OTS 1.1 vs. OTS 1.2

Approvers

Function	Name	Approvers comments

Reviewers

Function	Name	Reviewers comments

REFERENCE : 000xxx
CLASSIFICATION: Information
OWNER : Arjuna Lab

CONTENTS Page

1 Introduction

1.1 Scope

CORBA is a standard specification for distributed objects published by the OMG. It applies object-oriented concepts to client/server development and is designed to integrate independently developed applications.

Besides CORBA, the OMG developed a collection of system-level services, necessary to construct a distributed application, packaged with IDL-specified interfaces. Each CORBA service, as well as any specification defined by the OMG, follows a set of procedures or steps (RFI, RFP, RFC, …) until its adoption by OMG members then normally deployed into the market to ensure interoperability between products implementing a same service. An adopted OMG specification or CORBA service may be requested to change if some bugs or malfunctions have been noticed by the implementers (vendors) or by users, or even if some functionalities are missed in the adopted specification but requested to be added. A change to an existing specification can also be requested if it impacted by a new technology or specification: this is the case of the Object Transaction Service or OTS.

OTS is the CORBA service that defines transactions allowing building reliable distributed applications. The OTS specification in its version 1.1, widely implemented in the market, has been requested to change in order to provide some enhancement and to take into account the presence of CORBA Messaging, which defines a new communication paradigm between client and server, or simply objects, impacting the way transactions are managed between these partners. The result of this change is the new specification: OTS 1.2.

This document explains motivations that have led to change to OMG OTS specification, then it describes differences between version 1.1 and 1.2.

Note:

In addition to the way the transactional quality of service is specified, OTS 1.2 has introduced some new methods such *get_timeout()* on the *current* interface, and has also enhanced or clarified the behaviours of some methods and some protocols. This document focuses mainly on the new transactional management since we believe it's the main concept that distinguishes OTS 1.2 from previous OTS versions.

1.2 History

Date	Ver No.	Description	Updated By

1.3 Terminology

Term	Description

1.4 References

References	Description
OTS 1.1	OMG Object Transaction Service v1.1, November 1997

OTS 1.2	OMG OTS 1.2, May 2001, ptc/01-05-02
CORBA Messaging	

2 Motivation

"The introduction of asynchronous messaging (AMI) into CORBA requires a new form of transaction model to be supported. The current CORBA model, the *shared transaction model*, provides an end to end transaction shared by the client and the server. This model cannot be supported by asynchronous messaging. Instead, a new model, which uses a store and forward transport between the client and server, is introduced. In this new model, the communication between client and server is broken into separate requests, separated by a reliable transmission between routers. When transaction are used, this model uses multiple shared transactions, each executed to completion before the next one begins. This transaction model is called the *unshared transaction model*." [In OTS 1.2 specification].

According to this background or motivation, the OTS specification has been impacted mainly on the way to define the transactional behaviour of CORBA objects and the way the transaction context has to be managed according to the transactional behaviour or policy applied on a requested CORBA object.

Impact on the OTS architecture

Basically, from the application point of view entities defined by OTS do not change. These entities are:

- Transactional Client (TC)
- Transactional Objects (TO)
- Recoverable Objects
- Transactional Servers
- Recoverable Servers

