Dynamic Support to Transactional Remote Invocations over Multiple Transports*

Francisco Reverbel and Ivan Silva Neto

Department of Computer Science, University of São Paulo
{reverbel,ivanneto}@ime.usp.br

Abstract. XActor is a distributed transaction manager that affords transactional remote invocations over an open-ended set of transports. Its support to transactional interactions is dynamic, in the sense that the transaction manager fully exploits a collection of RMI mechanisms and transport protocols that grows with the addition of plug-in modules to running instances of XActor. A distributed transaction can employ any combination of the transports that the currently installed plug-ins provide. Two-phase commit (logging and failure recovery included) runs over any such combination of transports. Aimed at server-side application containers, XActor can be integrated with those systems in a way that allows its plug-in modules to take advantage of the dynamic deployment facilities of the container environment.

1 Introduction

Ensuring the atomicity of distributed transactions is one of the most traditional and crucial responsibilities of a middleware platform. TP monitors introduced *transactional RPC* as a combination of two paradigms: atomic transactions and remote procedure call (RPC). With the application of object-oriented principles to distributed computing, RPC took the form of the *remote method invocation* (RMI) mechanisms implemented by architectures such as CORBA, DCOM, and Java RMI. Transactional RPC then became *transactional RMI* (TRMI), or simply *transactional remote invocation*.

TRMI is typically implemented by two pieces of software working together in close cooperation: a *transaction manager* (TM) and an *object broker*. The former may be the TM module of a TP monitor or application server. The latter may be a CORBA ORB, may employ Web services technology, or may

* To appear in the *Proceedings of the 2008 ACM Symposium on Applied Computing*.

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

implement proprietary protocols and APIs. While the object broker offers a basic mechanism for remote invocations over some transport (IIOP, SOAP/HTTP, or another protocol stack), the TM ensures the atomicity of distributed transactions by running a two-phase commit protocol at appropriate times and by performing failure recovery tasks after server crashes. The term *object monitor* is sometimes used for the combination of a TM and an object broker [1].

1.1 RMI Support in Java EE

Even though it standardizes a transport (IIOP) for remote invocations, Java EE allows application servers to employ other transports as well. Many servers offer RMI over a proprietary transport as the preferred choice and support IIOP only as a second option, for interoperability in cross-vendor scenarios. Such servers typically offer three possibilities for remote interactions:

- A vendor-specific protocol stack is the default option, which may be more efficient or more convenient in the case of interactions between servers provided by the same vendor.
- IIOP is the transport of choice for interactions that take place between Java EE servers provided by different vendors (or between a Java EE server and a legacy system) and do not require firewall traversals (e.g., remote calls within a LAN).
- SOAP/HTTP is used for interactions through firewalls (e.g., over the Internet).

The diversity of RMI protocols is unavoidable in many Java EE settings. Some contemporary technologies indirectly support protocol heterogeneity by making it easy for a distributed application to perform remote calls over multiple transports:

- Java EE servers frequently use *dynamic proxies as client-side stubs*. Clients receive serialized proxies from other processes and use these proxies to issue remote invocations. Depending on the dynamic proxies it receives, a client may employ multiple protocols to interact with various services, without ever being aware of this fact [2].
- *Dependency injection frameworks* (e.g., Spring [18]) ease the task of creating applications that employ arbitrarily selected middleware components — multiple object brokers, for example — and therefore facilitate the usage of multiple RMI transports by those applications.

1.2 Distributed Transactions[1] in Java EE

Two current standards address transactional remote invocations. The Object Transaction Service (OTS) specification [7] adds transactional capabilities to

[1] We use the term *distributed transaction* for a transaction that spans more than one application server (or, equivalently, involves more than one TM). Such a transaction could be called *fully distributed* (or *inter-TM*) to distinguish it from transactions that are also distributed, but involve a single application server (or TM) and one or more resource managers.

the RMI mechanism provided by CORBA. A pair of WS specifications, WS-Coordination (WS-C) [10] and WS-AtomicTransaction (WS-AT) [9], does the same for the SOAP/HTTP stack. Neither of these standards, however, is a mandatory part of the Java EE platform [14], which does not require product providers to implement any particular protocol for transactional interoperability across application servers[2]. For that reason, transactions spanning more than one application server are problematic in Java EE environments. They are generally unavailable in multi-vendor settings.

Some application servers support distributed transactions only over proprietary protocols, thereby restricting transactional interactions to single-vendor environments. Other servers offer distributed transactions over IIOP, via OTS. Support to distributed transactions over SOAP/HTTP, via WS-C/WS-AT, starts to appear in a few Java EE products. Finally, there are still application servers whose TMs can coordinate transactions that affect the distributed resources (such as databases and persistent message queues) they manage, but cannot perform remote interactions with other TMs. Those servers cannot engage in transactions with other application servers.

