to optimize their locking strategy: when an access to locked data is attempted, this method can be used to test if the new access is on behalf of the same transaction as the one that has set the lock.
- *getCompositeCoordinator*: to get the coordinator interface for this transaction. The coordinator represents the common root part for different local transactions: different local subtransactions of the same root (either a remote root or a local one) will have the same coordinator instance. This coordinator represents the overall work done by all subtransactions of the same root, and all this work is subject to the same two-phase commit termination (as required by subtransaction semantics).
- *addParticipant*: A more powerful variant of the *enlistResource* in JTA. This method serves to add work to the transaction, but the exact definition of what 'work' is, is much wider in Atomikos' APIs. See the Participant interface for more details.
- *registerSynchronization*: Call this method to register a synchronization object that will receive callbacks on transactional (two-phase commit) termination. Note: the registration of a synchronization does *not* survive crashes! Consequently, you should not use this method to perform crash-recovery related tasks; if that functionality is needed then you should use the Participant interface instead.
- *addSubTxAwareParticipant*: With this method, you can register for the commit/rollback events of a (sub)transaction. Note: the name of this interface can be somewhat misleading: typical usage consists in resources waiting for some transaction's parent to finish because a related transaction wants to access the same data. By waiting for the right parent to commit/rollback, true lock inheritance can be implemented.

This callback mechanism also works for remote transactions, i.e.: *CompositeTransaction* instances that are part of the lineage and that represent a transaction that was imported from a remote server.

- *isSerial*: An optimized mechanism for implementing lock inheritance is merely testing if the transaction is being done serially or not. This property can only be set at the root level, and resources can allow shared access by two related transactions if they are serial. This is safe because the absence of concurrent behaviour among subtransactions also means that there is no need to protect them from corrupting each other's data (by concurrent interference).
- *isLocal*: Tests if the transaction represents local work, or is an imported instance instead. Atomikos TransactionsEssentials always creates a new and local subtransaction for each imported transaction. The imported transaction is represented (in the lineage) by a proxy that also adheres to the *CompositeTransaction* interface. For proxy instances, *isLocal* will return false.
- *createSubTransaction*: For the creation of a subtransaction of the corresponding transaction.

This method creates a subtransaction, but it does *not* associate the thread to the new subtransaction. In particular, calling `getCompositeTransaction()` on the `CompositeTransactionManager` will still return the old transaction. If you need to associate the thread with the new subtransaction, either use `CompositeTransactionManager.resume` or create the subtransaction by calling `CompositeTransactionManager.createCompositeTransaction`.

- *setSerial*: When called for a root transaction, this method will mark the entire transaction tree (the root and its descendants) as serial, meaning that subtransactions should not execute concurrently. This allows easy lock inheritance to be offered by resources. It should be noted that it is up to the application to respect the required serial behaviour; Atomikos TransactionsEssentials can not do a lot more to prevent malicious applications from tampering with the required behaviour. The only safety provided is that once set, the serial flag can not be unset. Also, the value of this flag is included in any `Propagation` instance.
- *getLocalSubTxCount*: This method returns the number of local subtransactions that were locally started.
- *setRollbackOnly*: Calling this method marks the transaction so that its only possible outcome is rollback.
- *setTag*: With this method, the high-level description of the work of this (sub)transaction can be set. After commit of the subtransaction, this tag will then become part of the returned list of tags upon calling `getTags` in the `CompositeCoordinator` interface.
- *getExtent*: Returns the extent of the subtransaction; this represents the remote work done on behalf of the subtransaction. The system uses this information for two-phase commit purposes.
- *getTimeout*: Get the (remaining) timeout of the transaction.
- *set/getProperty*: Set/get meta-information (properties) on the transaction. Properties can be used to distinguish different types of transactions, like ACID transactions or activities. Properties are inherited by subtransactions and may be propagated along with remote calls.
- *rollback*: Perform rollback of the effects of the (sub)transaction.
- *commit*: Commit the effects of the (sub)transaction.

For a root transaction, calling `commit` will trigger two-phase commit among all participants. For a subtransaction, calling `commit` will merely cause the commit of that subtransaction, the work still being subject to a later two-phase commit triggered by a commit at the level of the root transaction. The rollback of a subtransaction will merely trigger rollback of all its registered `Participant` instances, without affecting any other subtransactions.

2.3.2. CompositeCoordinator

The composite coordinator represents the common two-phase commit part of all local subtransactions of the same root transaction. By nested transaction semantics, these subtransactions' `Participant` instances must all be subject to the same two-phase commit outcome that determines the permanent commit of the root and all its subtransactions. The composite coordinator represents this fact.

- *getCoordinatorId*: This method returns the globally unique ID of the root transaction (the top-level transaction), also called the coordinator.

- *getRecoveryCoordinator*: This method returns a handle that can be useful for resolving remaining in-doubt participants. When a prepared participant remains in-doubt, it can ask 'its' RecoveryCoordinator for replay of two-phase commit outcome.
- *getTags*: This method returns a list of HeuristicMessage instances that describe the local work being done by all subtransactions of the same root. More precisely: this will return all tags set for *committed* subtransactions that share the same CompositeCoordinator instance. This is useful for getting comments about the effect of the joint work of all such committed subtransactions. Because rolledback subtransactions have no effect by definition, their tags are not included.
- *setRecoverableWhileActive*: Indicates that the coordinator should be logged even during its active phase. By default this is false (i.e., coordinators are recoverable only after the prepare phase by default), but long-lived transaction models can set this to true. The active recovery property is inherited by all later subtransactions.
- *isRecoverableWhileActive*: Tests whether this coordinator is recoverable in its active phase.

Subtransactions that are rolled back will *not* affect their CompositeCoordinator instance: Participant instances that were added to a subtransaction prior to that subtransaction's rollback will not take part in the global two-phase commit protocol.

2.3.3. RecoveryCoordinator

Prepared participants that remain in-doubt can use the RecoveryCoordinator instance (returned during addParticipant or obtained from the CompositeCoordinator) to try to resolve their ignorance of the two-phase commit outcome.

- *replayCompletion*: The participant instance can call this method with itself as an argument in order to try and resolve its in-doubt status. Note that this method is not guaranteed to work; in particular, if the party that the recovery coordinator represents has become unreachable (due to network failures, for instance) then the participant will not learn the outcome even though it uses this method. In that case, the participant is free to apply a heuristic termination.
- *getURI*: Get the unique URI identifier for this coordinator.

2.3.4. HeuristicMessage

The heuristic message is an important interface for resolving heuristic transaction termination states. By adding instances of class StringHeuristicMessage, your application can influence the kind of information available in the transaction logs: any information contained in these added instances will also be in the logs whenever a heuristic transaction has occurred. The LogAdministrator tool will then allow you to view that information whenever you need to.

2.3.5. SubTxThread (in package com.atomikos.icatch.system)

This class is a utility class for implementing different parallel threads that perform work on behalf of the same parent transaction. For each such thread, you can create one instance of this class. Because this class extends the java.lang.Thread class, you can start it like a regular thread. The SubTxCode argument supplied to the constructor will execute in a *subtransaction* of the transaction that the calling thread is associated with.

- *constructor* The constructor takes three arguments: a Waiter object to synchronize on, a SubTxCode with the execution logic for the thread, and a HeuristicMessage to document the work of the thread.

- *getException* After the thread has finished (i.e., after the Waiter instance's *waitForAll* has returned), this method can be used to retrieve any Exception that happened during the execution of SubTxCode's logic.
- *run* This method overrides the default thread behaviour by delegating to SubTxCode. You should not call this method directly; it will be called by calling *start* on the SubTxThread instance.

2.3.6. Waiter (in package com.atomikos.icatch.system)

This is a complementary class to the SubTxThread class. For a given set of parallel SubTxThread instances (that belong to the same parent transaction), you can use a Waiter object to wait for all these threads to finish. It is considered bad style to start subtransaction threads without waiting for them to terminate.

- *waitForAll* Calling this method will block the calling thread until the subtransactions have all finished.
- *getAbortCount* After calling *waitForAll*, this method can be used to retrieve the number of failed subtransactions.

2.4. Plug-In Interfaces

2.4.1. TSListener

This interface is useful for applications that want to be notified of transaction service startup and shutdown. Instances that implement this interface can be registered via the *UserTransactionService*. Registration can be done before or after startup, but in the latter case no start callbacks will be received. The following methods must be implemented:

- *init*: called during the startup of the transaction service, once before and once after initialization of the recovery service.
- *shutdown*: called before and after shutdown is about to happen.