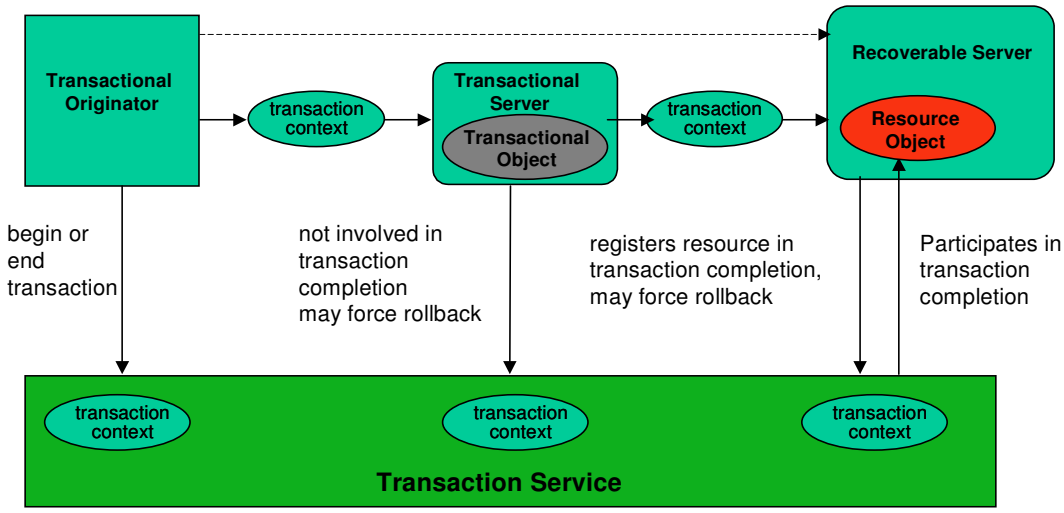

Figure 1 -

The term ***transactional object*** refers to an object whose behaviour is affected by being invoked within the scope of a transaction. Such object typically contains or refers to data that can be modified by requests.

The Transaction Service does not require that all requests have transactional behavior, even when issued within the scope of a transaction. An object can choose to not support transactional behavior, or to support transactional behavior for some requests but not others. Then, we distinguish a transactional object from a ***nontransactional object***, which refers to an object none of whose operations are affected by being invoked within the scope of a transaction created by a transactional client

From the Transaction Service point of view, a transactional object is seen as an object that should obtain information on the transaction for which it should perform task, then an object to which a **transaction context** should be propagated.

OTS 1.1 assumes that the transactional object is normally requested by a transactional client, and then should be involved in the same client's transaction. However, CORBA messaging introduces a new communication paradigm that consists to place a router between the client and the requested object as described in the figure 2.

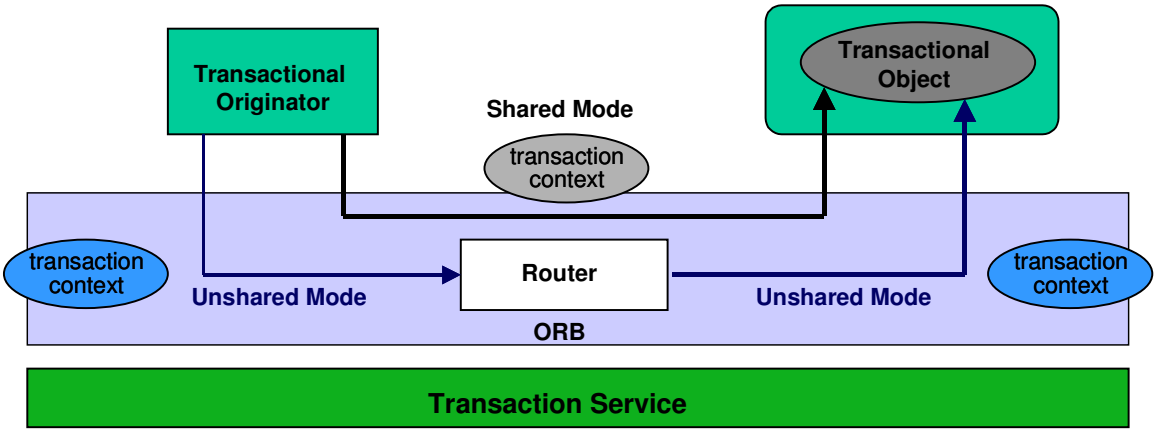

Figure 2 -

The new responsibility of the Transaction Service, as stated by new version 1.2, is not only to determine if a transaction context is needed to be propagated to the requested object, but also to determine if the this context is the one created by the transactional originator (shared context) or a different context (unshared context) created by a tier such the router.

To determine what is the strategy to adapt the Transaction Service or OTS 1.2 utilizes Transactional Policies defined as POA policies. These policies are encoded in the IOR as tag components and exported to the client when an object reference is created. According to these policies the Transaction Service can determine which transaction context is needed to be propagated until the end object. Furthermore, according to a policy, the Transaction Service can determine if the nature of the requested object, or the way that it could be invoked, fits the way the client want to invoke it – synchronously (shared) or asynchronously (unshared).