1.3 Problem Statement

Transactions across application servers should be supported to the extent allowed by the underlying RMI protocols. No further restrictions should be placed on the servers that may participate in a distributed transaction. If two servers can interact via RMI, then they should also be able to interact via TRMI.

Adding support to distributed transactions over yet another transport should not affect the performance or the scalability of already installed TRMI mechanisms in any significant way. A move to another transaction manager is an example of change that may have that effect. It should not be necessary to replace a TM just for the sake of supporting distributed transactions over a new transport. The addition of new TRMI mechanisms should not even require stopping a TM, let alone replacing it.

1.4 Proposed Solution

Instead of attempting to fight the diversity of RMI protocols, our approach embraces that diversity: we propose a transaction manager that accepts multiple TRMI plug-ins. Each such plug-in is a fairly thin software layer. It encapsulates an RMI mechanism (that is, an object broker) and provides the TM with the means for coordinating remote interactions carried out through that mechanism. In doing so, the plug-in extends the scope of the TM and effectively implements a TRMI mechanism.

Since TRMI plug-ins are neither large nor complex software modules, the task of developing such a plug-in is relatively simple. Support to new TRMI

[2] Transactional interoperability, based on CORBA OTS, is an optional part of Java EE [14].

mechanisms can be added to the TM at run time, by dynamically deploying plug-in components. Together, the TM, its TRMI plug-ins, and the corresponding object brokers act as a "*multi-domain object monitor*" — an object monitor that supports transactions spanning multiple *domains*[3].

This paper discusses the design and implementation of XActor, a distributed transaction manager based upon the ideas we have just outlined. XActor is intended to run within Java EE servers or integrated with dependency injection frameworks such as Spring. We are currently running it within the JBoss Application Server [2], with TRMI plug-ins that allow transactional interactions over three transports: IIOP, SOAP/HTTP, and JBoss Remoting [4]. XActor fully supports distributed transactions (logging and failure recovery included) over any combination of those transports. Moreover, its set of allowed transports is open-ended and may grow at run time. To the best of our knowledge, no other transaction manager offers such flexibility.

1.5 Organization of This Paper

The next section summarizes background material on distributed transactions and on the Java Transaction API; Section 3 introduces XActor, examines the concepts that are central to the design of the transaction manager, and discusses some implementation issues; Section 4 describes the TRMI plug-in architecture of XActor; Section 5 reviews related work; and Section 6 presents our concluding remarks.

2 Background

2.1 Distributed Two-Phase Commit

In the usual presentation of the two-phase commit (2PC) protocol (e.g., in [12]), a transaction coordinator interacts directly with a set of subordinates. At the end of a transaction, the coordinator drives the subordinates through the two phases of the protocol: the voting phase, in which the coordinator collects votes from the subordinates, and the completion phase, in which the coordinator informs the subordinates of the transaction outcome. The coordinator is the TM connected with the user application; the subordinates are the resource managers (RMs) that performed transactional work on behalf of that application. In this model, the *transaction tree* has height one: it is rooted at the coordinator and has the subordinates as leaves.

The 2PC protocol is also applicable to transaction trees with height greater than one [6, 3]. In *distributed 2PC* (or *multi-level 2PC*), the transaction tree has internal nodes, each of which takes a subordinate role with respect to its parent node, and a coordinator role with respect to its child nodes. The topmost node of a multi-level transaction tree is the *root coordinator*, the internal nodes are

[3] A *domain* is a collection of distributed objects implemented with a particular technology; e.g., a CORBA domain, a Web services domain, or a vendor-specific domain.

subcoordinators, and the remaining nodes are *leaf subordinates*. In this model, the root coordinator is the TM connected with the user application, the subcoordinators are TMs co-located with servers that directly or indirectly received transactional remote calls from the application, and the leaf subordinates are the RMs that performed transactional work on behalf of the application[4].

The X/Open Distributed Transaction Processing (DTP) model [16] is a distributed 2PC model. A key part of X/Open DTP is the XA specification [15], which defines the interface between a TM and an RM. XA specifies the prepare-to-commit API exposed by RMs that can take part in distributed transactions. In X/Open DTP and XA, a *transaction branch* represents the transactional work associated with a given global transaction and driven by a given TM. In other words, transaction branches correspond to non-leaf nodes of some transaction tree.

2.2 JTA

The Java Transaction API (JTA) [13] specifies Java interfaces for the local interactions between a TM and the following parties: the RMs, the application container, and the applications deployed into (or hosted by) that container. Our focus here will be in the interface that the TM uses to interact with the RMs and in the TM interfaces used by the application container.

RMs expose transactional resources to the TM as local objects that implement the interface `javax.transaction.xa.XAResource`[5]. This interface is a Java mapping of the XA interface. It has a `prepare` method, which is called by the TM in the voting phase of the 2PC protocol, along with `commit` and `rollback` methods, which are called by the TM in the completion phase of the 2PC protocol.

