

# **cosTransactions Application**

**version 1.2**

Typeset in L<sup>A</sup>T<sub>E</sub>X from SGML source using the DOCBUILDER 3.3.2 Document System.

# Contents

<b>1</b>	<b>cosTransactions User's Guide</b>	<b>1</b>
1.1	The cosTransactions Application . . . . .	1
1.1.1	Content Overview . . . . .	1
1.1.2	Brief Description of the User's Guide . . . . .	1
1.2	Introduction to cosTransactions . . . . .	1
1.2.1	Overview . . . . .	1
1.3	Installing cosTransactions . . . . .	2
1.3.1	Installation Process . . . . .	2
1.4	cosTransactions Examples . . . . .	3
1.4.1	A Tutorial on How to Create a Simple Service . . . . .	3
1.5	Resource Skeletons . . . . .	8
1.5.1	Resource Skeletons . . . . .	8
<b>2</b>	<b>cosTransactions Reference Manual</b>	<b>13</b>
2.1	CosTransactions_Control . . . . .	17
2.2	CosTransactions_Coordinator . . . . .	18
2.3	CosTransactions_RecoveryCoordinator . . . . .	22
2.4	CosTransactions_Resource . . . . .	24
2.5	CosTransactions_SubtransactionAwareResource . . . . .	26
2.6	CosTransactions_Terminator . . . . .	27
2.7	CosTransactions_TransactionFactory . . . . .	28
2.8	cosTransactions . . . . .	29
	<b>Glossary</b>	<b>31</b>



# Chapter 1

## cosTransactions User's Guide

The *cosTransactions* application is an Erlang implementation of the OMG CORBA Transaction Service.

### 1.1 The cosTransactions Application

#### 1.1.1 Content Overview

The cosTransactions documentation is divided into three sections:

- PART ONE - The User's Guide  
Description of the cosTransactions Application including services and a small tutorial demonstrating the development of a simple service.
- PART TWO - Release Notes  
A concise history of cosTransactions.
- PART THREE - The Reference Manual  
A quick reference guide, including a brief description, to all the functions available in cosTransactions.

#### 1.1.2 Brief Description of the User's Guide

The User's Guide contains the following parts:

- cosTransactions overview
- cosTransactions installation
- A tutorial example

### 1.2 Introduction to cosTransactions

#### 1.2.1 Overview

The cosTransactions application is a Transaction Service compliant with the OMG<sup>1</sup> Transaction Service CosTransactions 1.1.

---

<sup>1</sup>URL: <http://www.omg.org>

### Purpose and Dependencies

*cosTransactions* is dependent on *Orber version 3.0.1* or later (see the Orber documentation), which provides CORBA functionality in an Erlang environment.

*cosTransactions* is dependent on *supervisor/stdlib-1.7* or later.

Basically, *cosTransaction* implements a *two-phase commit protocol* and allows objects running on different platforms to participate in a transaction.

### Prerequisites

To fully understand the concepts presented in the documentation, it is recommended that the user is familiar with distributed programming, CORBA and the Orber application.

Recommended reading includes *CORBA, Fundamentals and Programming - Jon Siegel* and *Open Telecom Platform Documentation Set*. It is also helpful to have read *Concurrent Programming in Erlang* and, for example, *Transaction Processing: concepts and techniques - Jim Gray, Andreas Reuter*.

#### **Note:**

The *cosTransaction* application is compliant with the OMG CosTransactions specification 1.1. Using other vendors transaction service, compliant with the OMG CosTransactions specification 1.0, may not work since the 'TRANSACTION\_REQUIRED', 'TRANSACTION\_ROLLEDBACK' and 'INVALID\_TRANSACTION' exceptions have been redefined to be system exceptions, i.e., used to be transaction-specific ('CosTransactions\_Exc').

## 1.3 Installing cosTransactions

### 1.3.1 Installation Process

This chapter describes how to install *cosTransactions* [page 29] in an Erlang Environment.

#### Preparation

Before starting the installation process for *cosTransactions*, the application Orber must be running.

The *cosTransactions* application must be able to log progress to disk. The log files are created in the current directory as "oe\_name@machine\_type\_timestamp". Hence, `read` and `write` rights must be granted. If the transaction completes in an orderly fashion the logfiles are removed, but not if an error, which demenads human intervention, occur.

## Configuration

When using the Transaction Service the `cosTransactions` application must be started using either `cosTransactions:start()` or `application:start(cosTransactions)`.

The following application configuration parameters exist:

- `maxRetries` - default is 40 times, i.e., if a transaction participant is unreachable the application will retry to contact it `N` times. Reaching the maximum is considered to be a disaster.
- `comFailWait` - default is 5000 milliseconds, i.e., before the application retries to contact unreachable transaction participants the application wait `Time` milliseconds.

Then the Transaction Factory [page 28] must be started:

- `cosTransactions:start_factory()` - starts and returns a reference to a factory using default configuration parameters.
- `cosTransactions:start_factory(Options)` - starts and returns a reference to a factory using given configuration parameters.