2.4.2. SubTxCode

Implementations of this interface provide logic that you want to execute in a (parallel) thread that behaves as a subtransaction of the parent thread. This interface has only one method: *exec* that takes no parameters. The *exec* method is called by the SubTxThread instance inside the *run* method. By convention, a successful termination of *exec* will trigger commit of the subtransaction it executes in, whereas an exception will trigger rollback of the corresponding subtransaction. If an exception occurs, then it can be retrieved through the SubTxThread's *getException* method.

2.4.3. Synchronization

This interface is a means to register an application-level callback; it allows the application to be notified upon commit events. You can use this functionality by *implementing this interface in your application*.

Note: synchronizations are not persistent; after a crash, any recovered transactions' synchronizations will be lost. This is not in contradiction with their intended usage.

- *beforeCompletion*: this method is called before the transaction will start its commit. A typical usage of this method is to write pending updates to the database.

- *afterCompletion*: this method is called after the commitment was done, and indicates whether it was successful or not.

2.4.4. SubTxAwareParticipant

This interface is useful for applications that want to implement full lock inheritance according to the nested transaction model. Such applications can use the callback methods in this interface to trigger the granting of locks to related, concurrent transactions.

- *committed*: this method is called when a transaction that this participant was registered to has committed. The argument specifies which transaction that is (useful in case this instance registered with multiple transactions).
- *rolledback*: this method is called when a transaction that this participant was registered to has rolledback. The argument specifies which transaction that is (useful in case this instance registered with multiple transactions).

2.4.5. Participant

The Participant interface is the core representation of any transactional work done within the scope of a CompositeTransaction. It represents the core functions that the transaction kernel expects in order to perform its two-phase commit protocol. Implementations of this interface may choose to initiate heuristic termination on their own; this would lead to heuristic exceptions during two phase commit.

The participant mechanism is subject to several Atomikos patents and pending patents not included in the license for this product.

- *prepare*: called during the prepare phase. The instance should prepare according to two-phase commit semantics. If this method returns without exceptions then a *YES* vote is assumed by the transaction manager. For instances that represent read-only access, *READ_ONLY* can be returned to indicate that no later commit or rollback should be called. If the instance can not prepare successfully (for instance, because internal timeout has rolled back the work) then a *RollbackException* should be thrown. Internal failures may lead to a heuristic exception (as explained in the Exceptions section).
- *rollback*: called when the transaction manager has decided that the work should be rolled back. Implementations can return a list of *HeuristicMessage* instances that have been added. How such messages have been added to the Participant falls outside the scope of this specification; the transaction manager does not care where they come from. Internal timeouts or errors may lead to any of the declared heuristic exceptions being thrown, provided that a previous prepare has been performed.
- *commit*: called to signal a commit decision by the transaction manager. Implementations can return a list of *HeuristicMessage* instances that have been added. If there is only one Participant instance then this method will be called as part of one-phase commit (without prior prepare). In that case, internal timeout may have lead to rollback already, which triggers a *RollbackException*. In two-phase commit cases this method will be called after a prior prepare. In those cases, an internal heuristic timeout may lead to any of the declared heuristic exceptions.
- *forget*: called in case of a heuristic termination; this signals to the underlying implementation that any remaining logged state data on behalf of two-phase commit can be discarded.
- *getHeuristicMessages*: called by the transaction manager when heuristic decisions have been made. This method should return all heuristic messages known about the transactional work in this instance.
- *recover*: this method is called upon recovery, when the transaction logs are being read. The method allows participants to re-initialize their state and make sure that any external references can be reconstructed. Failure to do so should result in *false* being returned.

- *setCascadeList*: this method is called prior to prepare, in order to deal with the orphan detection information that the core passes on. This method should normally be left empty by implementations other than the core's internal implementation.
- *setGlobalSiblingCount*: this method is also called prior to prepare, in order to deal with the orphan detection information that the core passes on. This method should normally be left empty by implementations other than the core's internal implementation.
- *getURI*: Get a unique URI identifier for this participant, or null for local instances.

The Atomikos APIs were designed to be capable of dealing with generalized rollback scenarios such as compensation. Therefore, you should not be misled by the traditional meaning of rollback (which is classically interpreted as state-based restoration of a before-image of data). The APIs and the transaction service do not care about how you perform rollback, and the transaction service is in fact nothing less than a general two-phase commit engine. If you decide to implement a Participant for your application's needs, then feel free to implement rollback any way you like.

2.4.6. RecoverableResource

This interface (in package `com.atomikos.datasource`) is essentially a helper class for Participant recovery. You can provide an implementation if you have a custom Participant that needs external help during recovery, typically for resolving external references.

- *setRecoveryService* this method is called when the resource is initialized by the transaction service. The corresponding argument is a handle to the transaction service that the resource can use to provide help during recovery. In particular, the resource can use the recovery service to trigger recovery, and it can also find out the unique name of the transaction service (useful in determining what internal resource transactions are within the scope of recovery by this transaction service). It is assumed that the resource uses some internal mechanism to identify its resource transactions based on this unique name.
- *recover* this method can be called by your Participant implementations, to delegate Participant.recover functionality to this helper class. The return value should be true only if the resource can effectively help the Participant in recovering. If false, then the Participant may ask any other resources it knows, until one returns true. Resources can be looked up by their unique name.
- *getName* The unique name of the resource.
- *isSameRM* True iff the two instances represent the same resource. Resources are the same if they recover the same set of Participant implementations.
- *endRecovery* Called by the transaction service on each registered resource. This is an indication that recovery has been done, and that any remaining non-recovered Participant instances (internal to the resource) can be safely rolled back according to the *presumed abort* paradigm.
- *close* Called by the transaction service to indicate that the transaction core is about to shut down. This method can be used to cleanup internal resources such as connections or files.

2.5. Exceptions

This section discusses the different exceptions that are defined and used in the Atomikos TransactionsEssentials API. They are divided into two categories:

- Checked Exceptions
- Runtime Exceptions

2.5.1. Checked Exceptions

2.5.1.1. HeurCommitException

Thrown by a Participant when rollback is called on a heuristically committed instance. Also thrown by the CompositeTerminator's commit upon failure: when all Participants for the transaction have been in-doubt for too long, they may have committed the transaction although all replied positively during the prepare of two-phase commit. If the transaction manager later re-establishes contact and instructs the Participants to rollback then this exception will be thrown to the application. It indicates an anomaly in the transaction's outcome, where *all Participants involved* have chosen to commit heuristically, because all were left in-doubt. If you get this exception, it means that the entire transaction has been committed, although rollback was desired.

2.5.1.2. HeurRollbackException

Thrown by a Participant when commit is called on a heuristically rolledback instance. Also thrown by the CompositeTerminator's commit upon failure: if all Participants have decided to rollback although the final decision of the transaction manager was to commit. This is similar to the previous case; this time it means that the entire transaction has in fact been rolled back whereas the desired outcome was commit.

2.5.1.3. HeurHazardException

Thrown by a Participant for internal failures in any of the two-phase commit methods, when the instance can not determine the exact outcome. When thrown by the CompositeTerminator, this exception indicates that some Participants may not have received the final commit or rollback decision at the end of two-phase commit (because the transaction service could not establish contact after prepare). This means that there is a danger of heuristic termination if those Participants decide unilaterally after a timeout.

2.5.1.4. HeurMixedException

A participant can throw this exception on any of the two-phase commit methods. For the CompositeTerminator, this method can be thrown during commit. This is the most complex error, where some of the resources may have committed and others have rolled back. It hints that the transaction's effects are only partial; this is a clear violation of transactional semantics. More information should be in the logs.

2.5.1.5. RollbackException

Thrown by a Participant to indicate a *NO* vote on prepare, or when one-phase commit is requested for a timed-out instance. At application level, this error is thrown if a transaction is requested to commit (through the terminator or through the ImportingTransactionManager) when it has already been rolled back due to a timeout. If you get many such errors, then it may be helpful to create transactions with a higher timeout value.

2.5.2. Runtime Exceptions

2.5.2.1. SysException

This is the general internal error that is thrown by many transaction service components for exceptional conditions. Instances of this exception contain nested error information, which can be viewed by printing the stack trace.