The TM exposes its functionality to the application container through two interfaces: `javax.transaction.TransactionManager`, implemented by an object that represents the TM itself, and `javax.transaction.Transaction`, implemented by objects that locally represent global transactions. The `Transaction` interface has methods `commit` and `rollback`, which act upon the target transaction, as well as methods `enlistResource` and `delistResource`, which take an `XAResource` parameter and enlist/delist that resource as a participant of the target transaction.

3 XActor

XActor[6] ("transactor") is a TM written in Java and aimed at server-side *application containers* such as Java EE servers and dependency injection frameworks.

[4] Strictly speaking, the leaf subordinates represent *connections* with RMs that performed transactional work on behalf of the application.

[5] Since `XAResource` is a local interface, it is not actually implemented by RMs, but by *resource adapters* co-located with the TM.

[6] Project website: `http://xactor.sourceforge.net/`. All the source code for XActor is available as free software at the project website.

A running XActor instance is typically associated with an application container and exists in the same server process as that container. Each XActor instance can manage two general kinds of transactional resources: XA resources and remote resources.

- XA resources represent resource managers (database systems, message brokers, etc.) accessible to the application container through resource adapters (JDBC drivers, JMS providers, etc.) with support to XA.
- Remote resources represent systems directly or indirectly accessible to the application container through an RMI mechanism. Web services, CORBA servers, and other application container instances are examples of systems that may receive direct or indirect calls from the application container.

An instance of XActor can play the roles of root coordinator and subcoordinator. For a locally started transaction, it acts as the root coordinator. For transactions started in other processes, it acts as a subcoordinator. In either case, the XActor instance is responsible for driving the 2PC protocol[7] within a transaction branch that may involve both XA resources and remote resources accessible through various RMI mechanisms and transport protocols. It interacts with XA resources through the standard interface `javax.transaction.XAResource`, and with remote resources via TRMI plug-ins.

XActor fully supports failure recovery: it performs write-ahead logging of the relevant 2PC events and implements a recovery procedure that runs after server crashes.

3.1 Transaction Branches

Within XActor, transaction branches correspond to `TransactionBranch` objects. The TM creates a `TransactionBranch` whenever a new transaction is locally started, either by the application container or by an application. It also creates `TransactionBranch` objects for *foreign transactions*, i.e., for transactions started at other TMs, which may or may not be XActor instances. The creation of the local `TransactionBranch` of a foreign transaction happens when the transaction reaches the local server for the first time, that is, at the first arrival of a remote request issued within the scope of that transaction. At any given time, the collection of `TransactionBranch` objects within a running XActor instance represents the set of active global transactions known to that TM instance.

Coordinator Propagation. The set of all branches of a given transaction is distributed across the servers reached by that transaction. Each of those branches is a potential node of the transaction tree, which initially has only one node: the root `TransactionBranch`, located in the server that started the transaction.

[7] More precisely, XActor implements distributed 2PC with presumed abort and with commit optimizations for the read-only case and for the one-phase case [6, 3].

That server will be called *root server*. Whenever a server performs a transactional remote invocation, a *coordinator reference* is propagated along with the invocation; that reference identifies a `TransactionBranch`. In transactional invocations issued by the root server, the propagated coordinator reference always points to the root `TransactionBranch`. In transactional invocations issued by other servers, that reference may (at the discretion of the TM in the caller server) either be a copy of the coordinator reference received by the caller server or point to the `TransactionBranch` in the caller server. In the latter case, we say that the TM in the caller server performed *coordinator interposition*.

The Transaction Tree. The mere possession of a coordinator reference by a `TransactionBranch` does not make the branch a node of the transaction tree. In other words, the propagated coordinator reference does not correspond to an edge of the tree. Edges are created in a bottom-up fashion, upon requests from application containers or from child branches. More precisely, the following distributed algorithm establishes parent–child (or coordinator–subordinate) links:

1. Before using an `XAResource` to perform work for a transaction branch, the application container enlists the `XAResource` with the corresponding `TransactionBranch` object. The enlistment establishes a local link (an intra-process link) between the `XAResource`, which assumes the child role, and the `TransactionBranch`, which assumes the parent role.
2. When a non-root `TransactionBranch` gains its first child node, the `TransactionBranch` enlists itself with its propagated coordinator. The enlistment establishes an inter-process link between the `TransactionBranch`, which assumes the child role, and its propagated coordinator, which *at this point* assumes the parent role.

The recursive execution of step 2 above produces a rooted tree, whose leaf nodes are `XAResource`s, and whose root node is the root `TransactionBranch`.

3.2 Transaction Identifiers

XActor associates three identifiers with each `TransactionBranch`: a local id, a global id and an `Xid`.

