CHAPTER 3

The Java Transaction Service

In this chapter we look at one of the transaction specifications we briefly mentioned in Chapter 1, "Transaction Fundamentals," and see how many of those issues have been addressed by a real-world specification; this should help to make concrete some of the things we discussed earlier and give you an indication of just how much effort is involved in turning principles into practice. However, we're not looking at just any specification; we examine the one that is of most importance to J2EE: the Object Transaction Service from the Object Management Group. We look at the transaction service component of CORBA because that first became the standard for transactions in distributed object systems. It later became the standard for distributed transactions in J2EE when the Java Transaction API (JTA) recommended it for its underlying transaction system implementation (subtly renaming it the Java Transaction Service, or JTS). In this book we use the terms OTS and JTS interchangeably.

Although you are extremely unlikely to encounter OTS directly when writing your J2EE applications, you may be using it indirectly. The JTA that your beans or container actually uses may be layered on OTS for the purposes of interoperability with other transaction services or simply because of the legacy of the underlying transaction system; the JTA specification does not mandate the JTS but recommends it. Knowing whether or not there is a JTS operating in your favorite application server can be important, especially if you want to talk to foreign application servers/transactional applications.

NOTE We shall only be able to give an overview of OTS, since the actual specification is over 100 pages. Although we concentrate on the recent 1.4 version of the specification, we endeavor to indicate differences between it and the 1.1 version, which forms the basis of many existing implementations used in application servers.

The Common Object Request Broker Architecture (CORBA), as defined by the Object Management Group (OMG), is a standard derived by an industrial consortium including IBM, BEA and Hewlett-Packard, which promotes the construction of interoperable applications that are based upon the concepts of distributed objects. The architecture principally contains the following components:

- Object Request Broker (ORB), which enables objects to transparently make and receive requests in a distributed, heterogeneous environment. This component is the core of the OMG reference model.
- Object Services, a collection of services that support functions for using and implementing objects. Such services are considered to be necessary for the construction of any distributed application. Of particular relevance to this chapter is the Object Transaction Service (OTS).
- Common Facilities are other useful services that applications may need, but which are not considered to be fundamental such as desktop management and help facilities.

Now you may think that this sounds similar to J2EE and in many respects it is. You'll also find that much of the work the OMG did on various services such as transactions or messaging has found its way into J2EE: why reinvent the wheel when what exists works well?

NOTE Although we assume a basic level of CORBA understanding (particularly about IDL) you should be able to understand most of the concepts and interfaces described in this chapter without any prior knowledge.

CORBA predates J2EE by a decade and is not restricted to a single implementation language. Importantly for us, CORBA was the standard development platform for enterprise applications before Java and J2EE came along. Despite what you might have been lead to believe, there are a lot of legacy components and applications out there written in languages other than Java. In an enterprise application, you typically have to interact with these components sooner rather than later. As a result, interoperability between J2EE and CORBA quickly became important in J2EE.

The J2EE specification addressed this by requiring Java Remote Method Invocation (RMI) to utilize the CORBA message format IIOP (Internet Inter-ORB Protocol). This allows Java applications to invoke methods on CORBA objects written in any language. Transaction interoperability was obtained by recommending that JTA implementations use transaction services written using the JTS.

The OTS in a Nutshell

The OTS provides interfaces that allow multiple, distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. However, the transaction service does not require all objects to have transactional behavior. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others. Transaction information may be propagated between client and server explicitly, or implicitly, giving the programmer finer-grained control over an object's transactionality. Objects supporting (partial) transactional behavior must reside within domains (Portable Object Adapters in CORBA parlance) that have appropriate transaction *policies* defined on them. The OTS specification allows transactions to be nested. However, an implementation need not provide this functionality. Appropriate exceptions are raised if an attempt is made to use nested transactions in this case.

> **NOTE** In the 1.1 version of OTS specification objects had to have interfaces derived from the `TransactionalObject` interface. This was deprecated in 1.2.

The transaction service also distinguishes between *recoverable objects* and *transactional objects*:

- Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or rolls back (aborts) to ensure the consistency of the state changes. This is achieved be registering appropriate objects that support the `Resource` CORBA interface (or the derived `SubtransactionAwareResource` interface) with the current transaction. Recoverable objects are also by definition transactional objects.
- In contrast, a simple transactional object need not necessarily be a recoverable object if its state is actually implemented using other recoverable objects. A simple transactional object need not take part in the commit protocol used to determine the outcome of the transaction since it does not maintain any state itself, having delegated that responsibility to other recoverable objects that will take part in the commit process.

The OTS is simply a *protocol engine* that guarantees that transactional behavior is obeyed but does not directly support all of the transaction properties given above. As such it requires other co-operating services that implement the required functionality, including:

1. *Persistence/Recovery Service*. Required to support the atomicity and durability properties.
2. *Concurrency Control Service*. Required to support the isolation properties.

The application programmer is responsible for using these services to ensure that transactional objects have the necessary ACID properties.

NOTE It is important to understand that OTS does not define how to implement a transaction service, only what is expected of an implementation at the interface level and (roughly) behavior that is visible through these interfaces.

What this means is that many of the issues we discussed in Chapter 1, "Transaction Fundamentals," such as transaction logs, two-phase concurrency control and so on are assumed by OTS: you won't find anywhere in its 120+ pages a description of how to write a transaction log that performs well under load, for instance. So, although just taking the OTS specification and trying to write a conformant implementation is possible, it is unlikely to result in an implementation that performs or even works under all failure scenarios. Unfortunately, many new entrants to the field of transaction processing have failed to understand this, so you should beware: just because something is OTS compliant doesn't mean it is resilient, performs well, scales or can tolerate failures.

The Java Transaction Service

As with all OMG specifications, the Object Transaction Service is not language specific, i.e., it can be implemented just as easily in C++ as COBOL. Therefore, the Java Transaction Service specification is a Java language mapping of OTS. The advantage of using a JTS-compliant implementation is that, in theory at least, it allows interoperation with other JTS implementations.

Unfortunately, OTS interoperability has never been high on the agenda of transaction service vendors and hence the definers of the specification: it's not in their interests to allow you to buy implementations from different vendors. Problems with interoperability began to be addressed with truly interoperable versions of CORBA and then with OTS 1.2. So look out for implementations based on this version of the specification as you're much more likely to not suffer from vendor lock-in in the future.

NOTE Although the CORBA specification is now at version 1.4, the JTS specification only references OTS 1.1. Although transaction service implementations compliant with 1.2 and above provide the potential for better interoperability, unfortunately there is no requirement for JTS compliant implementations to support these versions.

Relationship to Other Transaction Standards

The OTS specification was developed by some of the main players in the transaction processing arena. Although it was meant as the model for next generation transaction processing systems, it was impractical to believe that existing systems (e.g., IBM's CICS) would be replaced by new implementations based on OTS. Much trust in the reliability, performance and functionality of

these systems has been built up over many years and for many critical applications it is this trust that is more important than the underlying model on which they are based.

Therefore, one of the important aspects of OTS was that it should be able to interoperate with the main legacy transaction processing implementations and their associated models. OTS implementations can therefore interact with many of the transaction systems we presented in Chapter 1, "Transaction Fundamentals." It is slightly ironic that interoperability with legacy systems was supported from the beginning whereas interoperability with other OTS implementations was not.

The OTS Architecture

The architecture of OTS is captured in Figure 3-1. The OTS defines interfaces to a transaction service implementation. That implementation may be a pre-existing system, based on another standard or vendor-specific protocol, or it may be written from scratch based on the requirements dictated by OTS. However, the OTS interfaces can essentially be grouped into those available to a client (shown in the box beneath the transaction originator in the figure) and those available to the server (shown in the box below the recoverable server). You'll notice that there is some overlap between these interfaces; as we see later this is because there are situations where these interfaces need to be available to both the client and the server.