3 Defined Policies

3.1 Transactional Policy

In OTS 1.1, an object declares its ability to support transaction semantic, in the sense that it able to recognize and to accept a transaction context, by inheriting from an empty interface called **TransactionalObject**. In OTS 1.2 terms, it is as supporting a shared transaction mode.

```
interface TransactionalObject {
};
```

This mechanism had weak transaction semantics, since it was also used by the infrastructure to control transaction propagation. Such an object always received a shared transaction if one was active, but did not receive one when there was no active transaction. This behavior is more accurately described as **allowing** a shared transaction, since it provided no guarantee to the client as to what the object might do if it did or did not receive a shared transaction. This weak semantic is not carried forward as an explicit policy. OTS 1.1 did not provide a mechanism to **require** a shared transaction at invocation time. This behavior produces possible choices for shared transaction support illustrated in Table 1.

Table 1 – Shared Transaction Behaviors in OTS 1.1

Transaction	None	Shared
Requires	No inheritance from TransactionalObject	Cannot be specified with OTS 1.1
Allows	No inheritance from TransactionalObject	inheritance from TransactionalObject

In OTS 1.2, although the use of **TransactionalObject** is maintained for backward compatibility, explicit transactional behaviors are now encoded using **OTSPolicy** values, which are independent of the transaction propagation rules used by the infrastructure. These policies and their OTS 1.1 equivalents are defined as shown in Table 2.

Table 2 – New Shared Transaction Behaviors in OTS 1.2

OTSPolicy	Policy Value	OTS 1.1 Equivalent
Reserved [1]	0	Inheritance from TransactionalObject
REQUIRES	1	No equivalent
FORBIDS	2	No inheritance from TransactionalObject [2]
ADAPTS [3]	3	No equivalent

[1] - The ALLOWS semantics associated with inheritance from **TransactionalObject** cannot be coded as an explicit **OTSPolicy** value in OTS 1.2.

[2] - FORBIDS is more restrictive than the absence of inheritance from **TransactionalObject** since it may raise the INVALID_TRANSACTION exception.

[3] - ADAPTS provides a stronger client-side guarantee than inheritance from **TransactionalObject**.

- REQUIRES - The behavior of the target object depends on the existence of a current transaction. If the invocation does not have a current transaction, a TRANSACTION_REQUIRED exception will be raised.
- FORBIDS - The behavior of the target object depends on the absence of a current transaction. If the invocation does have a current transaction, an INVALID_TRANSACTION exception will be raised.
- ADAPTS - The behavior of the target object will be adjusted to take advantage of a current transaction, if one exists. If not, it will exhibit a different behavior (i.e., the target object is sensitive to the presence or absence of a current transaction).

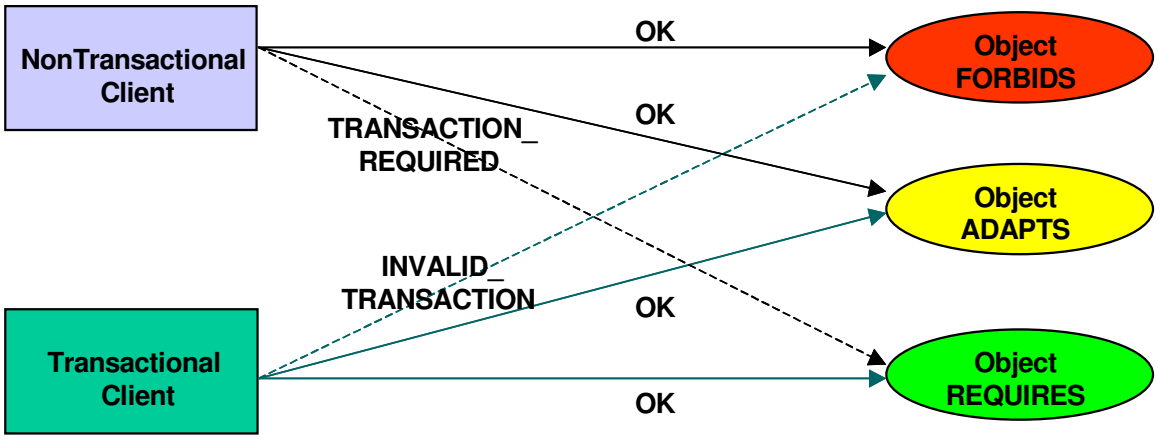

Figure 3 - OTS 1.2 OTSPolicy and client relationship

3.2 Invocation Policy

With the introduction of messaging, the unshared transaction model is used when the request is made via a router. The **InvocationPolicy** specifies which form of invocation the target object supports. The **InvocationPolicy** is defined in Table 3.