The following options exist:

- `{hash_max, HashValue}` - This value denotes the upper bound of the hash value the Coordinator [page 18] uses. Default is 1013. `HashValue` must be an integer.
- `{allow_subtr, Boolean}` - If set to true it is possible to create subtransactions [page 18]. Default is true.
- `{typecheck, Boolean}` - If set to true all transaction operation's arguments will be type-checked. Default is true.
- `{tty, Boolean}` - Enables or disables error printouts to the `tty`. If `Flag` is false, all text that the error logger would have sent to the terminal is discarded. If `Flag` is true, error messages are sent to the terminal screen.
- `{logfile, FileName}` - This function makes it possible to store all system information in `FileName` (`string()`). It can be used in combination with the `tty(false)` item to have a silent system, where all system information are logged to a file. As default no logfile is used.
- `{maxRetries, Integer}` - default is 40 times, i.e., if a transaction participant is unreachable the application will retry to contact it `N` times. Reaching the maximum is considered to be a disaster. This option overrides the application configuration parameter.
- `{comFailWait, Integer}` - default is 5000 milliseconds, i.e., before the application retries to contact unreachable transaction participants the application wait `Time` milliseconds. This option overrides the application configuration parameter.

The Factory is now ready to use. For a more detailed description see Examples [page 3].

## 1.4 cosTransactions Examples

### 1.4.1 A Tutorial on How to Create a Simple Service

#### Interface design

To use the `cosTransactions` application *participants* must be implemented. There are two types of participants:

- `CosTransactions_Resource` [page 24] - operations used to commit or rollback resources.

- `CosTransactions_SubtransactionAwareResource` [page 26] - operations used when the resources want to be notified when a subtransaction commits. This interface inherits the `CosTransactions_Resource`

The interfaces for these participants are defined in *CosTransactions.idl*

### Generating a Participant Interface

We start by creating an interface which inherits from *CosTransactions::Resource*. Hence, we must also implement all operations defined in the Resource interface. The IDL-file could look like:

```
#ifndef _OWNRESOURCEIMPL_IDL
#define _OWNRESOURCEIMPL_IDL
#include <CosTransactions.idl>

module ownResourceImpl {

    interface ownInterface:CosTransactions::Resource {

        void ownFunctions(in any NeededArguments)
            raises(Systemexceptions,OwnExceptions);

    };
};

#endif
```

Run the IDL compiler on this file by calling the `ic:gen/1` function. This will produce the API named `ownResourceImpl_ownInterface.erl`. After generating the API stubs and the server skeletons it is time to implement the servers and if no special options are sent to the IDL compiler the file name is `ownResourceImpl_ownInterface_impl.erl`.

### Implementation of Participant interface

If the participant is intended to be a plain Resource, we must implement the following operations:

- `prepare/1` - this operation is invoked on the Resource to begin the two-phase commit protocol.
- `rollback/1` - this operation instructs the Resource to rollback all changes made as a part of the transaction.
- `commit/1` - this operation instructs the Resource to commit all changes made as a part of the transaction.
- `commit_one_phase/1` - if possible, the Resource should commit all changes made as part of the transaction. This operation can only be used if the Resource is the only child of its parent.
- `forget/1` - this operation informs the Resource that it is safe to forget any *Heuristic decisions* and the knowledge of the transaction.
- `ownFunctions` - all application specific operations.

If the participant wants to be notified when a subtransaction commits, we must also implement the following operations (besides the operations above):

- `commit_subtransaction/2` - if the `SubtransactionAwareResource` have been registered with a transactions using the operation `CosTransactions.Coordinator:register_subtran_aware/2` it will be notified when the transaction has committed.
- `rollback_subtransaction/1` - if the `SubtransactionAwareResource` have been registered with a transactions using the operation `CosTransactions.Coordinator:register_subtran_aware/2` it will be notified when the transaction has rolled back.

**Note:**

The results of a committed subtransaction are relative to the completion of its ancestor transactions, that is, these results can be undone if any ancestor transaction is rolled back.

### Participant Operations Behavior

Each application participant must behave in a certain way to ensure that the two-phase commit protocol can complete the transactions correctly.

**prepare** This operation ask the participant to vote on the outcome of the transaction. Possible replies are:

- *'VoteReadOnly'* - if no data associated with the transaction has been modified `VoteReadOnly` may be returned. The Resource can forget all knowledge of the transaction and terminate.
- *'VoteCommit'* - if the Resource is able to write all the data needed to commit the transaction to a stable storage, `VoteCommit` may be returned. The Resource will then wait until it is informed of the outcome of the transaction. The Resource may, however, make a unilateral decision (Heuristic) to commit or rollback changes associated with the transaction. When the Resource is informed of the true outcome (rollback/commit) and it is equal to the Heuristic decision the Resource just return 'ok'. But, if there is a mismatch and the commit-operation is irreversible, the Resource must raise a Heuristic Exception [page 24] and wait until the `forget` operation is invoked. The Heuristic Decision must be recorded in stable storage.
- *'VoteRollback'* - the Resource may vote `VoteRollback` under any circumstances. The Resource can forget all knowledge of the transaction and terminate.

**Note:**

Before replying to the `prepare` operation, the Resource must record the `prepare` state, the reference of its superior `RecoveryCoordinator` [page 22] in stable storage. The `RecoveryCoordinator` is obtained when registering as a participant in a transaction.

**rollback** The Resource should, if necessary, rollback all changes made as part of the transaction. If the Resource is not aware of the transaction it should do nothing, e.g., recovered after a failure and have no data in stable storage. Heuristic Decisions must be handled as described above.