2.5.2.2. ResourceException

This exception (in package *com.atomikos.datasource*) is thrown by the *RecoverableResource* instances. This class is a subclass of *SQLException*, so you can also analyze the detailed error stack in the same way.

2.5.2.3. UnavailableException

This error is thrown mostly by local proxies of remote (imported) transactions. In particular, if you are trying to perform an unsupported operation on a transaction found in the lineage (the ancestor stack) of a transaction instance, then this exception will be thrown.

Chapter 3. Programming with the Atomikos TransactionsEssentials API

The purpose of this chapter is to clarify some important aspects of programming with our API. The structure of this chapter is as follows:

- Basic Level Programming
- Intermediate Level Programming
- Advanced Level Programming

The basic level deals with essential and minimal API usage in order to import or export a transaction. This is the minimal knowledge you will need if you use the JTA interfaces for everything else. The intermediate level shows the effects of commit or rollback through our API, as well as how to use synchronizations and perform parallel calls within the same transaction. The advanced level goes into lock inheritance, and how to implement custom Participant types and resources.

3.1. Basic Level

This section explains the basic usage of Atomikos TransactionsEssentials API, meaning those components whose functionality is not contained in the JTA interfaces. If you don't plan to do special things with Atomikos TransactionsEssentials (except shipping transaction context to another VM or importing a transaction from a remote VM) then this section should be all you need to read. Exporting/importing existing transactions is needed for architectures where different servers (in different VMs) need to co-operate within the context of the *same* transaction. Whenever one server calls another server within the scope of a transaction, the calling server will export the transaction (by adding a Propagation object to the arguments of the call) and the called server can then import it. Import is discussed in the next section; here we focus on exporting a transaction.

Exporting a transaction can be done through the *ExportingTransactionManager* interface. An instance can be gotten through the *UserTransactionService*. A typical code example for export is shown below.

Example 3.1. Typical export code fragment

```
//the result to be returned
//must be wrapped in a holder
//because the call should also
//return the extent.
//the extent contains the information
//needed for 2PC
ResultHolder res = null;
UserTransactionService uts =
    new UserTransactionServiceImp();
ExportingTransactionManager exptm =
    uts.getExportingTransactionManager();
Propagation propagation = exptm.getPropagation();
```

```
try {

    //call remote server with
    //propagation as argument.
    //the result is wrapped in a
    //ResultHolder

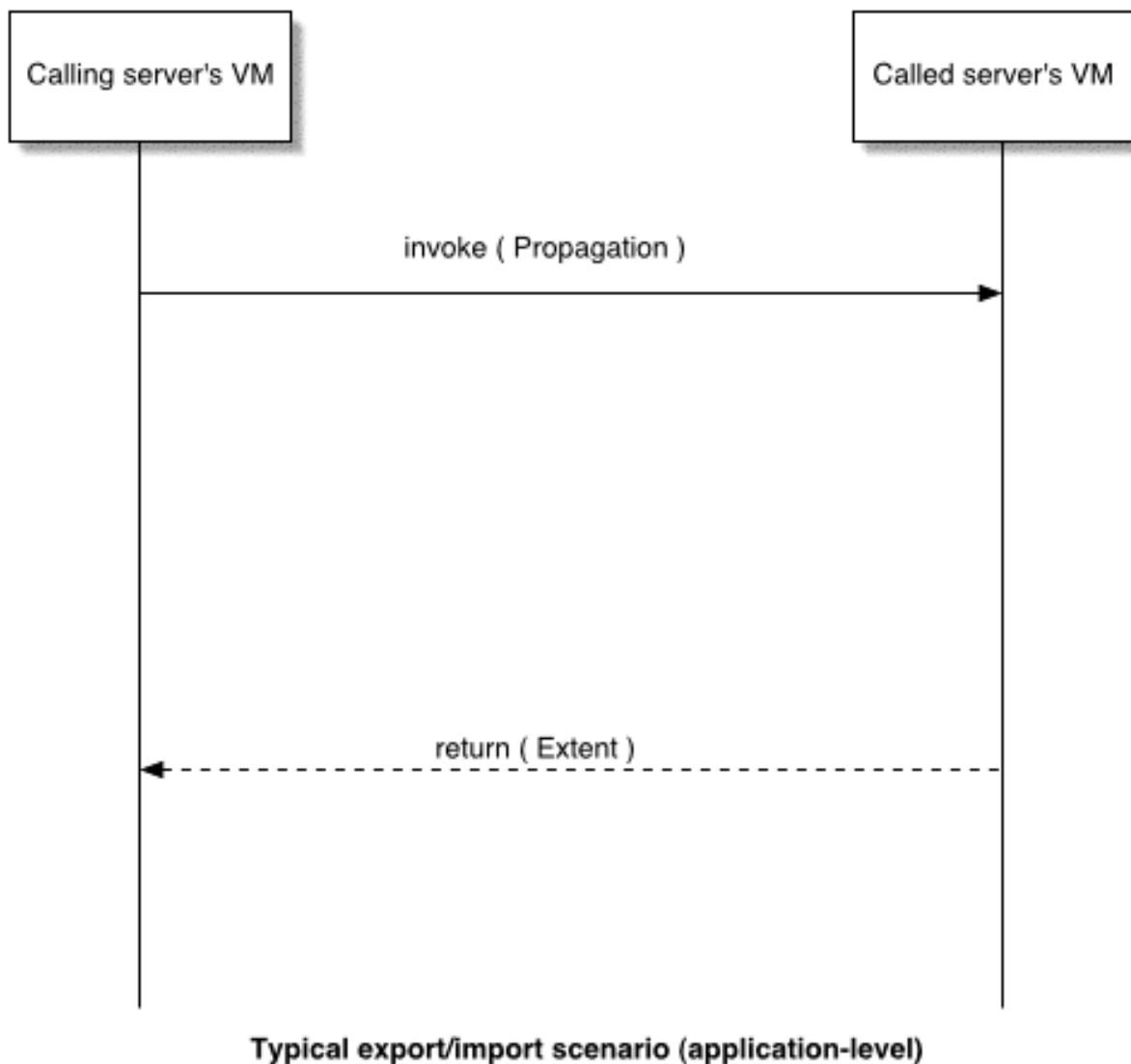
    res = ...
    //the return value is wrapped
    //in res!

    //here we are if call returned
    //without exception.
    //add the extent of the
    //call to the current tx

    exptm.addExtent ( res.extent );
}
catch ( SysException se ) {
    //error on adding the extent?
}
catch ( Exception e ) {
    //if you have a non-SysException, then
    //you may safely act as if the
    //call had never been executed.
}
```

The Propagation contains information about the nested structure of the distributed transaction, as well as its execution mode (serial or concurrent). You can export a transaction at any time as long as it is active (i.e., rollback or commit have not been called). Note that the Propagation interface extends *java.io.Serializable* so it can be passed over socket connections, RMI calls and so on.

It is important to understand that export is a two-way process: you do not only add the Propagation to the outgoing call, but you also need to retrieve the *Extent* from the return value of the call. This extent must then be added to the ExportingTransactionManager interface. The extent is used for two-phase commit and for detection of orphan transactions, and it is a crucial part of ensuring exactly-once semantics of the distributed computation. The following picture is an illustration of the application-level import/export scenario...



NOTE: Although JTS/OTS defines the mechanism of exporting/importing a transaction, the mechanism explained here is Atomikos-native. This is necessary because many of the platforms that we support are not supported by typical CORBA ORBs, nor by the original OTS specifications.

3.1.1. Importing an existing transaction

If a server receives a call with a *Propagation* as one of the arguments then it should import the corresponding transaction in order to "tie" the local work into the global transaction. This is done through the *ImportingTransactionManager* interface, also available through the *UserTransactionService*. A typical code example for import is shown below.