- *Local ids* are 64-bit integers that uniquely identify transaction branches within a TM instance. XActor locally generates these identifiers when it instantiates `TransactionBranch` objects. It does so even in the case of branches of foreign transactions.
- *Global ids* are structures that uniquely identify a (global) transaction. When a transaction starts, the TM of the root server generates a global id and assigns it to the root transaction branch. The global id is then propagated along with every remote invocation issued within the scope of the transaction. If the arrival of such an invocation triggers the creation of a new

`TransactionBranch` object within some server process, then the incoming global id is assigned to the newly created branch. All branches of a given transaction are therefore associated with the same global id. In interoperability scenarios (e.g, IIOP or SOAP/HTTP), however, some of these branches may not be actually represented by XActor-specific `TransactionBranch` objects, as the TMs involved in the transaction are not necessarily instances of XActor.

- `Xid`s are standardized structures that uniquely identify a transaction branch. The JTA specification defines the `javax.transaction.xa.Xid` interface, which is a Java mapping of the `XID` structure used by X/Open DTP. An `Xid` comprises the global id of a transaction, plus an additional field (the so-called *branch qualifier*) that identifies a particular branch of that transaction.

Local Id Usage. Local ids are a concise way of specifying a transaction branch when an instance of XActor has already been specified. They are used within remote references to transaction branches; e.g., in the "object id" part of an IOR [8] whose host and port fields already identify an instance of XActor, or in the "reference properties" element of an endpoint reference built in conformity with the WS-Addressing specification [17]. When such a reference is used to perform a remote invocation, its local id is sent out as a field of the invocation request. Local ids also appear within global ids and within `Xid`s generated by XActor.

3.3 The Transaction Branch Interface

In XActor, transaction branches implement an interface that extends the standard JTA interface `javax.transaction.Transaction`. JTA targets the more restricted scenario in which a single TM coordinates locally started transactions that may span multiple XA resources, but do not involve other TMs or application containers. In other words, JTA supports only transactions with a single branch. Accordingly, its `Transaction` interface has methods that allow the local TM to commit or rollback the root transaction branch (the only branch of a plain JTA transaction) and methods that allow the application container to enlist/delist `XAResource` objects with the transaction. In XActor, however, a transaction branch needs additional methods, because (*i*) it does not necessarily play the role of root coordinator, and (*ii*) it may have subordinates that are not XA resources.

Figure 1 shows the `TransactionBranch` interface[8]. The three groups of methods listed under the comments "Coordinator facet", "Resource facet", and "Recovery coordinator facet" correspond to different roles played by a `TransactionBranch` as a (potential) node of some transaction tree.

[8] For clarity, we have omitted any `throws` clauses from every method declaration presented in this paper.

```
interface TransactionBranch
        extends javax.transaction.Transaction {

    // Coordinator facet (for Resource enlistment):
    void enlistRemoteResource(Resource r,
                              short trmiMechId);
    ...

    // Resource facet:
    int prepare()
    void rollbackBranch()
    void commitBranch(boolean onePhase)
    void forget();

    // Recovery coordinator facet:
    int replayCompletion(Resource r);

    // Other methods:
    Object getPropagationContext(short trmiMechId);
    ...
}
```

Fig. 1. The `TransactionBranch` interface.

The "Coordinator facet" is used by the distributed algorithm that builds the transaction tree. It receives calls from remote `TransactionBranch` instances that do not have a parent branch yet. The single method shown in that facet, `enlistRemoteResource`, establishes a link between the target branch, which takes the parent (coordinator) role, and the caller branch, which takes the child (subordinate) role. When a transaction branch calls `enlistRemoteResource`, it passes a reference to itself as the first parameter of the call. Note, however, that the type of that parameter is not `TransactionBranch`, but `Resource`. This parameter type means that the passed reference does not allow remote invocations to each and every method of the referenced branch, but only to the methods in the "Resource facet" of that branch. In other words, the newly added child node exposes a "Resource view" of itself to its parent node. The second parameter to `enlistRemoteResource` identifies the transactional RMI mechanism used in the interactions between the server that contains the parent branch and the one that contains the child branch.

The "Resource facet" in Figure 1 is used at commit time. It allows a subordinate branch to receive 2PC requests from its remote coordinator, which may be either the root coordinator or a subcoordinator. The subordinate receives a `prepare` call in the voting phase, and either a `rollbackBranch` or a `commitBranch` call in the completion phase of the 2PC protocol. Alternatively, if the coordinator uses the one-phase commit optimization [6, 3], the subordinate receives a single `commitBranch` call with the `onePhase` parameter set to true. Fi-

nally, the `forget` method tells the subordinate branch to discard all information on previously raised heuristic exceptions.

The "Recovery coordinator facet" in Figure 1 is used only during failure recovery. It allows a (sub)coordinator branch to receive requests from subordinate branches that were not informed of the transaction outcome. A call to `replayCompletion` asks the (sub)coordinator to reissue the completion phase invocation (either to `rollbackBranch` or to `commitBranch`) on the subordinate branch specified by the `Resource` parameter.