Table 3 – InvocationPolicy Behaviors

InvocationPolicy	Policy Value
EITHER	0
SHARED	1
UNSHARED	2

- EITHER - The behavior of the target is not affected by the mode of client invocation. Both direct invocations (synchronous) and invocations using routers (asynchronous) are supported.
- SHARED - all invocations that do not involve a routing element (i.e., the client ORB directly invokes the target object with no intermediate routers).
- UNSHARED - all invocations that involve a routing element.

The **InvocationPolicy** component is significant only when transactions are used with CORBA messaging.

3.3 Interactions between InvocationPolicy and OTSPolicy

Although **InvocationPolicy** and **OTSPolicy** are distinct policies, not all combinations are valid. The valid choices are shown in Table 4.

Table 4 – InvocationPolicy and OTSPolicy combinations

InvocationPolicy/ OTSPolicy	EITHER	SHARED	UNSHARED
REQUIRES	OK Requires_Either	OK Requires_Shared	OK Requires_Unshared
FORBIDS	Invalid	OK Allows_None	Invalid
ADAPTS	Invalid	OK Allows_Shared	Invalid

Transactional target objects that accept invocations via routers must support shared transactions, since the routers use the shared transaction model to reliably forward the request to the next router or the eventual target object. Invalid policy combinations are detected when the POA is created.

3.4 NonTxTargetPolicy Policy

In OTS 1.1, when a client performs a request on a non-transactional object, its request is propagated. This is not the case with OTS 1.2, in which INVALID_TRANSACTION exception is raised.

OTSPolicy is used at the server, which means that the server imposes its transactional policy on the client. From the client side, in particular client in the scope of an active transaction, rather than to perform a transactional request on an object that may raise an exception, it would be useful to avoid propagating the transaction context to such object, which does not understand the transaction semantic. This is the aim of the **NonTxTargetPolicy** Policy.

A non-transactional object has an IOR that either contains a **TAG_OTS_POLICY** component with a value of FORBIDS or does not contain a **TAG_OTS_POLICY** component at all. The **NonTxTargetPolicy** policy is an ORB-policy that is set by the client application using the **ORB::create_policy** interface. Once set, the policy is used to control whether requests on non-transactional targets will raise the INVALID_TRANSACTION exception (PREVENT) or will be permitted to proceed normally (PERMIT). In other words, the request can be rejected, if necessary, at the client side rather at the server side.

4 Policy and Object Reference

Creating Transactional Object References

Object references are created as defined by the POA. An **OTSPolicy** object is created by invoking **ORB::create_policy** with a **PolicyType** of **OTSPolicyType** and a value of type **OTSPolicyValue**. An **InvocationPolicy** may also be associated with a POA using the same mechanism. When either or both of these policies are associated with a POA, the POA will create object references with either or both policies encoded as tagged components in the IOR:

Transactional Policy

OTSPolicy objects can only be used with POAs that support an OTS-aware ORB at the OTS 1.2 level or above. An ORB that recognizes such policies is referred as an **OTS-aware ORB**, while an ORB that does not recognize them is considered as an **OTS-unaware ORB**.

OTSPolicy values are normally encoded in the **TAG_OTS_POLICY** component of the IOR and will always be present when IORs are created by OTS-aware ORBs at the OTS 1.2 level or above.

If an **OTSPolicy** is not present in the IOR, the client may assume two possibilities:

- The object it was created by an OTS-unaware ORB or,
- an OTS-aware ORB at the OTS 1.1 level or below.

Invocation Policy

- **InvocationPolicy** objects can only be used with POAs that support an OTS-aware ORB at the OTS 1.2 level or above.
- **InvocationPolicy** values are encoded in the **TAG_INV_POLICY** component of the IOR.
- If an **InvocationPolicy** is not present in the IOR, it is interpreted as if the **TAG_INV_POLICY** was present with a value of EITHER.