**commit** The Resource should, if necessary, commit all changes made as part of the transaction. If the Resource is not aware of the transaction it should do nothing, e.g., recovered after a failure and have no data in stable storage. Heuristic Decisions must be handled as described above.

`commit_one_phase` If possible, the Resource should commit all changes made as part of the transaction. If it cannot, it should raise the `TRANSACTION_ROLLEDBACK` exception. This operation can only be used if the Resource is the only child of its parent. If a failure occurs the completion of the operation must be retried when the failure is repaired. Heuristic Decisions must be handled as described above.

`forget` If the Resource raised a Heuristic Exception to `commit`, `rollback` or `commit_one_phase` this operation will be performed. The Resource can forget all knowledge of the transaction and terminate.

`commit_subtransaction` If the `SubtransactionAwareResource` have been registered with a *subtransaction* using the operation `CosTransactions.Coordinator:register_subtran_aware/2` it will be notified when the transaction has committed. The Resource may raise the exception `'TRANSACTION_ROLLEDBACK'`.

### Note:

The result of a committed subtransaction is relative to the completion of its ancestor transactions, that is, these results can be undone if any ancestor transaction is rolled back.

`rollback_subtransaction` If the `SubtransactionAwareResource` have been registered with a *subtransaction* using the operation `CosTransactions.Coordinator:register_subtran_aware/2` it will be notified when the subtransaction has rolled back.

## How to Run Everything

Below is a short transcript on how to run `cosTransactions`.

```
%% Start Mnesia and Orber
mnesia:delete_schema([node()]),
mnesia:create_schema([node()]),
orber:install([node()]),
application:start(mnesia),
application:start(orber),

%% Register CosTransactions in the IFR.
'oe_CosTransactions': 'oe_register'(),

%% Register the application specific Resource implementations
%% in the IFR.
'oe_ownResourceImpl': 'oe_register'(),

%%-- Set parameters --
%% Timeout can be either 0 (no timeout) or an integer N > 0.
%% The later state that the transaction should be rolled
%% back if the transaction have not completed within N seconds.
TimeOut = 0,

%% Do we want the transaction to report Heuristic Exceptions?
%% This variable must be boolean and indicates the way the
```

---

```

%% Terminator should behave.
Heuristics = true,

%% Start the cosTransactions application.
cosTransactions:start(), %% or application:start(cosTransactions),

%% Start a factory using the default configuration
TrFac = cosTransactions:start_factory(),
%% ... or use configuration parameters.
TrFac = cosTransactions:start_factory([{:typecheck, false}, {:hash_max, 3013}]),

%% Create a new top-level transaction.
Control = 'CosTransactions_TransactionFactory':create(TrFac, TimeOut),

%% Retrieve the Coordinator and Terminator object references from
%% the Control Object.
Term = 'CosTransactions_Control':get_terminator(Control),
Coord = 'CosTransactions_Control':get_coordinator(Control),

%% Create two SubTransactions with the root-Coordinator as parent.
SubCont1 = 'CosTransactions_Coordinator':create_subtransaction(Coord),
SubCont2 = 'CosTransactions_Coordinator':create_subtransaction(Coord),

%% Retrieve the Coordinator references from the Control Objects.
SubCoord1 = 'CosTransactions_Control':get_coordinator(SubCont1),
SubCoord2 = 'CosTransactions_Control':get_coordinator(SubCont2),

%% Create application Resources. We can, for example, start the Resources
%% our selves or look them up in the naming service. This is application
%% specific.
Res1 = ...
Res2 = ...
Res3 = ...
Res4 = ...

%% Register Resources with respective Coordinator. Each call returns
%% a RecoveryCoordinator object reference.
RC1 = 'CosTransactions_Coordinator':register_resource(SubCoord1, Res1),
RC2 = 'CosTransactions_Coordinator':register_resource(SubCoord1, Res2),
RC3 = 'CosTransactions_Coordinator':register_resource(SubCoord2, Res3),
RC4 = 'CosTransactions_Coordinator':register_resource(SubCoord2, Res4),

%% Register Resource 4 with SubCoordinator 1 so that the Resource will be
%% informed when the SubCoordinator commits or roll-back.
'CosTransactions_Coordinator':register_subtran_aware(SubCoord1, Res4),

%% We are now ready to try to commit the transaction. The second argument
%% must be a boolean
Outcome = (catch 'CosTransactions_Terminator':commit(Term, Heuristics)),

```

**Note:**

For the cosTransaction application to be able to recognize if a Resource is dead or in the process of restarting the Resource must be started as persistent, e.g., 'OwnResource':oe\_create\_link(Env, [{regname, {global, RegName}}, {persistent, true}]). For more information see the Orber documentation.

The outcome of the transaction can be:

- ok - the transaction was successfully committed.
- {'EXCEPTION', HeuristicExc} - at least one participant made a Heuristic decision or, due to a failure, one or more participants were unreachable.
- {'EXCEPTION', #'TRANSACTION\_ROLLEDBACK' {}} - the transaction was successfully rolled back.
- Any system exception - the transaction failed with unknown reason.