In addition to the methods in the three facets just described, the `TransactionBranch` interface offers a number of other methods. Figure 1 shows one such method, `getPropagationContext`, which returns a transactional context to be propagated along with the remote invocations performed within the scope of the transaction branch.

3.4 Transactional Contexts

We have already seen two items that must be propagated with transactional invocations: the global id and the coordinator reference. In XActor, the latter is actually a reference to the "Coordinator facet" of some `TransactionBranch`. The *transactional context* is a data structure that holds those two items. It has a third mandatory field, which carries the value of the transaction timeout.

The transactional context is specific for a TRMI mechanism, because it includes a coordinator reference that affords remote invocations over a particular transport. In Figure 1, the first parameter to `getPropagationContext` identifies the TRMI mechanism for which that method will return a transactional context. Our implementation of that method does not actually create transactional contexts; it delegates this task to TRMI plug-ins.

Inbound and Outbound Coordinator References. Every foreign transaction branch has an inbound coordinator reference and an outbound coordinator reference. The inbound reference was in the coordinator field of the foreign context whose arrival triggered the instantiation of the `TransactionBranch` instance. The outbound reference is in the coordinator field of the context returned by a call to `getPropagationContext` on the given transaction branch. The outbound reference may either identify the "Coordinator facet" of that branch (coordinator interposition) or it may be a copy of the inbound coordinator reference (no interposition).

Interposition. The default behavior of the method `getPropagationContext` is to return a context with an interposed coordinator if and only if the TRMI mechanism identified by the `trmiMech` parameter is different from the one supported by the inbound coordinator. In other words, that method performs coordinator interposition when the server that contains the inbound coordinator and the one(s) that will receive the outbound context may not be able to interact directly with each other, as they are not known to support a common TRMI mechanism. This default behavior can be changed through a `TransactionBranch` attribute

(not shown in Figure 1) that tells `getPropagationContext` to always perform interposition.

4 TRMI Plug-ins

A TRMI plug-in encapsulates an RMI mechanism and completely isolates the TM from that mechanism. The plug-in cooperates closely with an instance of XActor and has the following responsibilities: (*i*) it exposes the non-leaf nodes of transaction trees as distributed objects accessible through the RMI mechanism; (*ii*) it provides XActor with stub wrappers that act as local proxies for the remote parties involved in the transaction; (*iii*) it creates transactional contexts upon requests from XActor; (*iv*) it includes transactional contexts into outgoing invocations; (*v*) it extracts transactional contexts from incoming invocations and passes the information in those contexts on to XActor; (*vi*) it supports externalization of remote references (conversions between remote references and strings) through an interface that is independent of the RMI mechanism.

4.1 Remote Interactions between Transaction Branches

A TRMI plug-in exposes the facets of a transaction branch as remote objects accessible through the underlying RMI mechanism. The functionality in each facet can be conveyed by a Java interface. Figure 2 shows the interfaces associated with "Coordinator facets", "Resource facets", and "Recovery coordinator

```
interface Coordinator extends java.rmi.Remote {
    RecoveryCoordinator registerResource(Resource r);
    ...
}
```

```
interface Resource extends java.rmi.Remote {
    Vote prepare();
    void rollback();
    void commit();
    void commitOnePhase();
    forget();
}
```

```
interface RecoveryCoordinator
        extends java.rmi.Remote {
    Status replayCompletion(Resource r);
}
```

Fig. 2. The `Coordinator`, `Resource`, and `RecoveryCoordinator` interfaces.

facets". Those Java interfaces were modeled after their IDL counterparts in OTS [7].

Note, however, that the Java interfaces `Coordinator`, `Resource`, and `RecoveryCoordinator` are not necessarily the ones through which a TRMI plug-in allows remote access to transaction branch facets. The interfaces actually exposed depend on the underlying RMI mechanism, and may not be Java interfaces at all. For example, the plug-in that supports transactional invocations over IIOP exposes IDL interfaces specified by OTS. The one that supports transactional invocations over SOAP/HTTP exposes WSDL interfaces specified by WS-C and WS-AT. Nevertheless, the remote interfaces actually exposed by a TRMI plug-in should be very similar to the ones in Figure 2.

Across a collection of transaction branches, a facet appears as a set of remotely accessible objects, e.g., the set of all `Coordinator` objects within a given instance of XActor. Each such object has its own identity, which is expressed in a way that depends on the underlying RMI mechanism: through a CORBA object reference, a Web service endpoint, or some similar artifact. Even so, a TRMI plug-in does not need to instantiate a Java object per facet. The typical TRMI plug-in has a single "servant object" that handles incoming invocations to all instances of a facet interface. For each such invocation, the servant object performs the following steps: (*i*) it extracts from the invocation the local id of a transaction branch, (*ii*) it queries the local TM (which has a hash map from local ids to transaction branches) to obtain the `TransactionBranch` with that local id, and (*iii*) it forwards the invocation to the `TransactionBranch` object. In step (*iii*), the servant object uses the natural mapping between the remotely accessible methods in facet interfaces (Figure 2) and the facet methods of the `TransactionBranch` interface (Figure 1).