Example 3.2. Typical import code sample


```
//assume a Propagation has been supplied as a parameter
//called 'propagation'

//the result to be returned must be wrapped in a holder
//because the call should also return the extent.
//the extent contains the information needed for 2PC
//NOTE: the holder class is not part of the distribution.
ResultHolder res = new ResultHolder();

//for termination: success or not? -> no if exception
boolean success = false;

//heuristic termination should mean commit
boolean heuristicCommit = true;

//the system should check for orphans
boolean orphanCheck = true;

UserTransactionService uts =
    new UserTransactionServiceImp();

ImportingTransactionManager imptm =
    uts.getImportingTransactionManager();

try {
    //import the transaction, and associate it with current thread
    imptm.importTransaction (
        propagation,
        orphanCheck,
        heuristicCommit );

    //execute business logic here...
    res.object = ... //the return value is wrapped in res!

    //if everything ok: set committable as the
    //last part of this block
    success = true;
}
catch ( Exception e ) {
    success = false;
}
finally {

    //ALWAYS properly terminate the imported transaction!
    //By setting the extent, a call
    //that returns fine will inform the client of
    //the information needed for 2PC.
    //Note: for exceptions, the extent will
    //not be received (because of no return value)
    //but that is absolutely fine.

    res.extent = imptm.terminated ( success );
}
```

```
}
return res;
```

Whenever you import a transaction this way, a *local* transaction is created that is automatically a *subtransaction* of the calling transaction. This happens transparently and allows Atomikos to provide the necessary quality of service.

IMPORTANT: make sure that any transaction that is imported will also be terminated. This is best done in a finally-block.

3.2. Intermediate Level

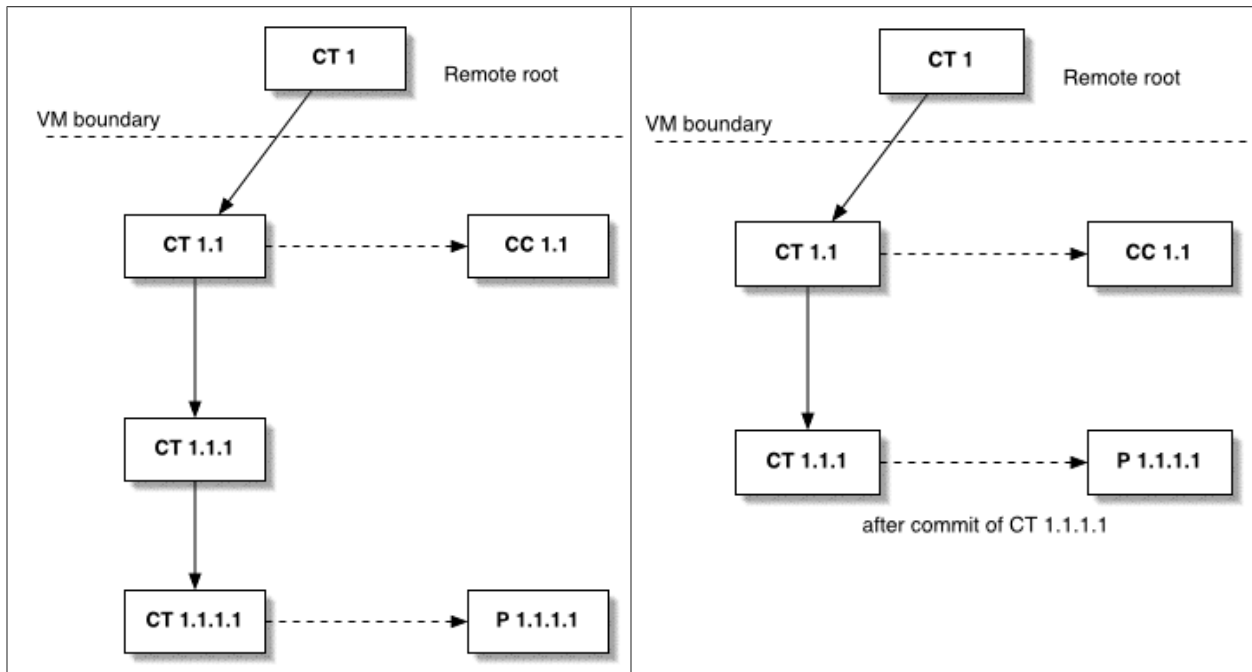
This section is about intermediate-level programming with our API. Read this if you want to understand the effects of commit/rollback of subtransactions, if you want to be notified of transaction events or if you need parallel subtransactions in your application.

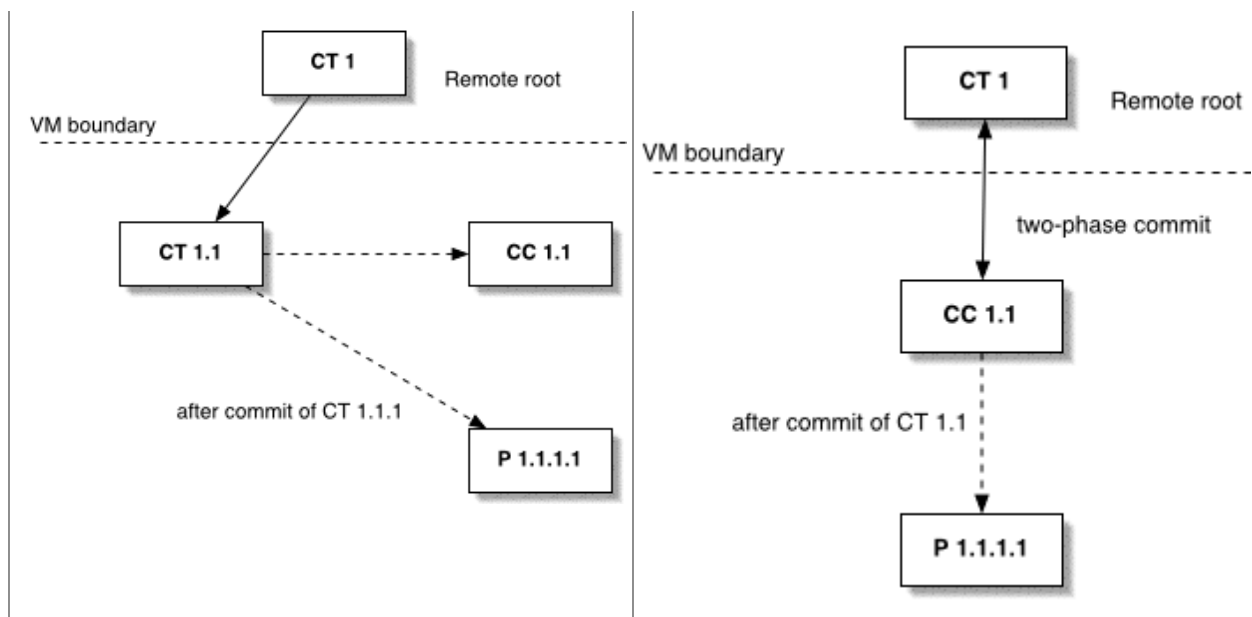
3.2.1. Subtransaction Commit

When a subtransaction is committed, its Participant instances are added to the set of Participants of the parent transaction. Alternatively, if there is no *local* parent then the participants become part of the CompositeCoordinator's set of Participants eligible for two-phase commit.

The following figures illustrate this. The case shown is for a remote root (called CT1). Upon import, a local subtransaction CT1.1 is created. For this example we assume that the application also creates a number of subtransactions, up to CT1.1.1.1. Now, suppose a Participant (called P1.1.1.1) is added to this last subtransaction. After commit, the Participant is added to the parent transaction. When the parent transaction in turn commits, the P1.1.1.1 is again propagated upwards in the hierarchy. Finally, the commit of CT1.1 will result in P1.1.1.1 becoming part of the CompositeCoordinator CC1.1, which will (internally) wait for two-phase commit events and delegate those to its registered Participants. In this case, P1.1.1.1 will take part in two-phase commit.