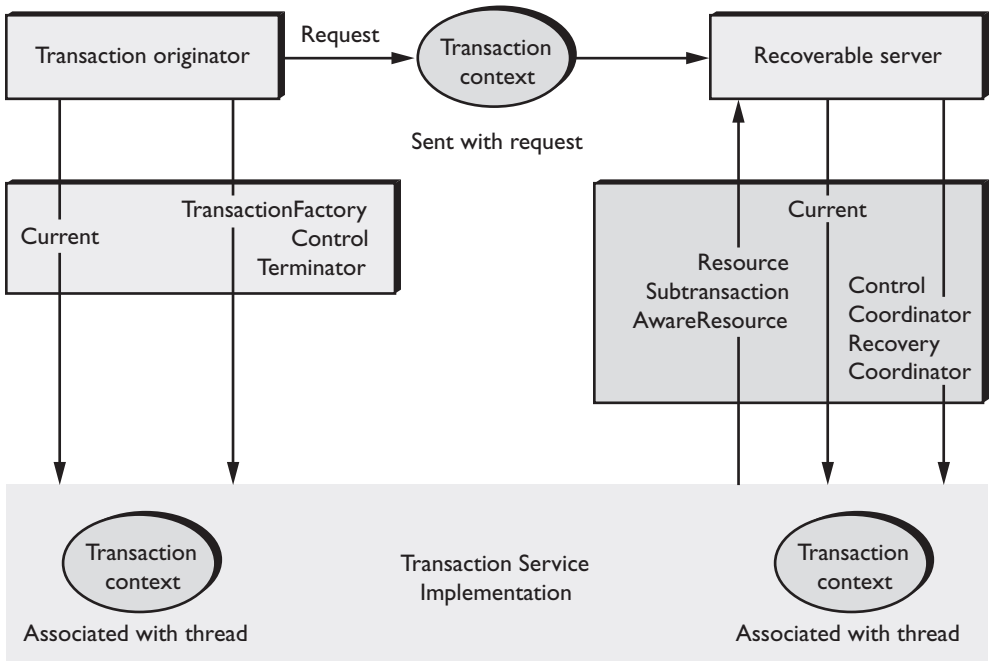

Figure 3-1 The OTS architecture.

We now briefly describe the roles that these various interfaces play in the OTS specification:

- `Current` is the application programmers' typical way to interact with the transaction implementation and allows transactions to be started and ended. Transactions created using `Current` are automatically associated with the invoking thread. It is a per-thread object, so that it must remember the transaction associated with each thread that has used it. The underlying implementation will typically use the `TransactionFactory` for creating top-level transactions. Nested transactions are an optional extra.
- The `Control` interface provides access to a specific transaction and actually wraps the transaction `Coordinator` and `Terminator` interfaces which are used to enlisting participants and ending the transaction respectively. One of the reasons for splitting this functionality into two interfaces was to allow a transaction implementation to have finer control over the entities that could terminate the transaction.
- The `Resource`/`SubtransactionAwareResource` interfaces represent the transaction participants and have a deliberately generic interface to allow any (two-phase) compliant implementation to be registered with the transaction rather than just an XA compliant implementation.
- Each top-level transaction has an associated `RecoveryCoordinator` that is available to participants in order for them to drive failure recovery. As we mentioned earlier, recovery after a crash will almost certainly be driven from the transaction coordinators' end but allowing participants to drive it as well can improve recovery time.
- The transaction context is fundamental to any distributed transaction system and OTS is no different in this respect.

In the following sections we look at these different interfaces and how they are used within OTS.

Application Programming Models

A client application program may use *direct* or *indirect* context management to manage a transaction. With indirect context management, an application uses `Current` to associate the transaction context with the application thread of control. In direct context management, an application manipulates the `Control` object and the other objects associated with the transaction: the application threads are not automatically associated with transactions through this mechanism, which gives the application more control over thread-to-transaction association.

Furthermore, OTS allows an object to specify whether transactions should be explicitly or implicitly propagated to its operations:

- *Explicit propagation* means that an application propagates a transaction context (defined by what OTS calls a `PropagationContext`) by passing objects defined by the transaction service as explicit parameters. How the context is explicitly propagated is not mandated by OTS: for example, a `PropagationContext` or `Control` object may be used.
- *Implicit propagation* means that requests are implicitly associated with (share) the client's transaction. The context is transmitted implicitly to the objects, without direct client intervention. Implicit propagation depends on indirect context management, since it propagates the transaction context associated with `Current`. An object that supports implicit propagation would not typically expect to receive a `PropagationContext` (for example) object as an explicit parameter.

A client may use one or both forms of context management, and may communicate with objects that use either method of context propagation. This results in four ways in which client applications may communicate with transactional objects:

1. *Direct Context Management/Explicit Propagation*: The client application directly accesses the `Control` object, and the other objects that describe the state of the transaction. To propagate the transaction to an object, the client must include the context as an explicit parameter of an operation.
2. *Indirect Context Management/Implicit Propagation*: The client application uses operations on `Current` to create and control its transactions. When it issues requests on transactional objects, the transaction context associated with the current thread is implicitly propagated to the object.
3. *Indirect Context Management/Explicit Propagation*: For an implicit model application to use explicit propagation, it can get access to the `Control` using the `get_control` operation provided by `Current`. It can then use an instance of `PropagationContext` as an explicit parameter to a transactional object. This is explicit propagation.
4. *Direct Context Management/Implicit Propagation*: A client that accesses the transaction service objects directly can use the `resume` operation provided by `Current` to set the implicit transaction context associated with its thread (set up the thread-to-transaction association). This allows the client to invoke operations of an object that requires implicit propagation of the transaction context.

The main difference between direct and indirect context management is the effect on the invoking thread's transaction context. If using indirect, the thread's transaction context *will* be modified automatically by OTS, e.g., if `begin` is called then the thread's notion of the current transaction will be modified to the newly created transaction; when that is terminated, the transaction previously associated with the thread (if any) will be restored as the thread's context (assuming nested transactions are supported by the OTS implementation). However, if using

direct management, no changes to the thread's transaction context are performed by OTS: the application programmer assumes responsibility for this.

Let's now look at the various components of OTS in more detail. We start at the top, with the entity that is responsible for creating transactions: the transaction factory.

The Transaction Factory

The `TransactionFactory` interface is provided to allow the transaction originator to begin a top-level transaction. Nested transactions must be created using the `begin` method of `Current`, or the `create_subtransaction` method of the parent's `Coordinator`. Operations on the factory and `Coordinator` to create new transactions are direct context management, and as such will not modify the calling thread's transaction context.

The `create` operation creates a new top-level transaction and returns its `Control` object, which can be used to manage or control participation in the new transaction. When an application invokes `create` it can also provide an application specific timeout value, in seconds: if the transaction has not completed before this timeout has elapsed then the transaction service will roll it back. If the parameter is zero, then no application specified timeout is established.

> **NOTE** Nested transactions do not have a timeout associated with them and so will only be automatically rolled back if their enclosing top-level transaction is rolled back.

This sequence of operations can be represented in UML as shown in Figure 3-2:

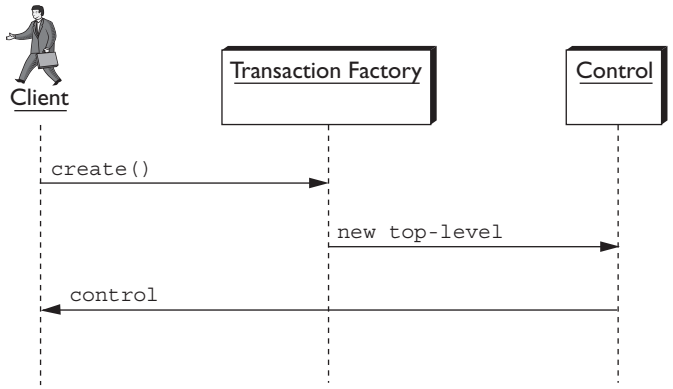

Figure 3-2 Top-level transaction creation (direct mode).

The client contacts the `TransactionFactory` and invokes the `create` method. This causes the factory to create and start a new top-level transaction and eventually return a `Control` reference to the client for the transaction. As we see, the client can then use this reference later to manipulate the transaction.

The specification allows the `TransactionFactory` to be a separate server from the application (e.g., a typical Transaction Monitor), which transaction clients and services share and which manages transactions on their behalf. However, the specification also enables the `TransactionFactory` to be implemented by an object within the same process (JVM) as the clients/services that use it. This has obvious performance and fault-tolerance benefits but can be a management nightmare. Nearly all OTS implementations support the Transaction Monitor approach, with several also offering the co-located option. In our experience, a well-designed transaction service that offers both styles is the best of both worlds, allowing you to choose the location of the transaction factory to best suit your applications. Remember that the choice may well vary from application to application, so only being given one option may severely limit your scope for future transactional developments.