## 1.5 Resource Skeletons

### 1.5.1 Resource Skeletons

This chapter provides a skeleton for application Resources. For more information see the Orber documentation.

```
%%%-----
%%% File      : Module_Interface_impl.erl
%%% Author   :
%%% Purpose  :
%%% Created  :
%%%-----

-module('Module_Interface_impl').

%%----- INCLUDES -----
-include_lib("orber/include/corba.hrl").
-include_lib("cosTransactions/include/CosTransactions.hrl").

%%----- EXPORTS-----
%%- Inherit from CosTransactions::Resource -----
-export([prepare/2,
        rollback/2,
        commit/2,
        commit_one_phase/2,
        forget/2]).

%%- Inherit from CosTransactions::SubtransactionAwareResource
-export([commit_subtransaction/3,
        rollback_subtransaction/2]).

%%----- gen_server specific -----
-export([init/1, terminate/2, code_change/3, handle_info/2]).
```

```

%%-----
%% function : gen_server specific
%%-----
init(Env) ->
    %% 'trap_exit' optional
    process_flag(trap_exit,true),

    %%--- Possible replies ---
    %% Reply and await next request
    {ok, State}.

    %% Reply and if no more requests within Time the special
    %% timeout message should be handled in the
    %% Module_Interface_impl:handle_info/2 call-back function (use the
    %% IC option {{handle_info, "Module::Interface"}, true}).
    {ok, State, TimeOut}.

    %% Return ignore in order to inform the parent, especially if it is a
    %% supervisor, that the server, as an example, did not start in
    %% accordance with the configuration data.
    ignore.

    %% If the initializing procedure fails, the reason
    %% is supplied as StopReason.
    {stop, StopReason}.

terminate(Reason, State) ->
    ok.

code_change(OldVsn, State, Extra) ->
    {ok, NewState}.

%% If use IC option {{handle_info, "Module::Interface"}, true}
handle_info(Info, State) ->
    %%--- Possible replies ---
    %% Await the next invocation.
    {noreply, State}.
    %% Stop with Reason.
    {stop, Reason, State}.

%%- Inherit from CosTransactions::Resource -----
prepare(State) ->

    %%% Do application specific actions here %%%

    %%-- Reply: --
    %% If no data related to the transaction changed.
    {reply, 'VoteReadOnly', State}
    %% .. or (for example):
    {stop, normal, 'VoteReadOnly', State}.

```

```
%% If able to commit
{reply, 'VoteCommit', State}

%% If not able to commit
{reply, 'VoteRollback', State}
%% .. or (for example):
{stop, normal, 'VoteRollback', State}.
```

rollback(State) ->

```
%%% Do application specific actions here %%%

%%-- Reply: --
%% If able to rollback successfully
{reply, ok, State}
%% .. or (for example):
{stop, normal, ok, State}.

%% If Heuristic Decision. Raise exception:
corba:raise(#'CosTransactions_HeuristicMixed' {})
corba:raise(#'CosTransactions_HeuristicHazard' {})
corba:raise(#'CosTransactions_HeuristicCommit' {})
```

commit(State) ->

```
%%% Do application specific actions here %%%

%%-- Reply: --
%% If able to commit successfully
{reply, ok, State}
%% .. or (for example):
{stop, normal, ok, State}.

%% If the prepare operation never been invoked:
corba:raise(#'CosTransactions_NotPrepared' {})

%% If Heuristic Decision. Raise exception:
corba:raise(#'CosTransactions_HeuristicMixed' {})
corba:raise(#'CosTransactions_HeuristicHazard' {})
corba:raise(#'CosTransactions_HeuristicRollback' {})
```

commit\_one\_phase(State) ->

```
%%% Do application specific actions here %%%

%%-- Reply: --
%% If able to commit successfully
{reply, ok, State}
%% .. or (for example):
{stop, normal, ok, State}.
```





# cosTransactions Reference Manual

## Short Summaries

- Erlang Module **CosTransactions\_Control** [page 17] – This module implements the OMG CosTransactions::Control interface.
- Erlang Module **CosTransactions\_Coordinator** [page 18] – This module implements the OMG CosTransactions::Coordinator interface.
- Erlang Module **CosTransactions\_RecoveryCoordinator** [page 22] – This module implements the OMG CosTransactions::RecoveryCoordinator interface.
- Erlang Module **CosTransactions\_Resource** [page 24] – This module implements the OMG CosTransactions::Resource interface.
- Erlang Module **CosTransactions\_SubtransactionAwareResource** [page 26] – This module implements the OMG CosTransactions::SubtransactionAwareResource interface.
- Erlang Module **CosTransactions\_Terminator** [page 27] – This module implements the OMG CosTransactions::Terminator interface.
- Erlang Module **CosTransactions\_TransactionFactory** [page 28] – This module implements the OMG CosTransactions::TransactionFactory interface.
- Erlang Module **cosTransactions** [page 29] – The main module of the cosTransactions application.