Stub Wrappers. Exposing transaction branch facets to other processes through some RMI mechanism is an important and necessary measure. But that measure, alone, does not solve the problem of enabling remote interactions between transaction branches without making XActor dependent upon particular RMI mechanisms. What is still needed is a way of accessing remote transaction branches that is uniform across all RMI mechanisms. TRMI plug-ins fulfill this requirement by implementing stub wrappers.

A *stub wrapper* is a proxy object that (*i*) implements one of the Java interfaces in Figure 2 (`Coordinator`, `Resource`, or `RecoveryCoordinator`), (*ii*) has a remote reference (i.e., a stub), specific to a given RMI mechanism, that identifies a transaction branch facet whose type (coordinator, resource or recovery coordinator) matches the one of the stub wrapper, and (*iii*) forwards all invocations to the remote facet, through the RMI mechanism. For example, a stub wrapper for an OTS resource implements the `Resource` interface in Figure 2, has a CORBA reference (an IIOP stub) for a remote OTS resource, and converts local calls to `Resource` methods into CORBA requests on the remote OTS resource.

Within XActor, all references to facets of remote transaction branches take the form of stub wrappers. Each such reference has a "native form" (a Java

reference to a stub that is specific to the underlying RMI mechanism) and a "wrapped form" (a Java reference to a stub wrapper), which is the only one seen by XActor. A TRMI plug-in has the responsibilities of instantiating stub wrappers and transparently converting remote references between the wrapped and the native forms.

Figure 3 shows a `TransactionBranch` as an internal node of a distributed transaction tree. That transaction branch lies within an instance of XActor with plug-ins for IIOP, SOAP/HTTP, and JBoss Remoting. It has five child nodes: two XA resources and three remote resources. The transaction branch interacts with its parent coordinator via SOAP/ HTTP and uses a distinct transport to communicate with each of its three remote subordinates. Figure 3 also shows the external facets of the transaction branch. Note that the branch exposes its `Resource` facet via SOAP/HTTP, which is the transport it uses to interact with its parent coordinator. Its `Coordinator` and `RecoveryCoordinator` facets, however, receive calls from three remote subordinates, each of which employs a different RMI mechanism. Those facets must therefore be exposed through all three RMI mechanisms. In Figure 3, each of the "facet symbols" for the `Coordinator` and `RecoveryCoordinator` facets actually represents a triple of remotely accessible facets: one accessible via IIOP, another via SOAP/HTTP, and the last via JBoss Remoting.