Managing Transaction Contexts

As we saw in Chapter 1, "Transaction Fundamentals," the *transaction context* is fundamental to any distributed transaction architecture and OTS is no different. Each thread is potentially associated with a context and this association may be null, indicating that the thread has no associated transaction, or it refers to a specific transaction. The OTS explicitly allows contexts to be shared across multiple threads. In the presence of nested transactions a context remembers the stack of transactions started within the environment such that when the nested transaction ends the context of the thread can be restored to that in effect before the nested transaction was started.

This relationship is shown in Figure 3-3 in UML where, as we've already seen, `Current` is the object used by a thread for manipulating its transaction context information (represented by `Control` objects) and performing the thread-to-transaction association:

What this diagram shows is that `Current` maintains a hierarchy of transactions (zero-to-many transactions) for every thread in the application. These transactions are represented by `Control` objects.

As we mentioned earlier, management of transaction contexts may be undertaken by an application in either a direct or an indirect manner. In the direct approach the transaction originator issues a request to a `TransactionFactory` to begin a new top-level transaction. The factory returns a `Control` object that enables two further interfaces to be obtained. These latter interfaces allow an application to end the transaction (via a `Terminator`), to become a participant in the transaction, or to start a nested transaction (both via a `Coordinator`). These interfaces (shown in Code Listing 3-1) may be passed as explicit parameters in operation invocations since transaction creation using these interfaces does not change a thread's current context.

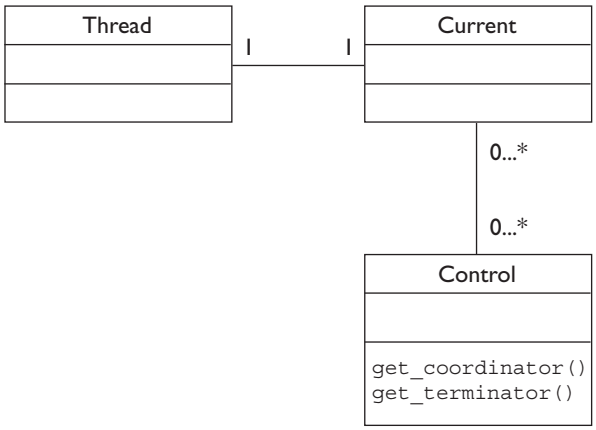

Figure 3-3 Thread and context relationship.

Code Listing 3-1 Direct context management interface.

```
interface Terminator
{
    void commit (in boolean report_heuristics) raises (HeuristicMixed,
HeuristicHazard);
    void rollback ();
};

interface Coordinator
{
    Status get_status ();
    Status get_parent_status ();
    Status get_top_level_status ();

    RecoveryCoordinator register_resource (in Resource r) raises
(Inactive);
    Control create_subtransaction () raises
(SubtransactionsUnavailable, Inactive);
    void rollback_only () raises (Inactive);
    . . .
};

interface Control
{
    Terminator get_terminator () raises (Unavailable);
    Coordinator get_coordinator () raises (Unavailable);
};
```

```
interface TransactionFactory
{
        Control create (in unsigned long time_out);
};
```

The relationship between a `Control` and its `Coordinator` and `Terminator` interfaces is shown in Figure 3-4. For every `Control` there is exactly one `Coordinator`, and every `Coordinator` is associated with exactly one `Control`. However, the same cannot be said for the `Terminator`. The OTS allows a `Control` to have no `Terminator` (the implementation may wish to restrict who can end a transaction, for example, and so could ensure that the `Control` has a null `Terminator`).

As mentioned earlier, when a transaction is created by the factory it is possible to specify a timeout value in seconds; if the transaction has not completed within this timeout then it is subject to possible rollback. If the timeout value is zero then no application specific timeout will be set.

In contrast to explicit context management, implicit context management is handled by the `Current` interface (shown in Code Listing 3-2), which provides simplified transaction management functionality and automatically creates nested transactions as required. Transactions created using this interface do alter a thread's current transaction context, i.e., the thread's notion of the current transaction changes appropriately.

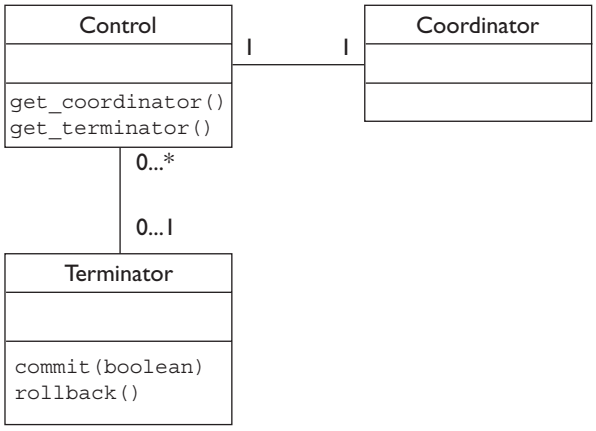

Figure 3-4 Control relationship.

Code Listing 3-2 Indirect Context Management Interface

```
interface Current : CORBA::Current
{
    void begin () raises (SubtransactionsUnavailable);
    void commit (in boolean report_heuristics) raises
(NoTransaction, HeuristicMixed, HeuristicHazard);
    void rollback () raises (NoTransaction);
    void rollback_only () raises (NoTransaction);
    . . .
    Control get_control ();
    Control suspend ();
    void resume (in Control which) raises (InvalidControl);
};
```

Nested Transactions

We saw in Chapter 1, "Transaction Fundamentals," how nested transactions can be a useful structuring tool for modularity and fault-tolerance. The OTS was the first industrial transaction specification to include support for nested transactions. Unfortunately, the provision of nested transaction by an OTS implementation is optional, so just because you find an implementation of OTS doesn't necessarily mean you've found something that supports nested transactions. Remember also that you'll need to have subtransaction-aware resources in order to actually be able to do anything useful with your newly found nested transactions.

When nested transactions (subtransactions) are provided, the transaction context forms a hierarchy. Resources acquired within a subtransaction should be inherited (retained) by parent transactions upon the commit of the subtransaction, and (assuming no failures) only released when the top-level transaction completes, i.e., they should be retained for the duration of the top-level transaction. If a subtransaction rolls back, it can release any resources it acquired, and undo any changes to resources it inherited.

Unlike top-level transactions, in OTS subtransactions behave differently at commit time. Whereas top-level transactions undergo a two-phase commit protocol, nested transactions in OTS do not perform any commit protocol: when a program commits a nested transaction, the transaction is considered committed and it simply informs any registered resources of its outcome. If a resource cannot commit then it raises an exception, and the OTS implementation is free to ignore this or attempt to roll back the subtransaction. Obviously rolling back a subtransaction may not be possible if some resources have already been told that the transaction has committed.

Because a two-phase commit protocol is not used by OTS, this can lead to non-atomic behavior. Therefore, most sensible implementations will either cause the enclosing transaction (which may or may not be a subtransaction) to rollback, thus guaranteeing that all work will be undone, or will extend the protocol and (in an implementation-specific manner) actually run a

two-phase commit protocol on all subtransaction-aware resources. (Several systems that do support nested transactions use a two-phase commit protocol to terminate them.)

If you use an OTS with subtransactions, you should certainly attempt to verify which, if any, mechanism the implementation uses to ensure the integrity of your application and its data. If there is no such mechanism is in place, then nested transactions should be used with extreme care.

Transaction Propagation

So you've written your transactional object/service and you now want to make sure that remote transactional clients can use it within the scope of their transactions, i.e., you need to ensure that transaction contexts are propagated between clients and services. How do you accomplish this? In this section we look at how things used to work in older versions of the OTS specification and how they're now supposed to work. The reason for comparing and contrasting is that at the time of writing most existing implementations of OTS still use the original, older form of context propagation.

The OTS supports both implicit (system driven) propagation and explicit (application driven) propagation of transactional behavior. In the implicit case transactional behavior is specified in an operation signature or Portable Object Adapter (POA) policy and any transaction context associated with the calling thread is automatically sent with each operation invocation.

With explicit propagation, applications must define their own mechanism for propagating transactions. This allows two options:

- A client can control whether its transaction is propagated with any operation invocation. When using POA policies, it is assumed that all methods of the invoked object are transactional, which may not actually be the case. If there are some non-transactional methods, propagating the context will be an unnecessary overhead.
- A client can invoke operations on both transactional and non-transactional objects within a transaction.