Table 3.1. Subtransaction commit

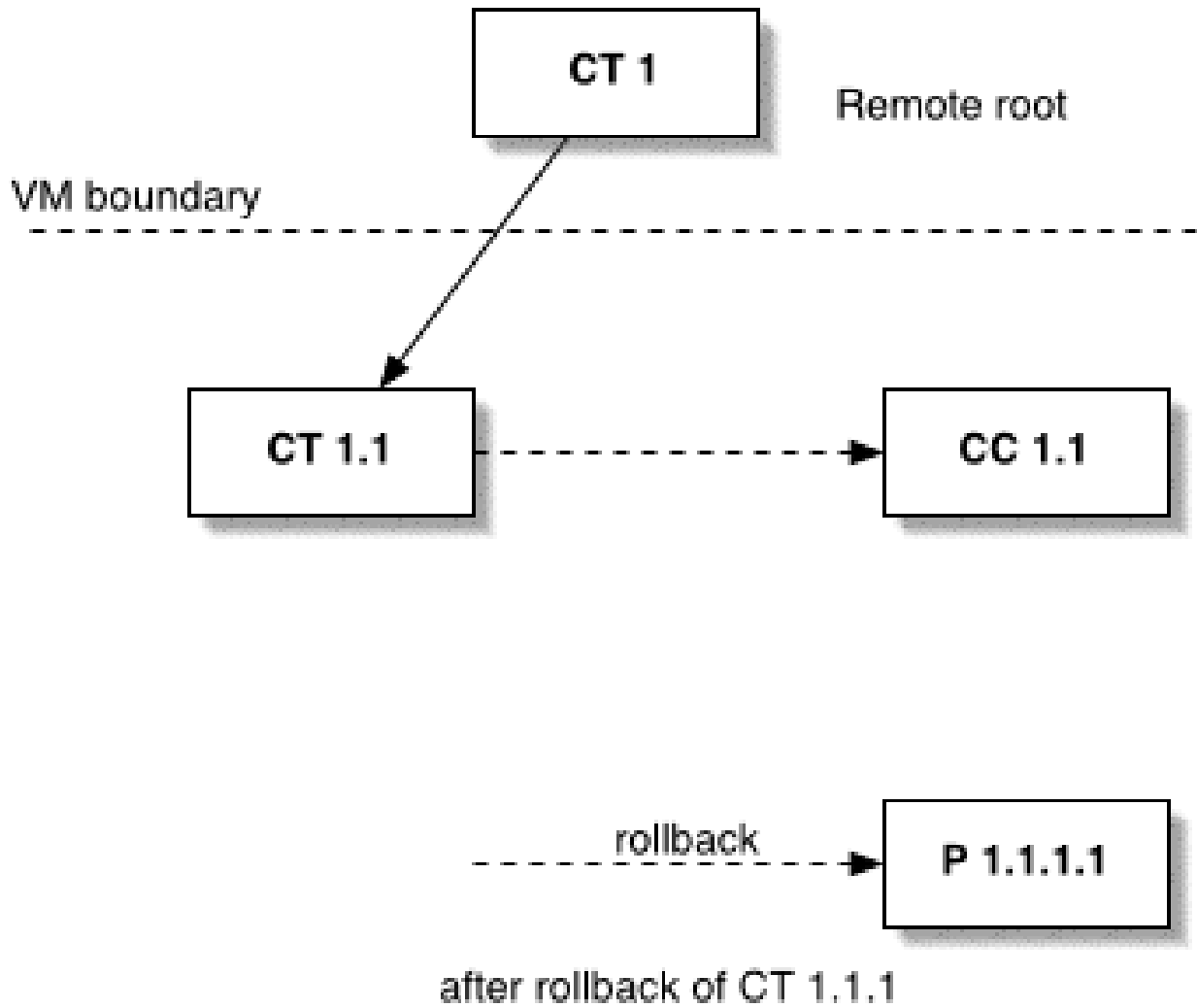




Note that the commit of a subtransaction does *not* trigger any method of the Participant. This is delayed until the global two-phase commit is triggered by the root transaction.

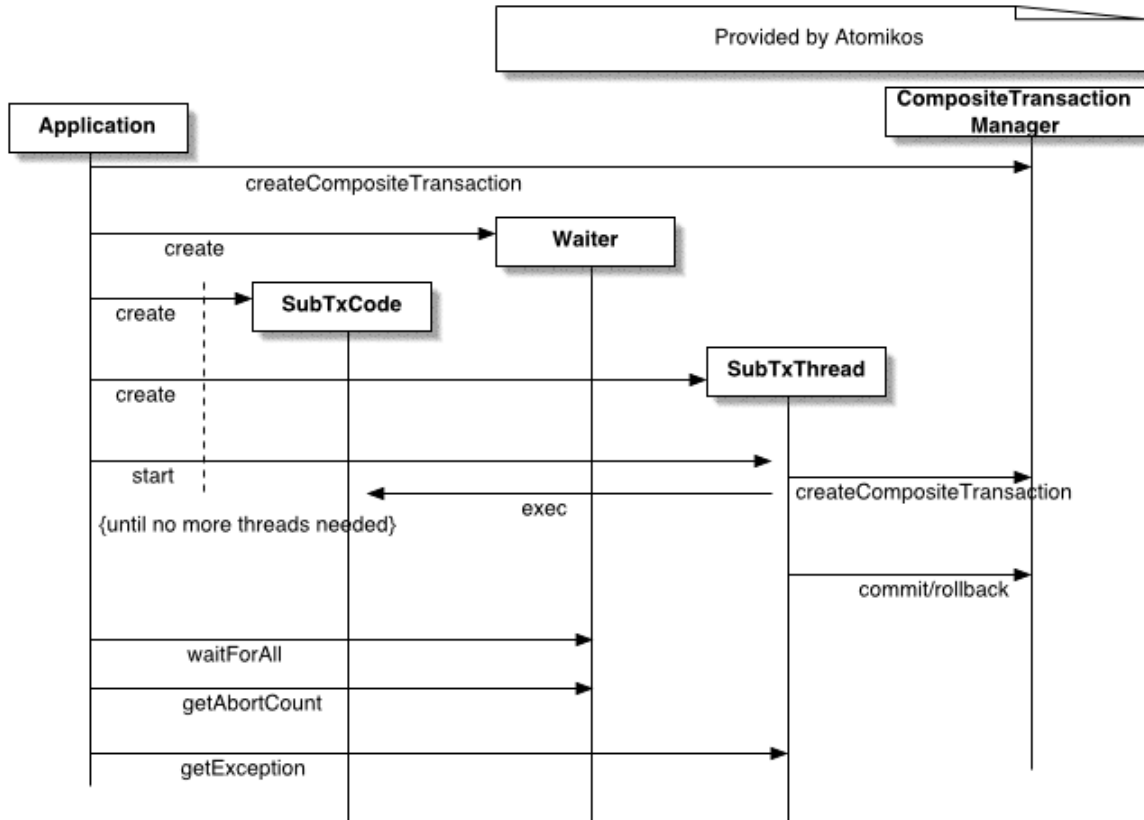
3.2.2. Subtransaction Rollback

If one of the subtransactions is rolled back, then the Participant **P 1.1.1.1** will not be added to the two-phase commit set of the CompositeCoordinator. This is illustrated in the next figure: rollback of **CT 1.1.1** implies that **P 1.1.1.1** is also rolled back, and no Participant is propagated to the parent of **CT 1.1.1**. This way, subtransactions can be rolled back without affecting the work of the parent transaction.



3.2.3. Implementing Parallel Subtransactions

If you want to implement parts of a transaction that execute in parallel threads (such as remote calls to different resources) then you can use the utility classes *SubTxThread* and *Waiter* and supply your own *SubTxCode* logic for each thread. The following picture illustrates the interactions between these:



In summary, the things you have to program are the following:

1. Start a transaction. The SubTxThread paradigm only works for an existing transaction.
2. Create a Waiter instance for synchronization. Typically, only one instance is needed.
3. For as many threads as needed, do the following:
 - Create a SubTxCode instance (which you have to provide implementations for). This encapsulates your application specific logic for the thread.
 - Create a SubTxThread with the Waiter and SubTxCode as arguments.
 - Start the thread.
4. Wait by calling the Waiter objects's `waitForAll` method.
5. Inspect the number of aborts, and decide what to do (commit or rollback the parent transaction depending on your own preferences).

The SubTxThread/SubTxCode/Waiter model is only provided for your convenience: it is not necessary to use them if you don't want to. You could design your own mechanism instead.

3.2.4. Adding Synchronization Implementations

Add a synchronization to a CompositeTransaction to have a notification about two-phase commit outcome. The most important use of synchronizations is for writing out persistent state before transaction commit (upon

notification through the *beforeCompletion* method. It is important to note that implementations should *not* rely on being called for recovered transactions. This does not invalidate the usefulness for the writing of persistent state before prepare time, because prepare is only performed for non-recovered transactions.

The only way of being notified about every two-phase commit event is by registering as a Participant with the CompositeTransaction. The synchronization is only a straightforward and light-weight alternative for particular application-level needs.