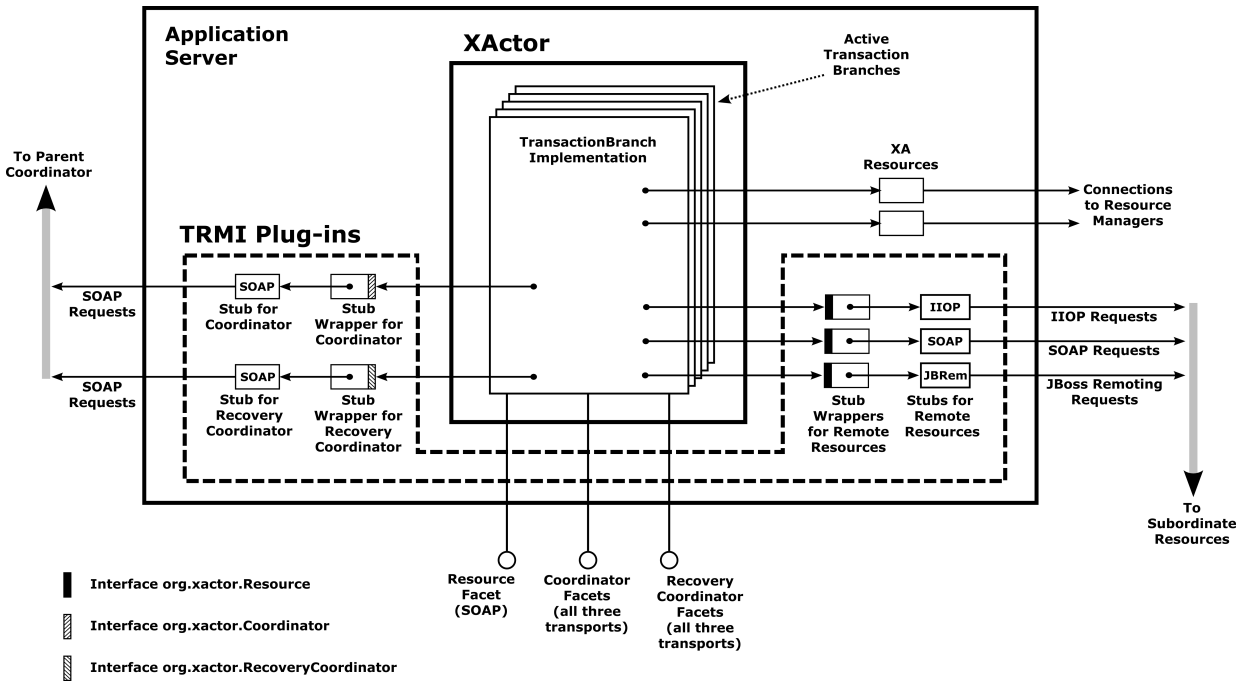

Fig. 3. A `TransactionBranch` instance within XActor.

4.2 Dynamic Deployment of TRMI Plug-ins

XActor maintains a registry of TRMI plug-ins. New plug-ins can be added to a running TM through the interface `TRMIMechanismRegistry`, which the transaction manager implements. If XActor is used in an application server that supports *hot deployment* of middleware components, as is the case with the JBoss and Apache Geronimo servers, then TRMI plug-ins can be packaged as middleware components and deployed into a running server simply by dropping deployment units into a well-known directory. This is what we are currently doing on the JBoss application server.

In our JBoss implementation, TRMI plug-ins are deployable MBeans [2]. Such a component has a `create` method, which is called when the MBean is deployed. Within that method, the TRMI plug-in performs all of its initialization steps and then uses the `TRMIMechanismRegistry` to register itself with XActor.

Figure 4 shows the interface `TRMIMechanismRegistry`. Each TRMI plug-in is assigned an identifier, a small integer passed as the first parameter to the method `addTRMIMechanism`. Moreover, each TRMI plug-in exposes itself to XActor as a triple of objects: a `ContextFactory`, a `ResourceFactory`, and a `StringRemoteRefConverter`.

```
interface TRMIMechanismRegistry {
    void addTRMIMechanism(short trmiMechId,
                          ContextFactory contextFactory,
                          ResourceFactory resourceFactory,
                          StringRemoteRefConverter strRefConverter);
    void removeTRMIMechanism(short trmiMechId);
    ContextFactory getContextFactory(short trmiMechId);
    ResourceFactory getResourceFactory(short trmiMechId);
    StringRemoteRefConverter getStringRemoteRefConverter(
                                                    short trmiMechId);
}
```

Fig. 4. The `TRMIMechanismRegistry` interface.

4.3 Resource Factories

Every TRMI plug-in offers XActor an implementation of the `ResourceFactory` interface (Figure 5). `TransactionBranch`es use this interface to obtain references to "Resource facets" of themselves. Before enlisting itself as a child node with its inbound coordinator, a `TransactionBranch` uses the `ResourceFactory` associated with the TRMI mechanism through which it received the coordinator reference. The `TransactionBranch` passes its own local id to the method `createResource` and receives back a wrapped reference for the resource that it should enlist with the inbound coordinator.

```
interface ResourceFactory {
    Resource createResource(long localId);
}
```

Fig. 5. The `ResourceFactory` interface.

4.4 Creation and Propagation of Transactional Contexts

Since transactional contexts are specific to TRMI mechanisms, each plug-in must provide XActor with an implementation of the `ContextFactory` interface, which appears in Figure 6. The two methods named `createContext` serve different purposes: while the first is typically used to create contexts with foreign coordinators, the second creates a context with a local coordinator, which may be either the root coordinator of a transaction or an interposed coordinator.

```
interface ContextFactory {
    Object createContext(GlobalId globalId,
                         Coordinator c);
    Object createContext(GlobalId globalId,
                         long localId);
    void setTimeout(Object context, int timeout);
}
```

Fig. 6. The `ContextFactory` interface.

The creation of transactional contexts happens upon requests from `TransactionBranch` instances, which keep the transactional contexts they created for each TRMI mechanism. Moreover, the TM maintains the association between application threads and `TransactionBranch` instances. As a consequence, application threads are associated with a (possibly null) transactional context for each TRMI mechanism. Plug-ins use this fact to handle the outbound propagation of transactional contexts. Whenever an application thread issues a remote call through some TRMI mechanism, the corresponding plug-in obtains the transactional context for that mechanism and inserts the context into the outgoing request. At the callee side, a matching plug-in extracts the inbound context from incoming requests.

RMI mechanisms typically offer an interceptor facility, such as CORBA portable interceptors or SOAP interceptors. A TRMI plug-in can rely on such a facility in order to insert transactional contexts into outgoing requests and to extract contexts from incoming requests. If the underlying RMI mechanism lacks that facility, the TRMI plug-in can still perform request interception by using aspect-oriented tools.