Transaction-unaware POAs

A transaction-unaware POA is any POA created on an OTS-unaware ORB. A transaction-unaware POA will never create a TAG_OTS_POLICY or TAG_INV_POLICY component in any IORs it creates. Transaction-unaware POAs cannot be created on an OTS-aware ORB with an associated OTS 1.2 or higher implementation, however it is possible to create a POA that does not support transactions on an OTS-aware ORB

Transaction-aware POAs

A transaction-aware POA is any POA that is created on an OTS-aware ORB with an associated OTS 1.2 or higher implementation. A transaction-aware POA will include tag components in IORs it creates for **OTSPolicy** values and optionally **InvocationPolicy** values.

- Transaction-aware POAs can only be created in a server, which has an OTS 1.2 or higher implementation associated with its ORB (i.e., an OTS-aware ORB).
- If an application attempts to create a POA with an **OTSPolicy** object in a server that does not have an associated OTS (i.e., an OTS-unaware ORB), the InvalidPolicy exception is raised.
- A POA that does not support transactions is created in an OTS-aware ORB with an **OTSPolicy** object with a FORBIDS policy value and is still called a transaction-aware POA.
- Transaction-aware POAs must have at least an **OTSPolicy** object associated with them. If an **OTSPolicy** is not provided explicitly, an **OTSPolicy** object is created implicitly with a value of FORBIDS.
- Transaction-aware POAs may (but need not) have **InvocationPolicy** objects associated with them.

- An attempt to create a transaction-aware POA with conflicting **OTSPolicy** and **InvocationPolicy** values (as defined in Table 4) will raise the InvalidPolicy exception.

Table 5 summarizes the relationship between POA creation and IOR components on both OTS-unaware and OTS-aware ORBs.

Table 5 - POA creation and IOR components

create_POA	**OTS-Unaware ORB**	**OTS-Aware ORB**		
POA Policies	**Result**	**Result**	**TAG_INV_POLICY**	**TAG_OTS_POLICY**
Neither	Ok	Ok	NO	YES (with FORBDIS)
InvocationPolicy SHARED	Raise InvalidPolicy	Ok	YES	YES (with FORBDIS)
InvocationPolicy EITHER or UNSHARED	Raise InvalidPolicy	Raise InvalidPolicy	-	-
OTSPolicy	Raise InvalidPolicy	Ok	NO	YES
Both with valid combinations	Raise InvalidPolicy	Ok	YES	YES
Both with valid combinations	Raise InvalidPolicy	Raise InvalidPolicy	-	-

OTS 1.1 on top of OTS-Aware ORB

An OTS 1.1 or below may reside on top of an OTS-Aware ORB. In that case any created object reference will not contain the tag TAG_OTS_POLICY.

Question: How an OTS-Aware ORB distinguishes OTS 1.1 from OTS 1.2 to determine if the tag TAG_OTS_POLICY should be added in the IOR with the default policy value FORBIDS?

5 Impact on the User's View

From the user's view the only difference between OTS 1.1 and OTS 1.2 consists on the way to define the transactional behavior of object. Rather than to use the deprecated interface TransactionalObject, it is widely recommended to adopt the transactional policies in order to define transactional behaviors of objects.

6 Impact on Implementer

ORB/TS Considerations

The Transaction Service and the ORB must cooperate to realize certain Transaction Service function. This cooperation is realized on the **client invocation path** and through the **transaction interceptor**. The client invocation path is present even in an OTS-unaware ORB and is required to make certain checks to ensure successful interoperability. The transaction interceptor is a request-level interceptor that is bound into the invocation path.

6.1 Policy Checking Requirements

This section describes the policy checks that are required on the client side before a request is sent to a target object and the server side when a request is received.

The client invocation path is used to describe components of the client-side ORB which may include the ORB itself, the generated client stub, CORBA messaging, and the OTS interceptor. The server side includes the server-side ORB, the POA, and the OTS interceptor.

Client behavior when making transactional invocations

When a client makes a request on a target object, the behavior is influenced by the type of invocation, the existence of an active client transaction, and the **InvocationPolicy** and **OTSPolicy** associated with the target object. The client invocation path must verify that the client invocation mode matches the requirements of the target object. This requires checking the **InvocationPolicy** encoded in the IOR and, in some cases, the **OTSPolicy**. The required behavior is completely described by the following tables.