## CosTransactions\_Control

The following functions are exported:

- `get_coordinator(Control)` -> Return [page 17] Return the Coordinator object associated with the target object
- `get_terminator(Control)` -> Return [page 17] Return the Terminator object associated with the target object

## CosTransactions\_Coordinator

The following functions are exported:

- `create_subtransaction(Coordinator) -> Control`  
[page 18] Create a new subtransaction.
- `get_transaction_name(Coordinator) -> Name`  
[page 18] Return the name associated with the object.
- `get_parent_status(Coordinator) -> Status`  
[page 18] Return the status of the parent transaction.
- `get_status(Coordinator) -> Status`  
[page 19] Return the status of the transaction associated with the target object
- `get_top_level_status(Coordinator) -> Status`  
[page 19] Return the status of the top-level transaction associated with the target object
- `hash_top_level_tran(Coordinator) -> Return`  
[page 19] Return a hash code for the top-level transaction associated with the target object
- `hash_transaction(Coordinator) -> Return`  
[page 19] Return a hash code for the transaction associated with the target object.
- `is_descendant_transaction(Coordinator, OtherCoordinator) -> Return`  
[page 19] Return a boolean which indicates whether the transaction associated with the target object is a descendant of the transaction associated with the parameter object
- `is_same_transaction(Coordinator, OtherCoordinator) -> Return`  
[page 19] Return true if the transaction associated with the target object is related to the transaction associated with the parameter object
- `is_top_level_transaction(Coordinator) -> Return`  
[page 20] Return true if the transaction associated with the target object is a top-level transaction
- `register_resource(Coordinator, Resource) -> RecoveryCoordinator`  
[page 20] Register the parameter Resource object as a participant in the transaction associated with the target object
- `register_subtran_aware(Coordinator, SubtransactionAwareResource) -> Return`  
[page 20] Register the parameter SubtransactionAwareResource object such that it will be notified when the transaction, associated with the target object, has committed or rolled back
- `rollback_only(Coordinator) -> Return`  
[page 21] Modify the transaction associated with the target object so the only possible outcome is to rollback the transaction

## CosTransactions\_RecoveryCoordinator

The following functions are exported:

- `replay_completion(RecoveryCoordinator, Timeout, Resource) -> Return`  
[page 22] Return the current status of the transaction

## CosTransactions\_Resource

The following functions are exported:

- `commit(Resource) -> Return`  
[page 24] Instruct the target object to commit the transaction
- `commit_one_phase(Resource) -> Return`  
[page 24] Instruct the target object to commit the transaction
- `forget(Resource) -> Return`  
[page 24] Instruct the target object to forget any heuristic decisions
- `prepare(Resource) -> Return`  
[page 25] Instruct the target object to begin the two-phase commit protocol
- `rollback(Resource) -> Return`  
[page 25] Instruct the target object to rollback the transaction

## CosTransactions\_SubtransactionAwareResource

The following functions are exported:

- `commit_subtransaction(SubtransactionAwareResource, Coordinator) -> Return`  
[page 26] Notify the target object that the transaction has committed
- `rollback_subtransaction(SubtransactionAwareResource) -> Return`  
[page 26] Notify the target object that the transaction has been rolled back

## CosTransactions\_Terminator

The following functions are exported:

- `commit(Terminator, ReportHeuristics) -> Return`  
[page 27] Try to commit a transaction
- `rollback(Terminator) -> Return`  
[page 27] Rollback a transaction

## CosTransactions\_TransactionFactory

The following functions are exported:

- `create(TransactionFactory, Timeout) -> Control`  
[page 28] Create a new top-level transaction

## cosTransactions

The following functions are exported:

- `start()` -> Return  
[page 29] Start the cosTransactions application
- `stop()` -> Return  
[page 29] Stop the cosTransactions application
- `start_factory()` -> TransactionFactory  
[page 29] Start a Transaction Factory
- `start_factory(FacDef)` -> TransactionFactory  
[page 29] Start a Transaction Factory with given options
- `stop_factory(TransactionFactory)` -> Reply  
[page 30] Terminate the target object

# CosTransactions\_Control

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`get_coordinator(Control) -> Return`

Types:

- Control = #objref
- Return = Coordinator | {'EXCEPTION', E}
- Coordinator = #objref
- E = #'CosTransactions.Unavailable' {}

This operation returns the Coordinator object associated with the target object. The Coordinator supports operations for termination of a transaction.

`get_terminator(Control) -> Return`

Types:

- Control = #objref
- Return = Terminator | {'EXCEPTION', E}
- Terminator = #objref
- E = #'CosTransactions.Unavailable' {}

This operation returns the Terminator object associated with the target object. The Terminator supports operations for termination of a transaction.

# CosTransactions\_Coordinator

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`create_subtransaction(Coordinator) -> Control`

Types:

- Coordinator = #objref
- Control = #objref

A new subtransaction is created whose parent is the Coordinator argument.

Raises exception:

- 'SubtransactionsUnavailable' - if nested transactions are not supported.
- 'Inactive' - if target transaction has already been prepared.

`get_transaction_name(Coordinator) -> Name`

Types:

- Coordinator = #objref
- Name = string() of type "oe\_name@machine\_type\_timestamp"

Returns a printable string, which describe the transaction. The main purpose is to support debugging.

`get_parent_status(Coordinator) -> Status`

Types:

- Coordinator = #objref
- Status = atom()

Returns the status of the parent transaction associated with the target object. If the target object is a top-level transaction this operation is equivalent to `get_status/1` operation.