Note that transaction context management and transaction propagation are different things that may be controlled independently of each other. Furthermore, mixing of direct and indirect context management with implicit and explicit transaction propagation is allowed. Use of implicit propagation requires co-operation from the ORB, in that the current context associated with the thread must be sent with any operation invocations by a client and extracted by the server prior to actually calling the target operation.

In the following sections we look at how implicit context propagation occurs. This uses the `PropagationContext` structure defined by OTS. As you might be able to guess from what we saw in Chapter 1, "Transaction Fundamentals," this structure contains a reference to the transaction coordinator and the timeout value associated with the transaction; if the transaction is nested, then the entire hierarchy is also defined.

`TransactionalObject` Interface

In the 1.1 version of the OTS specification, the empty `TransactionalObject` interface is used by an object to indicate that it is transactional. By supporting this interface, an object indicates that it wants the transaction context associated with the client thread to be associated with all of its operations.

An OTS implementation is not required to initialize the transaction context of every request handler. It is required to do so only if the interface supported by the target object is derived from `TransactionalObject`. Otherwise, the initial transaction context of the thread is undefined. A transaction service implementation can raise the `TRANSACTION_REQUIRED` exception if a `TransactionalObject` is invoked outside the scope of a transaction, i.e., the transaction context is null.

In a single-address space application (i.e., all objects reside within the same process), transaction contexts are implicitly shared between "clients" and objects, regardless of whether or not the objects support the `TransactionalObject` interface.

OTSPolicy

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 are similar in many respects to those possessed by J2EE containers.

The main problem with `TransactionalObject` was that it was an attribute of the type of object and not of its deployment. What this meant was that if you wanted a transactional and non-transactional instance of the object you'd be required to have two different types. With the advent of the CORBA POA, it was possible to remove this restriction and make transactionality a property of the way in which the object is deployed.

The POA policies and their OTS 1.1 equivalents are defined as shown in Table 3-1.

Table 3-1 OTS 1.2 transaction behaviors

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

- 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).

Examples

To aid in comprehension of the above discussions, Code Listing 3-3 illustrates a simple transactional client using both direct context management and explicit transaction propagation.

Code Listing 3-3 Simple transactional client using direct and explicit

```
{
    . . .
    org.omg.CosTransactions.Control c;
  org.omg.CosTransactions.Terminator t;
    org.omg.CosTransactions.PropagationContext pgtx;

  c = transFact.create(0);          // create top-level transaction

    pgtx = c.get_coordinator().get_txcontext();
  ...
  trans_object.operation(arg, pgtx);              // explicit
propagation
  ...
  t = c.get_terminator();            // get terminator
  t.commit(false);            // so it can be used to commit
  ...
}
```

NOTE Many of the examples we use in this chapter are only partial code fragments, illustrating the pertinent aspects. If you want to see working examples and the later demonstration code, then take a look at the Arjuna reference from **web-siteURL**#Need URL

In contrast, Code Listing 3-4 shows the same program using indirect context management and implicit propagation. This example is considerably simpler since the application only has to be concerned with starting and then committing or aborting actions.

Code Listing 3-4 Simple transactional client using indirect and implicit

```
{
    . . .
  ...
  current.begin();              // create new transaction
  ...
  trans_object2.operation(arg);              // implicit propagation
  ...
  current.commit(false);              // simple commit
  ...
}
```

Finally, Code Listing 3-5 illustrates the potential flexibility of OTS by using both direct and indirect context management in conjunction with explicit and implicit transaction propagation.

Code Listing 3-5 Mixed transactional client

```
{
    . . .
  org.omg.CosTransactions.Control c;
  org.omg.CosTransactions.Terminator t;
    org.omg.CosTransactions.PropagationContext pgtx;

  c = transFact.create(0);          // create top-level transaction
    pgtx = c.get_coordinator().get_txcontext();

  current.resume(c);            // set implicit context
  ...
  trans_object.operation(arg, pgtx);              // explicit
propagation
  trans_object2.operation(arg);              // implicit propagation
  ...
  current.rollback();            // oops! rollback
  ...
}
```

Handling Heuristics

In Chapter 1, "Transaction Fundamentals," we saw how transaction heuristics are an inevitable component of transaction systems and the OTS specification is no different. To recap, heuristics are the means whereby the traditional blocking nature of a transaction two-phase commit protocol can be unblocked: prepared participants can take autonomous decisions about whether to ultimately commit or rollback if they do not get the coordinator's final decision or it is not received in a timely manner. However, it's important to remember that heuristics should be used with care and only in exceptional circumstances since there is the possibility that the decision

will differ from that determined by the transaction service and will thus lead to a loss of integrity in the system.

In terms of OTS, if a heuristic decision is made by a participant then an appropriate exception is raised during commit/rollback processing. The OTS supports an exception for each type of heuristic, i.e., `HeuristicRollback`, `HeuristicCommit`, `HeuristicMixed` and `HeuristixHazard`. Heuristics are ordered such that `HeuristicMixed` takes priority over `HeuristicHazard`.

Rather than always being told about heuristics, a client can choose to ignore them; we'd obviously not recommending this approach unless you are sure about the resources that will participate within your transactions and how they will behave in the event a heuristic occurs. The commit operation of `Current` or the `Terminator` takes a Boolean parameter that allows the caller to specify whether or not heuristic reporting is required.

> **NOTE** As we saw in Chapter 2, the JTA took the approach of not allowing applications to ignore heuristics. If a heuristic occurs, you're application will be informed.

However, heuristics could occur after the transaction has officially terminated: for example, if not all participants can be contacted during the second phase of the commit protocol, the failure recovery system may be responsible for completing the transaction later. If that happens, any heuristic outcomes cannot be reported directly to the application, which may have long since terminated. Early versions of the OTS specification allowed heuristic outcomes to be reported via the CORBA Event service. However, this was removed in more recent versions. As such, heuristic notification can be something that is overlooked or provided in an implementation-specific manner. We recommend that you do a little investigation in this area to determine exactly what your favorite implementation does, especially in the event of heuristics occurring during failure recovery: if they aren't reported, we suggest going elsewhere.

Transaction Controls

The `Control` interface allows a program to explicitly manage or propagate a transaction context. An object supporting the `Control` interface is associated with one specific transaction. The `Control` interface supports two operations, `get_terminator` and `get_coordinator`, which return instances of the `Terminator` and `Coordinator` interfaces respectively. Both of these methods throw the `Unavailable` exception if the `Control` cannot provide the requested object, e.g., the transaction has terminated. An OTS implementation can restrict the ability for the `Terminator` and `Coordinator` to be used in other execution environments or threads; at a minimum the creator must be able to use them.

The `Control` object for a transaction can be obtained when the transaction is created either using the `TransactionFactory` or the `create_subtransaction` method defined by the `Coordinator` interface. In addition, it is possible to obtain a `Control` for the current transaction using the `get_control` or `suspend` methods provided by `Current`.

The `Terminator`

Each transaction has an individual `Terminator` and applications can use the `Terminator` to commit or rollback the transaction. Typically these operations are used by the same thread that started the transaction. As we mentioned earlier, you can use the `Terminator` to manage your transactions (direct context management) but if you do it will mean that the invoking thread's notion of the current transaction will not be changed.

It is possible for a transaction to be terminated directly (i.e., through the `Terminator`) and then an attempt to terminate the transaction again through `Current` can be made (or vice versa). In this situation, an exception will be thrown for the subsequent termination attempt.

The `commit` operation attempts to commit the transaction: as we saw in Chapter 1, "Transaction Fundamentals," to successfully commit, the transaction must not have been marked as rollback only (e.g., the `rollback_only` method provided by `Current`) and all of its participants must agree to commit. Otherwise, the transaction will be forced to rollback and the `TRANSACTION_ROLLEDBACK` exception will be thrown by `commit` to indicate this fact to the application.

If the `report_heuristics` argument is `false`, the `commit` operation can complete as soon as the `Coordinator` has made its decision to commit or rollback the transaction, i.e., as soon as the prepare phase has completed. The application is not required to wait for the transaction coordinator to complete the second phase of the protocol. This can significantly reduce the elapsed time for the `commit` operation, especially where participant `Resource` objects are located on remote network nodes. However, no heuristic conditions can be reported to the application in this case. So, as we said before, you'd better be confident that ignoring heuristics, as this does, will not adversely affect your application. You should weigh the possible performance gains against the fact that you may not know about loss of application consistency until much later (if at all).