Table 6 – InvocationPolicy checks required on the client invocation path

Invocation Mode	InvocationPolicy	Required Action
Synchronous	EITHER	Ok – Check OTSPolicy
	SHARED	Ok – Check OTSPolicy
	UNSHARED	Raise TRANSACTION_MODE
Asynchronous	EITHER	Ok – Check OTSPolicy
	SHARED	Raise TRANSACTION_MODE
	UNSHARED	Ok – Check OTSPolicy

An invocation is considered synchronous if it uses a standard client stub, the DII, or AMI with an effective routing policy of **ROUTE_NONE**. An invocation is considered asynchronous if it uses the features of CORBA messaging to invoke on a router rather than the target object.

Table 7 - OTSPolicy checks required on the Client Invocation Path

OTSPolicy	OTS-unaware ORB	OTS-aware ORB
REQUIRES	Raise TRANSACTION_UNAVAILABLE	Call OTS interceptor
FORBIDS	Process invocation	Call OTS interceptor
ADAPTS	Process invocation	Call OTS interceptor

In the case of routed invocations, the client invocation path must substitute an appropriate router IOR before the **OTSPolicy** checks are executed. This ensures that the **OTSPolicy** checks are done against the correct IOR.

The client OTS interceptor is required to make the following policy checks before processing the transaction context described later.

Table 8 - OTSPolicy checking required by client OTS interceptor

OTS Policy	Current Transaction	No Current Transaction
REQUIRES	Process invocation	Raise TRANSACTION_REQUIRED
FORBIDS [1]	PREVENT – raise INVALID_TRANSACTION PERMIT – process invocation	Process invocation
ADAPTS	Process invocation	Process invocation

[1] FORBIDS processing depends on the setting of the **NonTxTargetPolicy** policy.

Server-side behavior when receiving transactional invocations

Since the active transaction state as seen by the server-side can be different than the state observed by the client ORB, the server-side is also required to make the **OTSPolicy** checks. These checks will be made prior to the service context propagation checks.

Table 9 - OTSPolicy checks required on the Server-side

OTSPolicy	OTS-unware ORB	OTS-aware ORB
REQUIRES	Process transaction	Raise TRANSACTION_REQUIRED
FORBIDS	Raise INVALID_TRANSACTION	Process invocation
ADAPTS	Process transaction	Process invocation

The server OTS interceptor is required to make the following policy checks before processing the transaction context.

Table 10 - OTSPolicy checking required by server OTS interceptor

OTSPolicy	Current Transaction	No Current Transaction
REQUIRES	Process transaction	Raise TRANSACTION_REQUIRED
FORBIDS	Raise INVALID_TRANSACTION	Process invocation
ADAPTS	Process transaction	Process invocation

Alternate Client processing for FORBIDS OTSPolicy component

When the NonTxTargetPolicy policy is set to PERMIT, the processing of the FORBIDS value (whether it is explicitly encoded as a TAG_OTS_POLICY component or determined by the absence of inheritance from TransactionalObject) does not raise the INVALID_TRANSACTION exception. Instead it is altered as described below.

Since an OTS must be present for a client to have a current transaction at the time an invocation is made, the client OTS interceptors must also be present within the client environment. This permits an alternative behavior to be implemented on the client-side that maintains compatibility with prior versions of OTS and simplifies client programming when making invocations on non-transactional objects. This alternative behavior is summarized below:

- When the target object supports the FORBIDS policy, the alternative behavior is implemented if the NonTxTargetPolicy policy is set to PERMIT.
- The client-side request interceptor must ensure that the current transaction is inactive before the transaction propagation checks are executed.
- The current transaction must be made active after the request has successfully executed.

The current transaction can be made inactive by performing the equivalent of a **suspend** operation on the current transaction prior to implementing the transaction propagation rules and made active again by performing the equivalent of a **resume** operation when the response is returned to restore the client's current transaction. An implementation that produces equivalent results but does not use the **suspend** and **resume** operation defined by this specification is conformant.

This preserves the client-programming model of earlier OTS levels while still guaranteeing that transactions will not be exported to environments that do not understand transactional semantics.

Interoperation with OTS 1.1 servers and clients

When OTS 1.2 clients are interoperating with OTS 1.1 servers (i.e., the IOR does not contain **TAG_OTS_POLICY** component) the client invocation path must determine if the target object inherits from **TransactionalObjec**t. If it does, it processes the request as if the **OTSPolicy** value was ADAPTS. If it does not, it processes the request as if the **OTSPolicy** value was FORBIDS and uses the **NonTxTargetPolicy** policy to determine the correct behavior.