Possible Status replies:

- 'StatusCommitted'
- 'StatusCommitting'
- 'StatusMarkedRollback'

- 'StatusRollingBack'
- 'StatusRolledBack'
- 'StatusActive'
- 'StatusPrepared'
- 'StatusUnknown'
- 'StatusNoTransaction'
- 'StatusPreparing'

`get_status(Coordinator) -> Status`

Types:

- Coordinator = #objref
- Status = atom()

Returns the status of the transaction associated with the target object.

`get_top_level_status(Coordinator) -> Status`

Types:

- Coordinator = #objref
- Status = atom()

Returns the status of the top-level transaction associated with the target object.

`hash_top_level_tran(Coordinator) -> Return`

Types:

- Coordinator = #objref
- Return = integer()

Returns a hash code for the top-level transaction associated with the target object.

Equals the operation `hash_transaction/1` if the target object is a top-level transaction.

`hash_transaction(Coordinator) -> Return`

Types:

- Coordinator = #objref
- Return = integer()

Returns a hash code for the transaction associated with the target object.

`is_descendant_transaction(Coordinator, OtherCoordinator) -> Return`

Types:

- Coordinator = #objref
- OtherCoordinator = #objref
- Return = Boolean

Returns true if the transaction associated with the target object is a descendant of the transaction associated with the parameter object.

`is_same_transaction(Coordinator, OtherCoordinator) -> Return`

Types:

- Coordinator = #objref
- OtherCoordinator = #objref
- Return = Boolean

Returns true if the transaction associated with the target object is related to the transaction associated with the parameter object.

`is_top_level_transaction(Coordinator) -> Return`

Types:

- Coordinator = #objref
- Return = Boolean

Returns true if the transaction associated with the target object is a top-level transaction.

`register_resource(Coordinator, Resource) -> RecoveryCoordinator`

Types:

- Coordinator = #objref
- Resource = #objref
- RecoveryCoordinator = #objref

This operation registers the parameter `Resource` object as a participant in the transaction associated with the target object. The `RecoveryCoordinator` returned by this operation can be used by this `Resource` during recovery.

**Note:**

The Resources will be called in FIFO-order when preparing or committing. Hence, be sure to register the Resources in the correct order.

Raises exception:

- 'Inactive' - if target transaction has already been prepared.

`register_subtran_aware(Coordinator, SubtransactionAwareResource) -> Return`

Types:

- Coordinator = #objref
- Return = ok

This operation registers the parameter `SubtransactionAwareResource` object such that it will be notified when the transaction, associated with the target object, has committed or rolled back.

**Note:**

The Resources will be called in FIFO-order. Hence, be sure to register the Resources in the correct order.

rollback\_only(Coordinator) -> Return

Types:

- Coordinator = #objref
- Return = ok

The transaction associated with the target object is modified so the only possible outcome is to rollback the transaction.

# CosTransactions\_RecoveryCoordinator

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`replay_completion(RecoveryCoordinator, Timeout, Resource) -> Return`

Types:

- `RecoveryCoordinator = #objref`
- `Timeout = integer(), milliseconds | 'infinity'`
- `Resource = #objref`
- `Return = Status | {'EXCEPTION', E}`
- `E = #'CosTransactions_NotPrepared' {}`
- `Status = atom()`

The `RecoveryCoordinator` object is returned by the operation `CosTransactions_Coordinator:register_resource/3`. The `replay_completion/2` may only be used by the registered `Resource` and returns the current status of the transaction. The operation is used when recovering after a failure.

Possible Status replies:

- `'StatusCommitted'`
- `'StatusCommitting'`
- `'StatusMarkedRollback'`
- `'StatusRollingBack'`
- `'StatusRolledBack'`
- `'StatusActive'`
- `'StatusPrepared'`
- `'StatusUnknown'`
- `'StatusNoTransaction'`
- `'StatusPreparing'`

**Warning:**

*replay\_completion/3* is blocking and may cause dead-lock if a child calls this function at the same time as its parent invokes an operation on the child. Dead-lock will not occur if the timeout has any value except 'infinity'.

If the call is external incoming (intra-ORB) the timeout will not be activated. Hence, similar action must be taken if the Resource resides on another vendors ORB.

# CosTransactions\_Resource

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`commit(Resource) -> Return`

Types:

- Resource = #objref
- Return = ok | {'EXCEPTION', E}
- E = #'CosTransactions\_NotPrepared'{} | #'CosTransactions\_HeuristicRollback'{} | #'CosTransactions\_HeuristicMixed'{} | #'CosTransactions\_HeuristicHazard'{}

This operation instructs the Resource to commit all changes made as a part of the transaction.

The Resource can raise:

- Heuristic Exception - if a Heuristic decision is made which differ from the true outcome of the transaction. The Resource must remember the Heuristic outcome until the forget operation is performed.

`commit_one_phase(Resource) -> Return`

Types:

- Resource = #objref
- Return = ok | {'EXCEPTION', E}
- E = #'CosTransactions\_HeuristicHazard'{} | #'CosTransactions\_TransactionRolledBack'{}

If possible, the Resource should commit all changes made as part of the transaction, else it should raise the TRANSACTION\_ROLLEDBACK exception. This operation can only be used if the Resource is the only child of its parent.

`forget(Resource) -> Return`

Types:

- Resource = #objref
- Return = ok

This operation informs the Resource that it is safe to forget any Heuristic decisions and the knowledge of the transaction.