4.5 Externalization of References

To support failure recovery, each TM involved in a distributed transaction maintains a transactional log in stable storage. At key points of the execution of the 2PC protocol, the coordinator and the subcoordinators write specific records to their transactional logs [6, 3]. Such log records contain information that will be used by the failure recovery procedure, in the event of a server crash. A TM that has a subordinate role in some transaction tree writes a log record containing a reference to its parent coordinator. A TM that has a coordinative role in some transaction tree (either as the root coordinator or as a subcoordinator) writes a log record containing references to its remote subordinates, i.e., to all the remote resources enlisted in its transaction branch.

Since the externalization of `Resource` and `RecoveryCoordinator` references depends on the underlying RMI mechanism, all TRMI plug-ins must provide XActor with an implementation of the interface `StringRemoteRefConverter`, which supports conversions between remote references and strings. At appropriate times, XActor calls that interface to obtain strings that will be stored in specific records written out to the transaction log. The failure recovery procedure, when scanning the transaction log, uses that same interface to recover coordinator and/or resource references.

5 Related Work

JBoss [2] is an extensible application server that supports hot deployment of middleware components and, as a special case, allows dynamic deployment of components that implement new RMI mechanisms. Nevertheless, JBoss does not support dynamic deployment of components that implement *transactional* RMI mechanisms, as its TM has hard dependencies on the transports over which transactional interactions can be performed. This restriction is removed when XActor runs in JBoss. XActor fully exploits the dynamic deployment capabilities of that server and takes them to their logical consequence, with respect to TRMI mechanisms.

JBoss Transactions (JBossTS) [5], formerly Arjuna Transaction Service [11], is an open-source transaction manager that bears similarities to XActor. The former has modules that implement the OTS and WS-C/WS-AT standards atop a transaction manager core; the latter implements the same standards as dynamically deployable plug-ins. Some of the architectural differences between JBossTS and XActor reflect the time frames in which each project was started. XActor was specifically designed for Java EE environments, so it is centered at JTA and heavily influenced by OTS. Its key abstraction, the `TransactionBranch` interface, is an extension of `javax.transaction.Transaction`. This fact has many consequences. For example, it facilitates the transparent flow of incoming and outgoing transactional contexts. Such flow relies on the association between threads and `TransactionBranch`es, which is itself a byproduct of the thread–transaction association specified by JTA. The Arjuna project, on the other hand, predated OTS and Java EE. Its key abstractions (`BasicAction` and `AbstractRecord`)

were mapped to JTA interfaces, to OTS objects, and to WS-C/WS-AT endpoints. The JBossTS modules that support OTS and WS-C/WS-AT are much larger than the corresponding XActor plug-ins. Even though it is integrated with the JBoss application server, JBossTS does not yet afford dynamic deployment of new TRMI mechanisms.

6 Concluding Remarks

XActor demonstrates that transaction managers can support an open-ended set of transactional RMI mechanisms and transport protocols. Moreover, its integration with JBoss shows that it is possible to dynamically deploy new TRMI mechanisms into a running application server. The XActor innovations include the conceptual view of the remote interfaces of a transaction service as facets of transaction branch objects, the ability of performing distributed transactions over any combination of the currently deployed transports, and the deep integration, all the way down to transaction log, with TRMI plug-ins added to the transaction manager at runtime.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer-Verlag, 2004.

[2] M. Fleury and F. Reverbel. The JBoss Extensible Server. In *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[4] JBoss Remoting website, 2007. `http://labs.jboss.com/jbossremoting/`.

[5] JBoss Transactions website, 2007. `http://labs.jboss.com/jbosstm/`.

[6] C. Mohan and B. G. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *2nd ACM PODC*, pages 76–88, 1983.

[7] Object Management Group. *CORBA Transaction Service Specification, version 1.4*, March 2003.

[8] Object Management Group. *Common Object Request Broker Architecture: Core Specification, version 3.0.3*, March 2004.

[9] Organization for the Advancement of Structured Information Standards. *Web Services Atomic Transaction (WS-AtomicTransaction) 1.1*, April 2007.

[10] Organization for the Advancement of Structured Information Standards. *Web Services Coordination (WS-Coordination) 1.1*, April 2007.

[11] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *Computing Systems*, 8(3):255–308, 1995.

[12] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[13] Sun Microsystems. *Java Transaction API (JTA), v1.0.1B*, November 2002.

[14] Sun Microsystems. *Java Platform, Enterprise Edition (Java EE) Specification, v5*, May 2006.

[15] The Open Group. *Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., December 1991.
[16] The Open Group. *Distributed Transaction Processing: Reference Model, version 3*. X/Open Company Ltd., February 1996.
[17] W3 Consortium. *Web Services Addressing 1.0 — Core*, May 2006.
[18] C. Walls and R. Breidenbach. *Spring in Action, Second Edition*. Manning, 2007.