The `report_heuristics` option set to `true` guarantees that the `commit` operation will not complete until the coordinator has completed the `commit` protocol with all `Resource` objects involved in the transaction. This guarantees that the application will be informed of any non-atomic outcomes of the transaction via the `HeuristicMixed` or `HeuristicHazard` exceptions, but increases the application-perceived elapsed time for the `commit` operation. You might wonder why `HeuristicRollback` isn't one of the possible exceptions that can be thrown by `commit`: if a participant heuristically rolls back, the coordinator will either be able to roll back all other participants (if the heuristic participant was the first one encountered during the second phase of the commit protocol, for instance), or it will have told the other participants to commit. In the former case, the `TRANSACTION_ROLLEDBACK` exception is thrown, whereas in the latter it will be `HeuristicMixed` or `HeuristicHazard`.

When a transaction is committed, the coordinator will drive any registered `Resources` using their `prepare`/`commit` methods. It is the responsibility of these `Resources` to ensure that any state changes to recoverable objects are made permanent to guarantee the ACID properties, as described in Chapter 1, "Transaction Fundamentals."

When `rollback` is called, the registered `Resources` are responsible for guaranteeing that all changes to recoverable objects made within the scope of the transaction (and its descendants) is undone. All resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the resources. Remember that no heuristics can occur here because there is no prepare phase.

The `Coordinator`

As you might guess from the name, the `Coordinator` is the main interface to the transaction coordinator. Each `Coordinator` represents a specific transaction and is obtained by the `get_coordinator` method of `Control`. For a number of reasons, including architecture and compatibility with existing transaction service implementations, the `Coordinator` does not actually contain operations for starting or ending the transaction. What it does contain, however, are operations needed by servers to enable them to enlist participants (resources) in the transaction represented by the `Coordinator`. These participants are typically either recoverable objects or agents of recoverable objects, for example a resource controlling updates on a database. In addition, as we saw in Chapter 1, "Transaction Fundamentals," these participants could be subordinate coordinators for interposition.

The operations supported by the `Coordinator` interface of interest to application programmers are:

- `get_status`, `get_parent_status`, `get_top_level_status`: these operations return the status of the associated transaction. At any given time a transaction can have one of the following status values representing its progress:
  - `StatusActive`: The transaction is currently running and has not been asked to prepare or been marked for rollback.
  - `StatusMarkedRollback`: The transaction has been marked for rollback. This is the only possible outcome for the transaction.
  - `StatusPrepared`: The transaction has been prepared, i.e., all subordinates have responded `VoteCommit`.
  - `StatusCommitted`: The transaction has completed commitment. It is likely that heuristics exist, otherwise the transaction would have been destroyed and `StatusNoTransaction` returned.
  - `StatusRolledBack`: The transaction has rolled back. It is likely that heuristics exist, otherwise the transaction would have been destroyed and `StatusNoTransaction` returned.
  - `StatusUnknown`: The transaction service cannot determine the current status of the transaction. This is a transient condition, and a subsequent invocation will ultimately return a different status.
  - `StatusNoTransaction`: No transaction is currently associated with the target object. This will occur after a transaction has completed successfully (either commit or rollback).

  - `StatusPreparing`: The transaction is in the process of preparing and has not yet determined the final outcome.
  - `StatusCommitting`: The transaction is in the process of committing.
  - `StatusRollingBack`: The transaction is in the process of rolling back.
- `is_same_transaction`: This operation and a number of similar operations can be used for transaction comparison. Resources may use these various operations to guarantee that they are registered only once with a specific transaction.
- `hash_transaction`, `hash_top_level_tran`: This operation returns a hash code for the specified transaction.
- `register_resource`: This operation registers the specified `Resource` as a participant in the transaction. The `Inactive` exception is raised if the transaction has already been prepared: it makes no sense to enlist a participant with a transaction that has already completed the first phase of the commit protocol, since the new participant wouldn't be able to take part in the termination protocol. The `TRANSACTION_ROLLEDBACK` exception is raised if the transaction has been marked rollback only; the OTS designers decided that there was little point in being able to enlist a participant with a transaction when the transaction was guaranteed to rollback. If the `Resource` is a `SubtransactionAwareResource` *and* the transaction is a subtransaction, then this operation registers the `Resource` with this transaction and indirectly with the top-level transaction when the subtransaction's ancestors have committed. Otherwise, the `Resource` will only be registered with the *current* transaction. This operation returns a `RecoveryCoordinator` that can be used by this `Resource` during recovery, as we see later. Note that there is no ordering of registered `Resources` implied by this operation, i.e., if `Resource` A is registered after `Resource` B, OTS is free to operate on them in any order when the transaction terminates. Therefore, `Resources` should not be implemented that assume (or require) such an ordering to exist.

> **NOTE** Some implementations of OTS do allow users to specify orderings of resources (the Hewlett-Packard/Arjuna Technologies Transaction Service, for example) where this is required, but obviously this is not a feature of the standard and may not exist if you are migrating code between implementations.

- `register_subtran_aware`: This operation registers the specified subtransaction-aware resource with the current transaction such that it will be informed when the subtransaction commits or rolls back. This method does not register the resource as a participant in the top-level transaction, however. The `NotSubtransaction` exception is raised if the current transaction is not a subtransaction. As with `register_resource`, no ordering is implied by this operation.

- `register_synchronization`: This operation registers the `Synchronization` object with the transaction such that it will be invoked prior to prepare and after the transaction has completed. `Synchronizations` can only be associated with top-level transactions and an exception (`SynchronizationsUnavailable`) will be raised if an attempt is made to register a `Synchronization` with a subtransaction. As with `register_resource`, no ordering is implied by this operation.
- `rollback_only`: This operation marks the transaction so that the only possible outcome is for it to roll back. The `Inactive` exception is raised if the transaction has already been prepared or completed. This could happen if one thread terminates the transaction while another one (possibly in another process) tries to mark it or roll back only.
- `create_subtransaction`: A new subtransaction is created whose parent is the current transaction. The `Inactive` exception is raised if the current transaction has already been prepared or completed. As we mentioned earlier, an implementation of the transaction service need not support nested transactions, in which case the `SubtransactionsUnavailable` exception is raised. It's unlikely you're going to write subtransaction-aware applications without first determining that the transaction service you are using supports subtransactions, so you may never see this exception.

`Current`

We've seen the interfaces that define the transaction coordinator and control the interactions with it, but how do application programs actually start or end transactions, and in a multithreaded environment, how are transactions associated with specific threads? The answer to this question is simple: the `Current` interface. We've seen that applications (clients or servers) could create transactions through the `TransactionFactory` and terminate them via the associated `Terminator`: direct context management as OTS calls it. However, as we mentioned earlier, transactions manipulated in this way don't affect the thread. Also, having to go through all of these various interfaces to start or end a transaction, especially if you want to create nested transactions, can be cumbersome.

`Current` defines operations that allow a client to explicitly manage the association between threads and transactions, i.e., indirect context management. Importantly, it also defines operations that simplify the use of the transaction service. For example, starting a transaction is extremely simple via `Current`, regardless of whether or not that transaction is nested.

As you might imagine, `Current` supports many of the same operations of the `Terminator` and `Coordinator`. However, there are several important differences and omissions: `Current` was designed with clients in mind, so there is no direct way to register resources – you can register them, but you'll first have to get the `Control` and then the `Coordinator` from it. Transaction comparison isn't possible directly—again, you'll need to get the `Coordinator` if you want to do this.