`prepare(Resource) -> Return`

Types:

- Resource = #objref
- Return = Vote | {'EXCEPTION', E}
- Vote = 'VoteReadOnly' | 'VoteCommit' | 'VoteRollback'
- E = #'CosTransactions\_HeuristicMixed'{} | #'CosTransactions\_HeuristicHazard'{} |

This operation is invoked on the Resource to begin the two-phase commit protocol.

The Resource can reply:

- 'VoteReadOnly' - if no persistent data has been modified by the transaction. The Resource can forget all knowledge of the transaction.
- 'VoteCommit' - if the Resource has been prepared and is able to write all the data needed to commit the transaction to stable storage.
- 'VoteRollback' - under any circumstances but must do so if none of the alternatives above are applicable.
- Heuristic Exception - if a Heuristic decision is made which differ from the true outcome of the transaction. The Resource must remember the Heuristic outcome until the `forget` operation is performed.

`rollback(Resource) -> Return`

Types:

- Resource = #objref
- Return = ok | {'EXCEPTION', E}
- E = #'CosTransactions\_HeuristicCommit'{} | #'CosTransactions\_HeuristicMixed'{} | #'CosTransactions\_HeuristicHazard'{} |

This operation instructs the Resource to rollback all changes made as a part of the transaction.

The Resource can raise:

- Heuristic Exception - if a Heuristic decision is made which differ from the true outcome of the transaction. The Resource must remember the Heuristic outcome until the `forget` operation is performed.

# CosTransactions\_SubtransactionAwareResource

Erlang Module

This interface inherits the `CosTransactions::Resource` interface. Hence, it must also support all operations defined in the `Resource` interface.

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`commit_subtransaction(SubtransactionAwareResource, Coordinator) -> Return`

Types:

- `SubtransactionAwareResource = #objref`
- `Coordinator = #objref`
- `Return = ok`

If the `SubtransactionAwareResource` have been registered with a *subtransaction* using the operation `CosTransactions_Coordinator:register_subtran_aware/2`, it will be notified when the transaction has committed.

### Note:

The results of a committed subtransaction are relative to the completion of its ancestor transactions, that is, these results can be undone if any ancestor transaction is rolled back.

`rollback_subtransaction(SubtransactionAwareResource) -> Return`

Types:

- `SubtransactionAwareResource = #objref`
- `Return = ok`

If the `SubtransactionAwareResource` have been registered with a transactions using the operation `CosTransactions_Coordinator:register_subtran_aware/2` it will be notified when the transaction has rolled back.

# CosTransactions\_Terminator

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

`commit(Terminator, ReportHeuristics) -> Return`

Types:

- Terminator = #objref
- ReportHeuristics = boolean()
- Return = ok | {'EXCEPTION', E}
- E = #'CosTransactions\_HeuristicMixed'{} | #'CosTransactions\_HeuristicHazrd'{} | #'CosTransactions\_TransactionRolledBack'{} |

This operation initiates the two-phase commit protocol. If the transaction has not been marked 'rollback only' and all the participants agree to commit, the operation terminates normally. Otherwise, the TransactionRolledBack is raised. If the parameter ReportHeuristics is true and inconsistent outcomes by raising an Heuristic Exception.

`rollback(Terminator) -> Return`

Types:

- Terminator = #objref
- Return = ok

This operation roles back the transaction.

# CosTransactions\_TransactionFactory

---

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

## Exports

```
create(TransactionFactory, Timeout) -> Control
```

Types:

- TransactionFactory = #objref
- Timeout = integer()
- Control = #objref

This operation creates a new top-level transaction.

The Timeout argument can be:

- 0 - no timeout.
- N (integer() > 0) - the transaction will be subject to being rolled back if it does not complete before N seconds have elapsed.

# cosTransactions

Erlang Module

To get access to the record definitions for the structures use:

```
-include_lib("cosTransactions/include/CosTransactions.hrl").
```

This module contains the functions for starting and stopping the application. If the application is started using `application:start(cosTransactions)` the default configuration is used (see listing below). The Factory reference is stored using the CosNaming Service under the id "oe\_cosTransactionsFac\_IPNo".

The following application configuration parameters exist:

- *maxRetries* - default is 40 times, i.e., if a transaction participant is unreachable the application will retry to contact it N times. Reaching the maximum is considered to be a disaster.
- *comFailWait* - default is 5000 milliseconds, i.e., before the application retries to contact unreachable transaction participants the application wait Time milliseconds.

## Exports

`start()` -> Return

Types:

- Return = ok | {error, Reason}

This operation starts the cosTransactions application.

`stop()` -> Return

Types:

- Return = ok | {error, Reason}

This operation stops the cosTransactions application.

`start_factory()` -> TransactionFactory

Types:

- TransactionFactory = #objref

This operation creates a Transaction Factory [page 28]. The Factory is used to create a new top-level transaction [page 17] using default options (see listing below).

`start_factory(FacDef)` -> TransactionFactory

Types:

- FacDef = [Options], see Option listing below.
- TransactionFactory = #objref

This operation creates a Transaction Factory [page 28]. The Factory is used to create a new top-level transaction.