3.2.5. Adding SubTxAware Implementations

The primary usefulness of SubTxAware is for being notified about any (sub)transaction's preliminary commit or rollback. The most obvious need exists in cases where one wants to provide full lock inheritance to related subtransactions.

The SubTxAwareParticipant is different from the Participant interface because the former does not receive *two-phase commit* notifications. Rather, it only receives notification of *subtransaction* commit or rollback, which happens before any two-phase commit. An implementation should *not* rely on the terminating subtransaction still being associated with the calling thread, not even if the implementation is in the same VM as the transaction it is registered with. If the implementation needs information about the transaction that commits or rolls back, then it should do so by inspecting the argument of the callback method.

3.2.6. Adding Participant Implementations

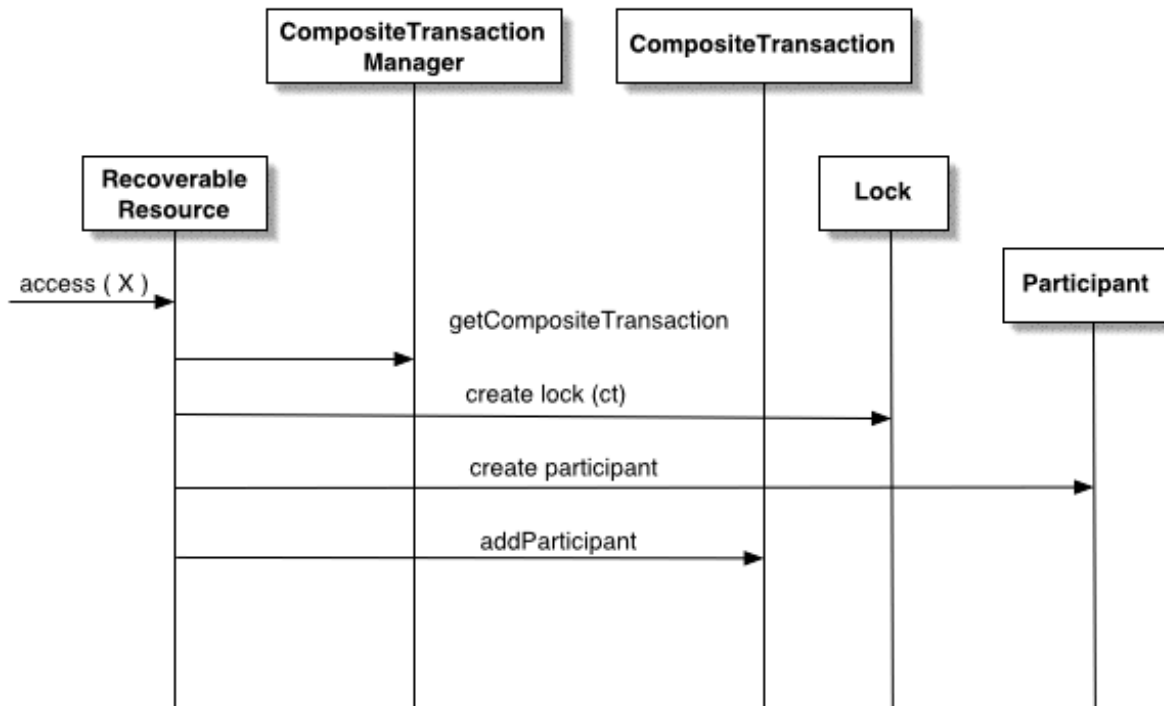
The Participant interface is the only one that has reasonable guarantees about being notified of global two-phase commit events. Implement this interface if this is a requirement for your application.

3.3. Advanced Level

In this section, we discuss how to program your own transaction-aware resources, make their data accesses subject to two-phase commit of the overall transaction, and how to control access among related subtransactions by implementing lock inheritance.

3.3.1. The Big Picture

This subsection first introduces the big picture of the different aspects that come into play when you are implementing one of the features in this section. As it turns out, these tasks are often related in that you will have to implement most of them: programming a transactional data source will also involve controlling access to data (locks and lock inheritance) because locks have a strong impact on the ability to perform rollback (in your implementation of the Participant interface).



The picture above illustrates a typical interaction that takes place when the application accesses some entity X that belongs to your Resource. Access(X) could mean anything from an update of record X to the posting of a message with identifier X. Each step is discussed in some more details next. In this discussion, every object except the CompositeTransactionManager and the CompositeTransaction are assumed to be programmed by you. Also, it is you who is responsible for making sure that the interactions shown are actually performed.

1. Assuming that the resource expects transactional access, it first asks the CompositeTransactionManager if a transaction exists for the calling thread. If none is returned, an exception should probably be thrown by the resource.
2. If a transaction was found, then a lock can be set for the transaction. Locks are typically needed to guarantee correctness of access operations, but also for being able to perform correct rollback.
3. Next, a Participant implementation (specific to your resource implementation) is created and subsequently added to the CompositeTransaction instance.
4. After this, the access operation is performed, and the result is returned to the application. The commit or rollback of the composite transaction (by the application) will eventually trigger the commit or rollback operation of the Participant instance you registered, and this should also lead to the lock being released.

Although the previous picture suggests that your resource be an instance of `com.atomikos.datasource.RecoverableResource`, this is only a requirement if your Participant implementation needs help during recovery. See the section about implementing a RecoverableResource for more information on this.

3.3.2. Implementing Lock Inheritance

The big picture shown before does not tell us what to do if a lock already exists on behalf of *another* CompositeTransaction, and for the desired access of X. If that is the case, then a design decision that rests on you

is the following: should *related* composite transactions (those that are for the same top-level root transaction) be treated in a preferred way or not?

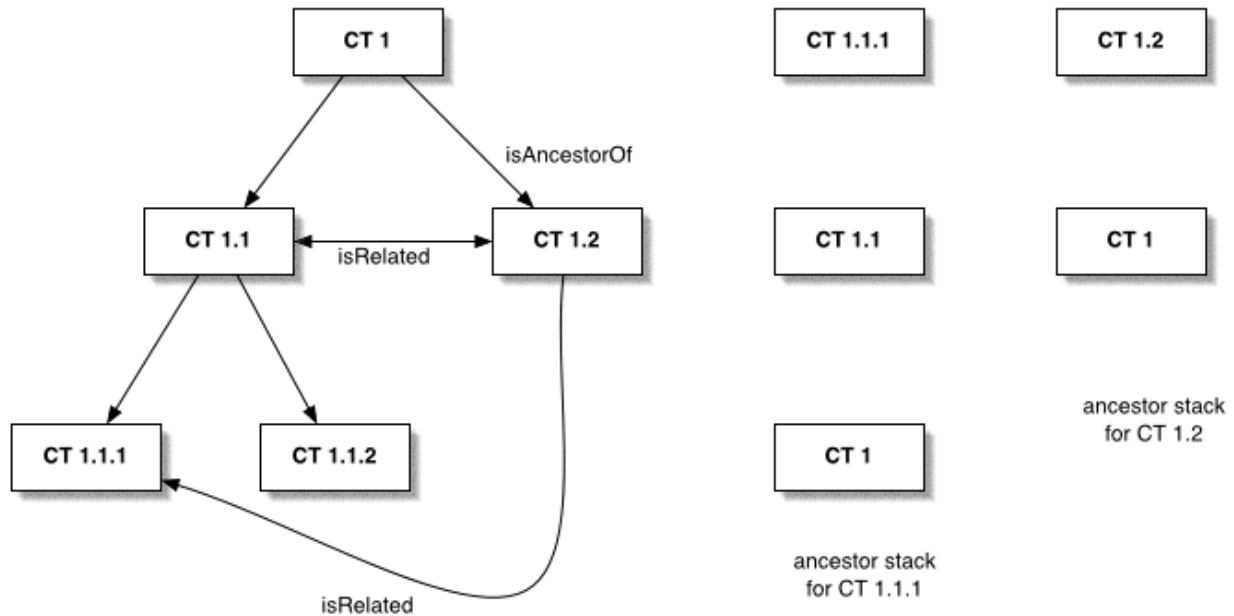
In case a lock already exists, and it is for a *non-related* transaction, then there is only one thing to do according to traditional locking rules: the access has to be put on a waiting list, or denied completely. If, on the other hand, the lock was acquired on behalf of a *related* transaction then a valid option is to provide lock inheritance. This means that the access operation will be allowed eventually, as soon as some conditions are met (which are discussed in more detail below). An alternative option is to treat a related transaction in the same way as non-related transactions, but this can result in deadlocks within the same top-level transaction's execution (can you see why?).