With this in mind, we now examine the operations:

- `begin`: A new transaction is created and associated with the current thread. If the client thread is currently associated with a transaction and the OTS implementation supports nested transactions, the new transaction is a subtransaction of that transaction. Otherwise, the new transaction is a top-level transaction. If the OTS implementation does not support nested transactions, the `SubtransactionsUnavailable` exception may be thrown. The thread's notion of the current context will be modified to this transaction.
- `commit`: The transaction commits; if the client thread does not have permission to commit the transaction, the `NO_PERMISSION` exception is raised. The effect is the same as performing the `commit` operation on the corresponding `Terminator`. The thread's transaction context is returned to the state prior to the `begin` request. As with the `Terminator`, heuristic reporting can be turned on or off depending upon the Boolean parameter to `commit`.
- `rollback`: The transaction rolls back; if the client thread does not have permission to terminate the transaction, the standard exception `NO_PERMISSION` is raised. The effect is the same as performing the rollback operation on the corresponding Terminator object. The client thread transaction context is returned to the state prior to the `begin` request.
- `rollback_only`: The transaction is modified so the only possible outcome is for it to rollback. If the transaction has already been terminated (or is in the process of terminating) an appropriate exception will be thrown.
- `get_status`: This operation returns the status of the current transaction, or `StatusNoTransaction` if there is no transaction associated with the thread.
- `set_timeout`: This operation modifies the timeout associated with *top-level transactions* for subsequent begin requests *for this thread only.* Subsequent transactions will be subject to being rolled back if they have not completed after the specified number of seconds. It is implementation dependant as to what timeout value will be used for a transaction if one is not explicitly specified prior to begin. Most implementations impose a default value of tens of seconds, although some may not impose a timeout value at all. You should obviously find out what the implementation you are using does.
- `get_timeout`: This operation is used for obtaining the current timeout associated with a thread. Because timeouts may be changed at any time, this value isn't necessarily the value associated with a transaction created by the same thread. Unfortunately the specification doesn't define an easy way in which to get the timeout value of a transaction. You can, however, get hold of the `PropagationContext` for the current transaction and read the timeout from there.
- `get_control`: If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, a `Control` object is returned that represents the

current transaction. The operation is not dependent on the state of the transaction, so you can get hold of the transaction `Control` no matter its state; of course, what you can do with it afterward will depend upon the state.

- `suspend`: If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, the `Control` that represents the transaction context is returned. This object can be given to the `resume` operation to re-establish this context in a thread. The operation is not dependent on the state of the transaction. When this call returns, the current thread has no transaction context associated with it.
- `resume`: If the parameter is a null `Control`, the client thread becomes associated with no transaction. Otherwise, if the parameter is valid in the current execution environment, the client thread becomes associated with that transaction. Any previous transaction will be forgotten by the thread.

If we consider the creation of a top-level transaction using `Current`, the course of events within OTS can be represented as shown in Figure 3-5.

The client starts the transaction via `Current`, which will invoke the `create` method on a `TransactionFactory`. How the reference to the factory is obtained will depend on the OTS implementation. For example, it might be co-located with the client or it might be published in some naming service. The factory creates and starts a new top-level transaction and returns the reference to that transaction (the `Control`) to `Current`, which associates it with the client's thread.

Likewise, creation of a subtransaction through `Current` can be represented as shown in Figure 3-6. As you can see, this is different to the first case because the creation of a subtransac-

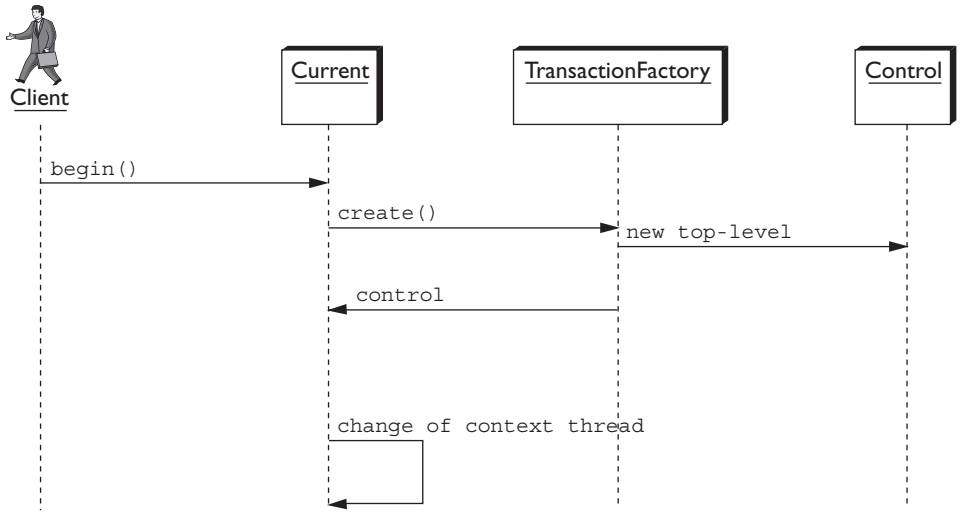

Figure 3-5 Using `Current` to create a top-level transaction

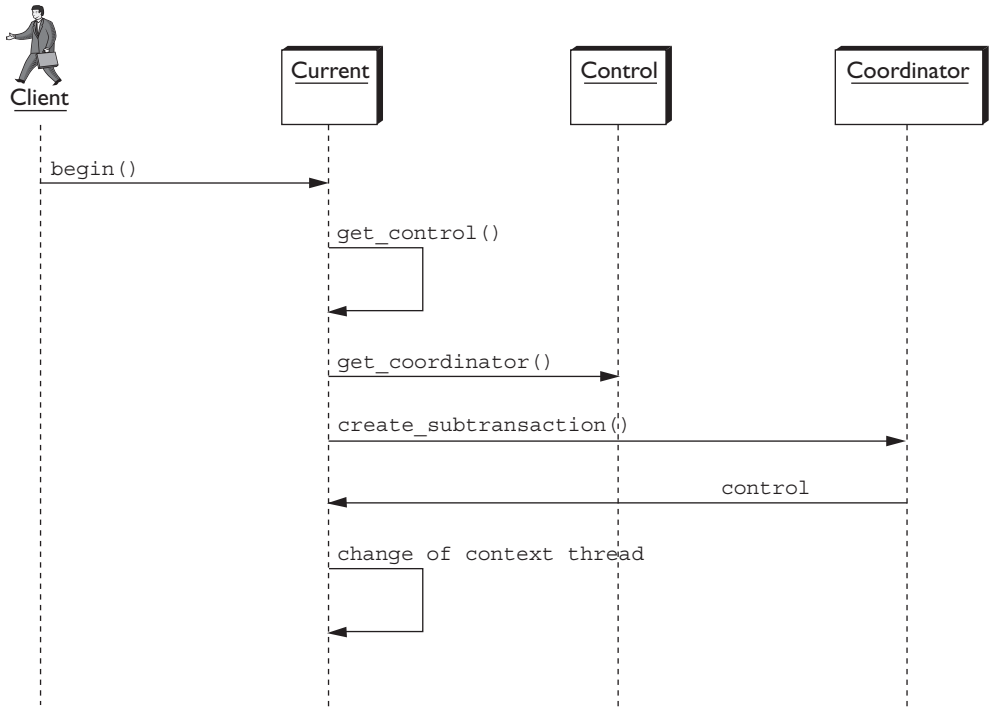

Figure 3-6 Using `Current` to create a nested transaction (subtransaction).

tion does not involve the `TransactionFactory`—the parent transaction (whether it's a top-level transaction or another subtransaction) is responsible for creating and starting the new subtransaction. That's why `Current` gets the `Coordinator` from the current transaction `Control` and then invokes the `create_subtransaction` operation on that.

Participating in an OTS Transaction

We've looked at how transactions can be created, terminated and propagated, but what kind of participants can you register with them? In the following sections we look at the three different types of participants that OTS allows.

Resource

To take part in the two-phase commit protocol, a participant must support the `Resource` interface. Remember, nested transactions in OTS don't use two-phase commit, so any `Resource` will only be driven by the completion of the top-level transaction. As you can see from Code Listing 3-6, the `Resource` supports all of the operations you'd expect. When a `Resource` is registered with a transaction a `RecoveryCoordinator` is returned by the transaction system; as we'll see later, this object can be used to drive failure recovery.

Code Listing 3-6 The `Resource` Interface

```
interface Resource
{
    Vote prepare ();
    void rollback () raises (HeuristicCommit, HeuristicMixed,
                             HeuristicHazard);
    void commit () raises (NotPrepared, HeuristicRollback,
                           HeuristicMixed, HeuristicHazard);
    void commit_one_phase () raises (HeuristicRollback,
HeuristicMixed,
                                     HeuristicHazard);
    void forget ();
}
```

Each `Resource` object is implicitly associated with a single top-level transaction. A given `Resource` should not be registered with the same transaction more than once. This is because when a `Resource` is told to prepare, commit or roll back it must do so on behalf of a specific transaction; however, the `Resource` methods do not specify the transaction identity and neither is the transaction context propagated to the `Resource`. So, the only way a Resource can know which transaction is it being driven by is because it can only be associated with a single transaction.