The FacDef list must be a list of {Item, Value} tuples, where the following values are allowed:

- {hash\_max, HashValue} - This value denotes the upper bound of the hash value the Coordinator [page 18] uses. Default is 1013. HashValue must be an integer.
- {allow\_subtr, Boolean} - If set to true it is possible to create subtransactions [page 18]. Default is true.
- {typecheck, Boolean} - If set to true all transaction operation's arguments will be type-checked. Default is true.
- {tty, Boolean} - Enables or disables error printouts to the tty. If Flag is false, all text that the error logger would have sent to the terminal is discarded. If Flag is true, error messages are sent to the terminal screen.
- {logfile, FileName} - This function makes it possible to store all system information in FileName (string()). It can be used in combination with the tty(false) item in to have a silent system, where all system information are logged to a file. As default no logfile is used.
- {maxRetries, Integer} - default is 40 times, i.e., if a transaction participant is unreachable the application will retry to contact it N times. Reaching the maximum is considered to be a disaster. This option overrides the application configuration parameter.
- {comFailWait, Integer} - default is 5000 milliseconds, i.e., before the application retries to contact unreachable transaction participants the application wait Time milliseconds. This option overrides the application configuration parameter.

`stop_factory(TransactionFactory) -> Reply`

Types:

- TransactionFactory = #objref
- Reply = ok | {'EXCEPTION', E}

This operation stop the target transaction factory.

# Glossary

## Heuristic decisions

Heuristic decisions is a unilateral decision by a participant to commit or rollback without receiving the true outcome of the transaction from its parents coordinator.

Local for chapter 1.



# Index of Modules and Functions

Modules are typed in *this way*.

Functions are typed in *this way*.

commit/1  
    *CosTransactions\_Resource* , 24

commit/2  
    *CosTransactions\_Terminator* , 27

commit\_one\_phase/1  
    *CosTransactions\_Resource* , 24

commit\_subtransaction/2  
    *CosTransactions\_SubtransactionAwareResource* ,  
    26

*cosTransactions*  
    start/0, 29  
    start\_factory/0, 29  
    start\_factory/1, 29  
    stop/0, 29  
    stop\_factory/1, 30

*CosTransactions\_RecoveryCoordinator*  
    replay\_completion/3, 22

*CosTransactions\_SubtransactionAwareResource*  
    commit\_subtransaction/2, 26  
    rollback\_subtransaction/1, 26

*CosTransactions\_TransactionFactory*  
    create/2, 28

*CosTransactions\_Control*  
    get\_coordinator/1, 17  
    get\_terminator/1, 17

*CosTransactions\_Coordinator*  
    create\_subtransaction/1, 18  
    get\_parent\_status/1, 18  
    get\_status/1, 19  
    get\_top\_level\_status/1, 19  
    get\_transaction\_name/1, 18  
    hash\_top\_level\_tran/1, 19  
    hash\_transaction/1, 19  
    is\_descendant\_transaction/2, 19  
    is\_same\_transaction/2, 19

is\_top\_level\_transaction/1, 20

register\_resource/2, 20

register\_subtran\_aware/2, 20

rollback\_only/1, 21

*CosTransactions\_Resource*  
    commit/1, 24  
    commit\_one\_phase/1, 24  
    forget/1, 24  
    prepare/1, 25  
    rollback/1, 25

*CosTransactions\_Terminator*  
    commit/2, 27  
    rollback/1, 27

create/2  
    *CosTransactions\_TransactionFactory* , 28

create\_subtransaction/1  
    *CosTransactions\_Coordinator* , 18

forget/1  
    *CosTransactions\_Resource* , 24

get\_coordinator/1  
    *CosTransactions\_Control* , 17

get\_parent\_status/1  
    *CosTransactions\_Coordinator* , 18

get\_status/1  
    *CosTransactions\_Coordinator* , 19

get\_terminator/1  
    *CosTransactions\_Control* , 17

get\_top\_level\_status/1  
    *CosTransactions\_Coordinator* , 19

get\_transaction\_name/1  
    *CosTransactions\_Coordinator* , 18

hash\_top\_level\_tran/1  
    *CosTransactions\_Coordinator* , 19

hash\_transaction/1  
    *CosTransactions\_Coordinator* , 19

is\_descendant\_transaction/2  
    *CosTransactions\_Coordinator* , 19

is\_same\_transaction/2  
    *CosTransactions\_Coordinator* , 19

is\_top\_level\_transaction/1  
    *CosTransactions\_Coordinator* , 20

prepare/1  
    *CosTransactions\_Resource* , 25

register\_resource/2  
    *CosTransactions\_Coordinator* , 20

register\_subtran\_aware/2  
    *CosTransactions\_Coordinator* , 20

replay\_completion/3  
    *CosTransactions\_RecoveryCoordinator* , 22

rollback/1  
    *CosTransactions\_Resource* , 25  
    *CosTransactions\_Terminator* , 27

rollback\_only/1  
    *CosTransactions\_Coordinator* , 21

rollback\_subtransaction/1  
    *CosTransactions\_SubtransactionAwareResource* ,  
    26

start/0  
    *cosTransactions* , 29

start\_factory/0  
    *cosTransactions* , 29

start\_factory/1  
    *cosTransactions* , 29

stop/0  
    *cosTransactions* , 29

stop\_factory/1  
    *cosTransactions* , 30