The decision to implement lock inheritance has far-reaching consequences with respect to rollback: it means that rollback can not be done in arbitrary order, but has to respect the *inverse order* in which the different related transactions were granted access. This implies that you should register *only one Participant for all accesses* done by related transactions that benefit from lock inheritance. The reason for this is that the transaction service does not guarantee any particular order on the invocation of rollback for the registered Participant instances! The implication is that for lock-inheritance schemes, you should provide a Participant implementation that is able to rollback the effects of multiple accesses done on behalf of related composite transactions.

The Atomikos APIs provide two ways of implementing lock inheritance. Each one is discussed next.

3.3.2.1. Lock Inheritance Through SubTxAwareParticipant Callbacks

This implementation makes sure that you implement true lock inheritance as it was first introduced in computer science literature. It depends on the capability of subtransactions to notify resources when they terminate. This can trigger the granting of locks for other, related subtransactions that the resource has kept waiting for those locks. In order to explain how this technique works, we will use an example shown in the following picture.



The example involves one root transaction named CT1, and its descendants (subtransactions) which are obviously all related to each other (because they are descendants of the same root and this is the definition of being related).

Now, suppose that a resource is holding a lock on behalf of CT1.1.1 when an access is requested on behalf of the concurrent but related transaction CT1.2. How should the resource make sure that CT1.2 will eventually be able to access the data, but without jeopardizing the integrity of each subtransaction?

The answer is the following: the resource can do so by registering as a SubTxAwareParticipant with CT1.1. Subtransaction commit or rollback of CT1.1 will notify any SubTxAwareParticipants who registered. Those instances then know that *all subtransactions of CT1.1 have finished*. This means that CT1.1's isolation (integrity) can not be violated by CT1.2, and CT1.2 can be granted its lock. Without going into more explanation as to how and why, we can generalize this into the following rule:

A resource that keeps a lock for some subtransaction, say CT1.1.1, and receives a request for a related transaction, say CT1.2, should register as a SubTxAwareParticipant with the *highest ancestor of CT1.1.1 that is not also an ancestor of CT1.2*. This information can be extracted by comparing instances in the lineages (ancestor stacks) available for both CT1.1.1 and CT1.2, as shown in the picture above. The `getLineage()` method of the CompositeTransaction interface will return this ancestor information.

3.3.2.2. Lock Inheritance Through CompositeTransaction.isSerial()

If you don't want the programmatic and communication-related overhead of true lock inheritance, then you can resort to an Atomikos optimization by using the serial flag associated with each CompositeTransaction instance.

The way this method works is simple: it suffices to note that locks are only needed among related transactions if these are *concurrent*. If all related transactions are guaranteed to be executing one after the other, then no malicious effects can arise among related transactions, and a lock for one of them should not block any others. Your resource implementation can easily check this by calling the `CompositeTransaction.isSerial()` test. Atomikos propagates this flag from the root transaction to all subtransactions (even across VMs), and this flag can only be changed at the root level.

Although Atomikos propagates this flag across the system, it is the application's responsibility to respect this flag's setting: before an application starts subtransactions in parallel threads (or exports multiple propagations for the same transaction in parallel) it should assert that the current transaction is not serial.

3.3.3. Implementing a Participant

A Participant is a two-phase commit handle that you supply to the Atomikos core in order to take part in two-phase commit of a CompositeTransaction.

A Participant implementation should properly override the *equals* and *hashCode* methods: two Participants should be equal if and only if they represent the same work's context for two-phase commit. The *hashCode* method should be consistent with this notion of equality.

Implementations of Participant should be either *Serializable* or *Externalizable* in order to be saved in the transaction logs.

In general, a resource creates one Participant instance per CompositeTransaction access. If you implement lock inheritance then you will need one Participant instance per root (top-level) transaction. The reason for this is that related resource accesses that share locks can not be rolled back in arbitrary order. Since

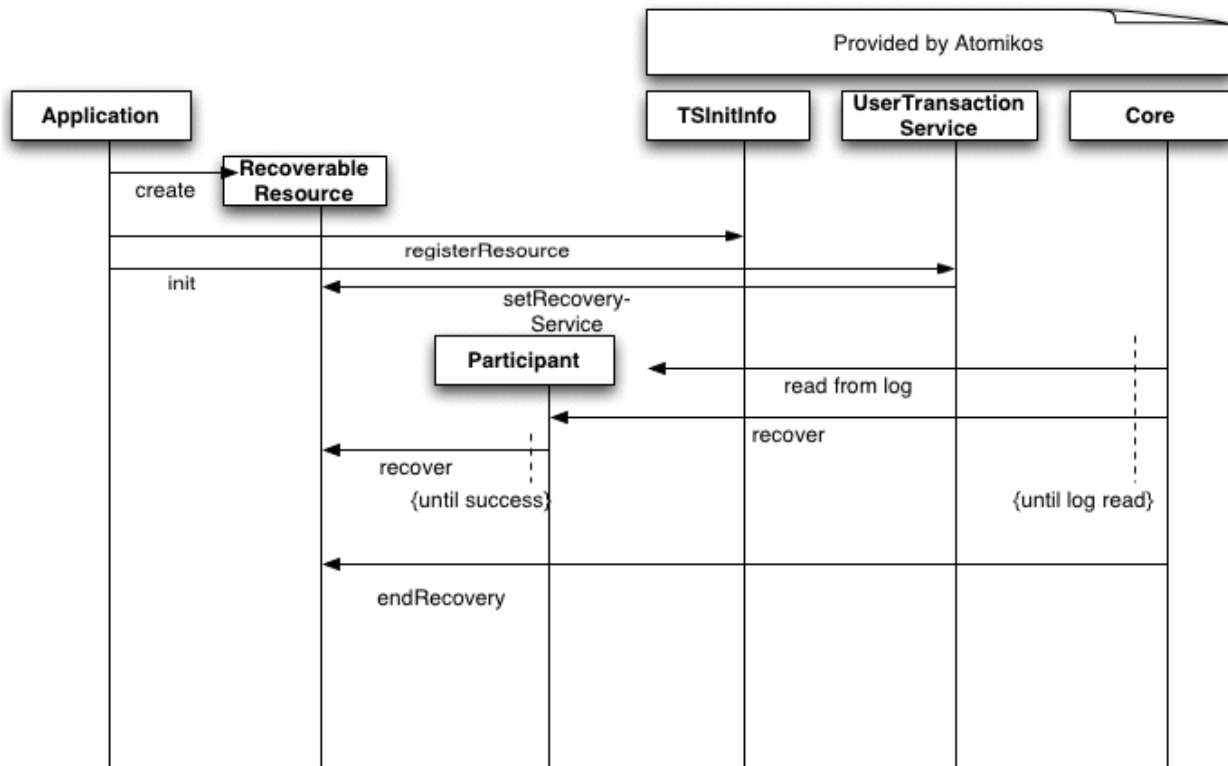
the transaction service offers no guarantee about the relative order in which Participant instances will be called for rollback, related subtransactions that share locks will have to be represented by the same Participant instance. It is that Participant's responsibility to perform the proper rollback logic for each related subtransaction it represents, *in the required order*. This is especially relevant for non-traditional rollback implementations such as compensation.

3.3.4. Implementing a RecoverableResource

The interface `com.atomikos.datasource.RecoverableResource` is a helper class for recovery of Participant instances. In particular, you need to *register* an implementation of this interface if your Participant implementation needs to resolve external references or needs external help during recovery; most notably when the Participant is being read in from the log and any transient internal state has been lost.

Example usage: for recovery of Participant instances for XA transactions, because the `javax.transaction.xa.XAResource` is not serializable. Upon recovery of indoubt XA transactions, an equivalent XAResource is needed from the back-end data source. This is done via interaction with the `com.atomikos.datasource.xa.XATransactionalResource`.

The following picture illustrates the interactions during recovery.



If you implement your own resource and participant then these implementations should respect this contract:

1. Your application creates an instance of the `RecoverableResource` and registers it before initializing the core.
2. Your application calls `init` on the transaction service. This will trigger recovery.
3. The core engine will read any `Participant` instances from the logs and call their `recover` method. Your `Participant` implementations can delegate recovery to their `RecoverableResource` instance(s) if required.