Transactional objects must register objects that support the `Resource` interface with the current transaction using the `register_resource` method of the transaction's `Coordinator`. An object supporting the `Coordinator` interface will either be passed as a parameter (if explicit propagation is being used) or may be retrieved using operations on `Current` (if implicit propagation is used). If the transaction is a subtransaction, then the `Resource` will not be informed of the subtransaction's completion, and will be registered with its parent upon commit.

This is illustrated in Figure 3-7 where, for simplicity, we assume the hierarchy is only two deep.

As shown in the activity diagram, the client (or server) gets the `Coordinator` for the current transaction and then invokes its `register_resource` operation, passing it the `Resource`. As part of this enlistment process, the transaction will return a reference to a `RecoveryCoordinator` object. We describe what this is for later, but for now all you need to know is that this is a critical component in the way OTS defines failure recovery. The client, or whoever registered the `Resource`, is responsible for ensuring this reference is stored in a durable manner, so that it can be used in the event of a failure. When the client commits the transaction, the coordinator for the nested transaction will propagate participants (`Resources`) to its parent.

A single `Resource` or group of `Resources` is responsible for guaranteeing the ACID properties for the recoverable object they represent. The work `Resources` should perform can be summarized for each phase of the commit protocol:

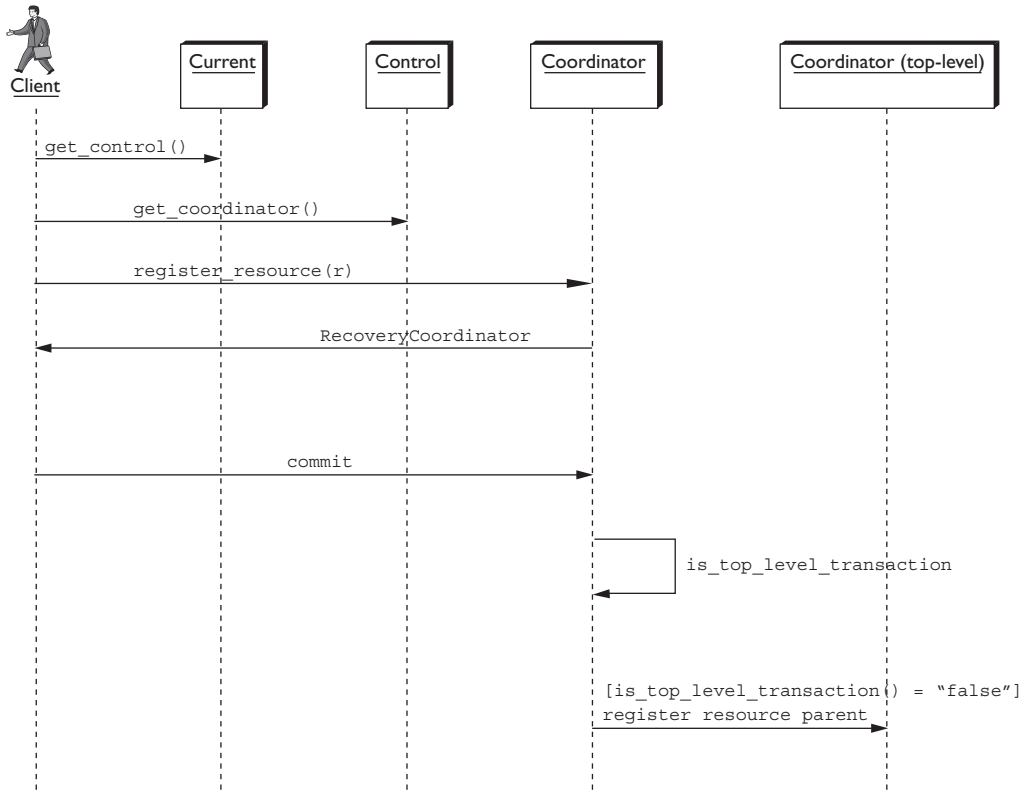

Figure 3-7 Registering a `Resource` with a transaction.

- `prepare`: If no persistent data associated with the resource has been modified within the transaction, then the `Resource` can return `VoteReadOnly` and forget about the transaction; it need not be contacted during the second phase of the commit protocol since it has made no state changes to either commit or roll back. If the resource is able to write (or has already written) all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction, it can return `VoteCommit`. After receiving this response, the transaction service will eventually either commit or roll back. To support recovery, the `Resource` should store the `RecoveryCoordinator` reference in stable storage. The resource can return `VoteRollback` under any circumstances; after returning this response the `Resource` can forget the transaction. Inconsistent outcomes are reported using the `HeuristicMixed` and `HeuristicHazard` exceptions. Heuristic decisions must be made persistent and remembered by the `Resource` until the transaction

coordinator issues the `forget` method; this essentially tells the `Resource` that the heuristic decision has been noted (and possibly resolved).

- `rollback`: This operation can either be invoked after `prepare` (the first phase of the `commit` protocol has failed for some reason and the coordinator is now rolling back), or if the client explicitly rolls back the transaction. If necessary, the resource should undo any changes made as part of the transaction. Heuristic exceptions can be used to report heuristic decisions; but as we've already mentioned, this can only occur if `prepare` had already been received. If a heuristic exception is raised, the resource must remember this outcome until the `forget` operation is performed so that it can return the same outcome in case `rollback` is performed again. Otherwise, the resource can forget the transaction.
- `commit`: If necessary, the resource should commit all changes made as part of this transaction. As with `rollback`, heuristic exceptions can be raised. The `NotPrepared` exception is raised if the `Resource` has not been prepared.
- `commit_one_phase`: This is used when only a single resource is registered with the transaction; the one-phase optimization we discussed in Chapter 1, "Transaction Fundamentals." Since there is only a single participant, the `HeuristicHazard` exception is used to report heuristic decisions related to that resource. This may be important if, for example, the participant is masking multiple subresources from the coordinator (e.g., if interposition is being used).
- `forget`: This operation is performed if the `Resource` raised a heuristic exception. Once the coordinator has determined that the heuristic situation has been addressed, it will issue `forget` on the resource. The resource can then forget all knowledge of the transaction.

What work a participant in the two-phase protocol does when instructed by the coordinator is typically not of interest to the coordinator. It may update a database, modify a file on disk, etc: it depends upon the type of transactional resource it is responsible for manipulating. Some older transaction implementations place restrictions on the types of participants that can be used within the two-phase protocol; for example, as we saw in Chapter 1, "Transaction Fundamentals," in the X/Open DTP standard participants must support the XA protocol, which imposes restrictions on the underlying participant implementations, typically resulting in only databases being used. From the outset, OTS was designed to allow arbitrary participant implementations to be enrolled with transactions. Therefore, the `Resource` interface does not imply or mandate a specific implementation. It is possible to drive XA compliant database if necessary, but it is certainly not a requirement.

SubtransactionAwareResource

Recoverable objects that wish to participate within a nested transaction may support the `SubtransactionAwareResource` interface, which is a specialization of the `Resource` interface.

```
interface SubtransactionAwareResource : Resource
{
    void commit_subtransaction (in Coordinator parent);
    void rollback_subtransaction ();
}
```

Only by registering a `SubtransactionAwareResource` will a recoverable object be informed of the completion of a nested transaction. Generally, a recoverable object will register `Resources` to participate within the completion of top-level transactions and `SubtransactionAwareResources` to be notified of the completion of subtransactions. The `commit_subtransaction` method is passed a reference to the parent transaction in order to allow subtransaction resources to register with these transactions, e.g., to perform propagation of locks.

It is important to remember that `SubtransactionAwareResources` are informed of the completion of a transaction *after* it has terminated, i.e., they cannot affect the outcome of the transaction. It is implementation-specific as to how the OTS implementation will deal with any exceptions raised by `SubtransactionAwareResources`. As we discussed earlier, this can lead to non-atomic outcomes and sensible implementations will always err on the safe side and ensure data and application integrity by forcing the enclosing transaction to roll back.