OTS 1.1 clients may not interoperate with OTS 1.2 servers unless they unconditionally propagate the transaction context. The OTS 1.2 server determines the proper **OTSPolicy** from the **TAG_OTS_POLICY** component in the IOR.

An OTS 1.2 object that also inherits from the deprecated **TransactionalObject** (for backward compatibility) must create POAs with a **OTSPolicy** value of REQUIRES or ADAPTS - any other policy value is illegal and is an implementation error.

6.2 Transaction Service Portability

To enable a single Transaction Service to work with multiple ORBs, it is necessary to define a specific interface between the ORB and the Transaction Service, which conforming ORB implementations will provide, and demanding Transaction Service implementations can rely on.

Identification of the Transaction Service to the ORB

Prior to the first transactional request, the Transaction Service will identify itself to the ORB within its domain to establish the transaction callbacks to be used for transactional requests and replies.

The Transaction Service identifies itself to the ORB using the following interface.

```
interface TSIdentification { // PIDL
exception NotAvailable {};
exception AlreadyIdentified {};
void identify_sender(in CosTSPortability::Sender sender)
raises (NotAvailable, AlreadyIdentified);
void identify_receiver(in CosTSPortability::Receiver receiver)
raises (NotAvailable, AlreadyIdentified);
};
```

The callback routines identified in this operation are always in the same addressing domain as the ORB. On most machine architectures, there is a unique set of callbacks per address space. Since invocation is via a procedure call, independent failures cannot occur.

The Transaction Service Callbacks

Callback routines are actually operations defined on the Sender and Receiver interfaces. Both interfaces, defined as PIDL, are specified in the **CosTSPortability** module.

```
module CosTSPortability { // PIDL
  typedef long ReqId;
  interface Sender {
    void sending_request(in ReqId id,
                         out CosTransactions::PropagationContext ctx);
    void received_reply(in ReqId id,
                        in CosTransactions::PropagationContext ctx,
                        in CORBA::Environment env);
  };
  interface Receiver {
    void received_request(in ReqId id,
```

```
                        in CosTransactions::PropagationContext ctx);
    void sending_reply(in ReqId id,
                        out CosTransactions::PropagationContext ctx);
  };
};
```

The **Sender** interface defines a pair of operations, which are called by the ORB sending the request before it is sent and after its reply is received.

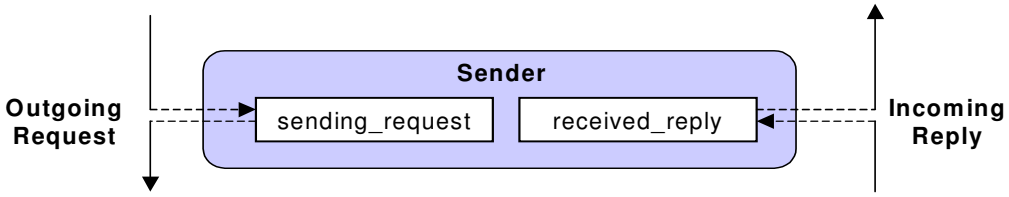

Figure 4 -

The **Receiver** interface defines a pair of operations that are called by the ORB receiving the request when the request is received and before its reply is sent.

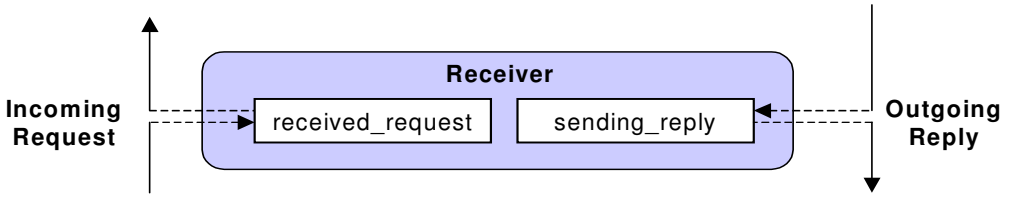

Figure 5 -

Behavior of the Callback Interfaces with OTS 1.2

The following describes the behavior of the ORB and Transaction Service in managing the callback interfaces. The behavior is based on a combination of an active connection between the transaction service and the ORB and the presence or absence of a transaction service context in the GIOP message. The new behavior is summarized below:

Client sending a Request

When the client ORB sends a request, there are three possible transaction service states in the client:

- **OTS_NOT_CONNECTED** - The transaction service has not connected to the client ORB. In this state, the client ORB does not invoke the **Sending_Request** operation and no transaction service context is inserted in the GIOP request message.
- **OTS_NO_CURRENT_TRANSACTION** - The transaction service has connected to the client ORB, but there is no **Current** transaction associated with the client's request. In this state, the client ORB invokes the **Sending_Request** operation and the transaction service returns a null **PropagationContex**t. The client ORB does **not** place a transaction service context in the GIOP request message.
- **OTS_CURRENT_TRANSACTION** - The transaction service is connected to the client ORB and there is a **Current** transaction associated with the client's request. In this state, the client ORB invokes the **Sending_Request** operation and receives a **PropagationContext** from the transaction service. The **PropagationContext** is inserted into the transaction service context of the GIOP request message.

The client ORB cannot distinguish between states 2 and 3 and knows both as OTS (a transaction service is connected to the ORB). This difference is known by the transaction service, which implements the difference in behavior.

Server Receiving a Request

The server ORB receiving a request has two transaction service states:

- **OTS_NOT_CONNECTED** - as defined for the client, and
- **OTS** - a transaction service is connected to the server ORB.

Additionally the server ORB has two states defined by the presence or absence of a transaction service context in the GIOP request message. The server ORB behavior is captured below:

- If no transaction service context is present in the GIOP request message, the server ORB does **not** call the **Receiving_Request** operation and sets NO_REPLY to TRUE. This will be tested when the reply is ready to be sent.
- If a transaction service context is present in the GIOP request message and the transaction service state is OTS_NOT_CONNECTED, the server ORB raises the TRANSACTION_UNAVAILABLE exception back to the client and does not deliver the method request.
- If a transaction service context is present and the transaction service state is OTS, the server ORB invokes **Receiving_Request** passing the transaction service context to the server ORB's transaction service as a **PropagationContex**t.

Server sending a Reply

The server ORB sending a reply is driven by the NO_REPLY state set by receiving this request and the transaction service state. Its behavior is as follows:

- If NO_REPLY is TRUE for this reply (there can be multiple outstanding with deferred synchronous), then the server ORB does not call Sending_Reply and does not insert a service context in the GIOP reply message.
- If NO_REPLY is FALSE and the transaction service state is OTS_NOT_CONNECTED, the server ORB raises the TRANSACTION_ROLLEDBACK exception back to the client. The client is then required to either initiate Rollback or mark the transaction rollback_only. This can only happen if the transaction service abnormally terminates between the time the request is received and the reply is ready to be sent.
- If NO_REPLY is FALSE and the transaction service state is OTS, invoke Sending_Reply and insert the returned PropagationContext in the transaction service context of the GIOP reply message.

Client Receiving a Reply

A client ORB receiving a reply is driven by the presence or absence of a transaction service context in the GIOP reply message and the two transaction service states (OTS and OTS_NOT_CONNECTED). The behavior is outlined below:

- If a transaction service context is not present in the GIOP reply message, the client ORB does **not** call **Receiving_Reply**.
- If a transaction service context is present in the GIOP reply message and the transaction service state is OTS_NOT_CONNECTED, the client ORB raises the TRANSACTION_ROLLEDBACK exception back to the client. Like it's analog in the server, this can only happen if the client transaction service abnormally terminates between the time the request is sent and the reply is received. Since the client's transaction service is no longer active, subsequent operations on any of the OTS interfaces will fail (OBJECT_NOT_EXIST) and the in-flight transaction will rollback when the transaction service is subsequently restarted.
- If a transaction service context is present in the GIOP reply message and the transaction service state is OTS, the client ORB invokes **Receiving_Reply** passing the transaction service context as a **PropagationContext**.

7 Impact on Existing Interfaces

In OTS 1.1, the Synchronization interface inherits the TransactionalObject interface. Since the TransactionalObject interface has been deprecated and replaced by the use of the OTSPolicy component, Synchronization will use the OTSPolicy ADAPTS.

Within the OTS 1.2 specification, the Synchronization interface still inherits the TransactionalObject interface, but this has been maintained for backward compatibility.

```
interface Synchronization : TransactionalObject {
void before_completion();
void after_completion(in Status s);
};
```