4. After all log contents have been read, the core will call `endRecovery` on all registered resources. This is an indication to those resources that all indoubt Participant instances should have been recovered, meaning that any non-recovered Participants' work can be rolled back (because it was not indoubt according to the transaction manager). This is important in case a crash has happened right after a Participant had voted YES on prepare, but before the transaction service had completed the prepare protocol round for all Participants of that transaction. In that case, the resource will typically keep an indoubt log entry that can be discarded because the transaction never really reached the indoubt state.

The transaction service will initiate recovery when it starts up, and all registered resources will be recovered as part of this process. Any resource that is registered *afterwards* (when the transaction service is already running) should initiate recovery by itself. It can do this by calling the *recover* method of the *RecoveryService* that will be supplied to it during *setRecoveryService*.

Chapter 4. The Administration API

This part discusses the administration API, which allows you to present new administration tools for administering log contents and active transactions. The elements discussed here are all part of the *com.atomikos.icatch.admin* package.

The administration API allows you to implement custom tools for inspecting the transaction logs and forcing termination of problematic transactions. It also allows you to present logged information about transactions in a customized way.

4.1. Administration API Overview

First we will present an overview of the administration API.

4.1.1. LogControl

The LogControl interface is the interface towards the transaction core. If you write a custom administration tool, then the LogControl is where you can get all information from. During initialization, the transaction service will register itself as a LogControl interface with all registered LogAdministrator implementations. A LogAdministrator implementation should save this LogControl handle in an attribute for later usage. The LogControl's only functionality is retrieval of AdminTransactions from the core transaction service.

4.1.2. LogAdministrator

The LogAdministrator is your initialization hook for getting a LogControl handle. You should implement this interface and register it before initialization of the transaction service (by calling `TSInitInfo.registerLogAdministrator`).

- *registerLogControl* This method is called during startup of the transaction service, which will thereby provide the LogAdministrator with a hook to retrieve AdminTransaction instances from.
- *deregisterLogControl* Called during shutdown, to notify the instance that it should no longer use the provided LogControl because the logs will be closed.

4.1.3. AdminTransaction

The AdminTransaction interface is the main administrative tool. It provides you with an administration view of a CompositeCoordinator and allows inspection and control of state information.

The AdminTransaction represents all local CompositeTransaction instances of the same top-level transaction. As such, it is actually an administrative view on a *CompositeCoordinator* rather than on individual transaction instances. This is because the two-phase commit protocol is performed at CompositeCoordinator granularity and not at CompositeTransaction granularity.

- *getTid* Returns the unique transaction identifier of the particular transaction in the logs. This ID corresponds to the *root transaction*, and can be used by administrators to correlate problematic work across different server

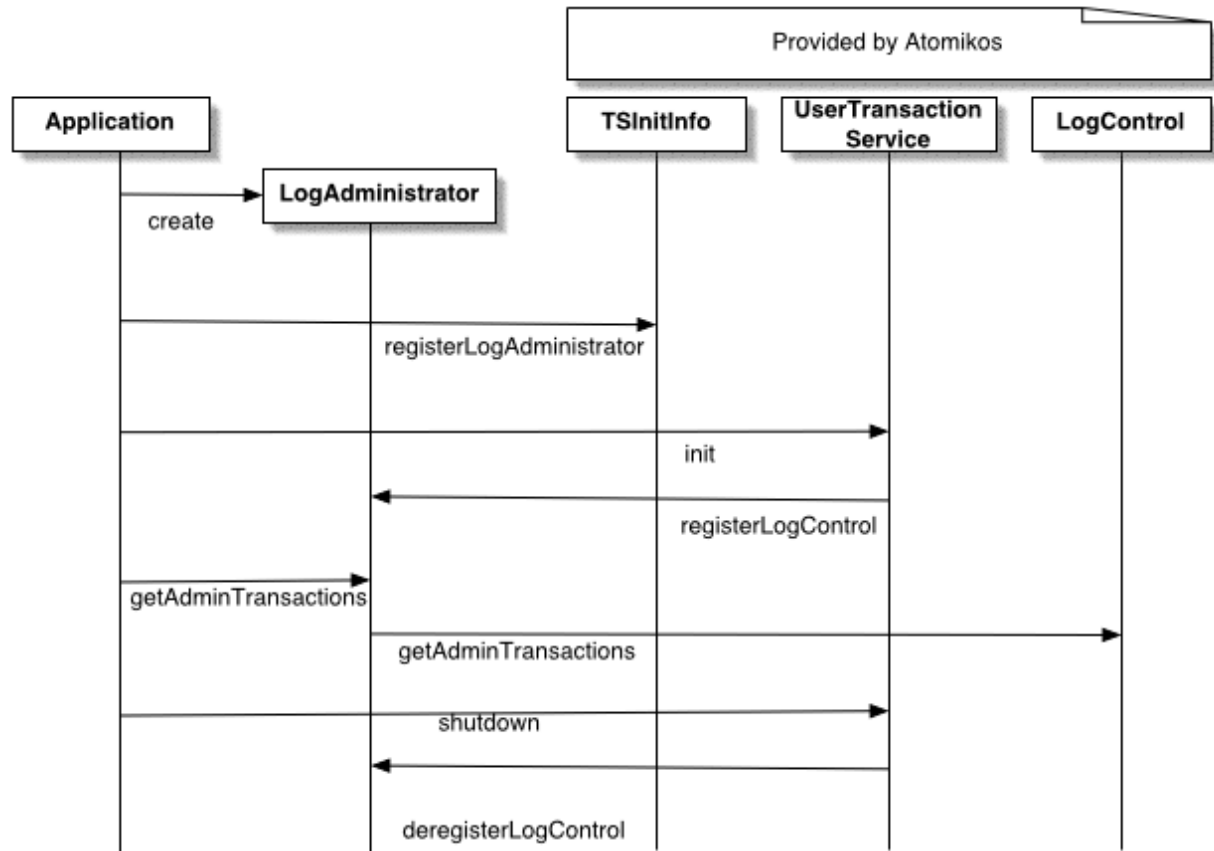
VMs: all log entries of problematic transactions will contain the same root identifier, regardless which node they executed on.

- *getState* Retrieves an integer state code that represents the two-phase commit state at the time of the call.
- *getTags* Retrieves any heuristic messages that were added as a tag to the corresponding CompositeTransaction instances.
- *getHeuristicMessages* Retrieves the union set of all heuristic messages returned by all Participant instances of all committed CompositeTransactions for the underlying CompositeCoordinator.
- *wasCommitted* For heuristic mixed/hazard termination cases, this test allows you to determine which outcome was desired.
- *forceCommit* Long duration indoubts can be forced to commit with this method. This will trigger the commit of all registered Participant instances.
- *forceRollback* Long duration indoubts can be forced to rollback with this method. This will trigger the rollback of all registered Participant instances.
- *forceForget* Heuristic cases can be purged from the logs with this method.

Although this interface extends `Serializable`, the default implementation returned by the `LogControl` provided during startup is *not* meant to be serialized. If you want to implement a remote administration tool then you should be prepared to provide proxy instances that can be shipped to a remote administration client.

4.2. Interactions with the Transaction Service

The following picture summarizes the interactions with the transaction service. If your application needs custom log administration tools, then it will need to offer its own `LogAdministrator` implementation. Note that there is a default implementation offered by class `com.atomikos.icatch.admin.LocalLogAdministrator`. This default implementation will show the `AdminTransaction` information in a Swing screen on the VM that is running the transaction service.



Appendix A. References

- <http://java.sun.com>: Sun's Java website with the JTS specifications and extra information.
- <http://www.atomikos.com>: Atomikos' website; please check regularly for updates and support information.
- Distributed Transaction Processing: The XA Specification (ISBN 1-872630-24-3). Published by The Open Group (<http://www.opengroup.org>).
- G. Pardon. Composite Systems: Decentralized Nested Transactions. Ph.D. thesis Nr. 13993, Swiss Federal Institute of Technology Zurich.
- J. Elliot B. Moss. Nested transactions: an approach to reliable distributed computing. Ph.D. thesis, Massachusetts Institute of Technology, 1981. Available as Technical Report MIT/LCS/TR-260.
- On the Cost of Lock Inheritance in Lock Managers Supporting Nested Transactions (1994). Laurent Daynes, Olivier Gruber, Patrick Valduriez