As we've already mentioned, traditionally nested transactions are terminated using the same two-phase protocol that top-level transaction termination uses. The reason for the difference in OTS is a combination of performance (running a two-phase protocol imposes an overhead) and adoption (most of the companies involved in defining OTS could not support nested transactions, did not intent to support them and so were not concerned with the reliability aspects). As such, the nested transaction support in OTS is unfortunately less than perfect.

A `SubtransactionAwareResource` is registered with a transaction using either the `register_resource` method, or the `register_subtran_aware` method. Both methods have subtly different requirements and effects:

- `register_resource`: If the transaction is a subtransaction then the `Resource` will be informed of its completion and automatically registered with the subtransaction's parent if it commits.
- `register_subtran_aware`: If the transaction is not a subtransaction, then an exception will be thrown. Otherwise, the participant will be informed of the completion of the subtransaction. However, unlike `register_resource`, it will *not* be propagated to the subtransaction's parent if the transaction commits. If the participant requires this it must re-register using the supplied parent parameter.

Both of these registration techniques are illustrated in the following diagrams.

> **NOTE** It's important to understand the differences between these enlistment mechanisms—you might be surprised at the number of OTS implementations (commercial and open source) that get this wrong.

Figure 3-8 shows how a `SubtransactionAwareResource` is registered with a subtransaction using the `register_subtran_aware` method. As you can see in the activity diagram, this is different from the act of registering a `Resource` that we saw earlier: obviously the `register_subtran_aware` operation is invoked on the corresponding `Coordinator`, but when the transaction is committed, its `Terminator` invokes the `commit_subtransaction` operation on all `SubtransactionAwareResources`. A reference to the parent is passed to each such participant in case they wish to register another participant with the parent.

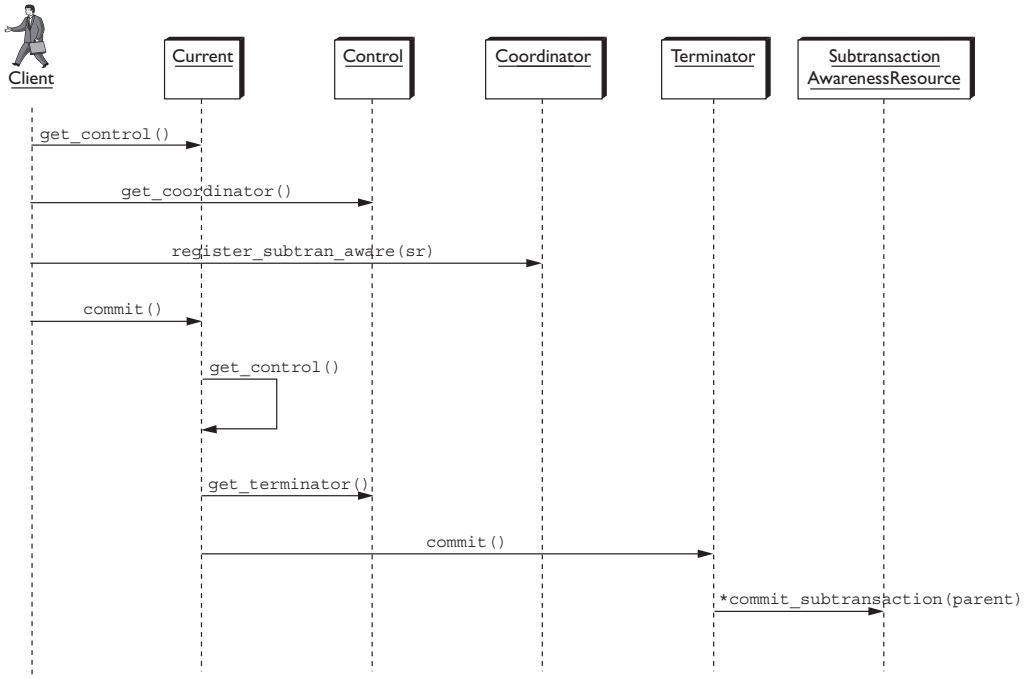

Figure 3-8 Registering a SubtransactionAwareResource with a subtransaction.

Figure 3-9 illustrates the mechanisms involved when a `SubtransactionAwareResource` is registered using the `register_resource` operation. In this case, when the transaction commits, the `Terminator` invokes `commit_subtransaction` on each `SubtransactionAwareResource` and then automatically registers that participant with the parent transaction.

In either case, the participant cannot affect the outcome of the transaction completion. It is only informed of the transaction decision, and should attempt to act accordingly.

Synchronization

In Chapter 1, "Transaction Fundamentals," we saw how many enterprise transaction systems introduced the notion of participants that take part in the transaction protocol before and after the

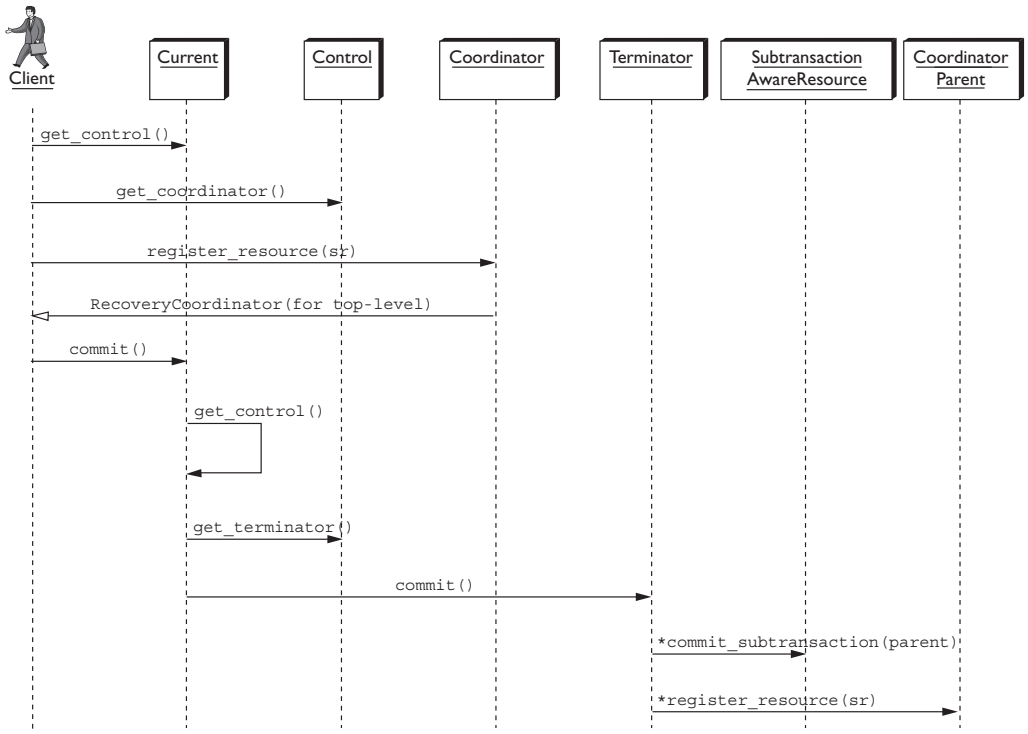

Figure 3-9 Registering a `SubtransactionAwareResource` with a subtransaction as a `Resource`.

two-phase commit protocol: *synchronizations*. The OTS incorporates this notion in the `Synchronizations`, which are enlisted with the transaction using the `register_synchronization` operation of the `Coordinator`. `Synchronizations` are typically employed to flush volatile (cached) state, which may be being used to improve performance of an application, to a recoverable object or database prior to the transaction committing.

> **NOTE** `Synchronizations` can only be associated with top-level transactions and an exception will be raised if an attempt is made to register a `Synchronization` with a subtransaction. Each object supporting the `Synchronization` interface is associated with a single top-level transaction.

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

The method `before_completion` is called prior to the start of the two-phase commit protocol, and `after_completion` is called after the protocol has completed (the final status of the transaction is given as a parameter). If `before_completion` raises an exception, the transaction will be forced to roll back. Any exceptions thrown by `after_completion` will have no effect on the outcome of the transaction.

Participant Relationships

Given the previous description of `Control`, `Resource`, `SubtransactionAwareResource`, and `Synchronization`, the UML relationship diagram shown in Figure 3-10 can be drawn:

What this shows is that each transaction `Control` can have any number of `Synchronizations`, `Resources` or `SubtransactionAwareResources` registered with it. However, each of those resources can only be associated with a single transaction.

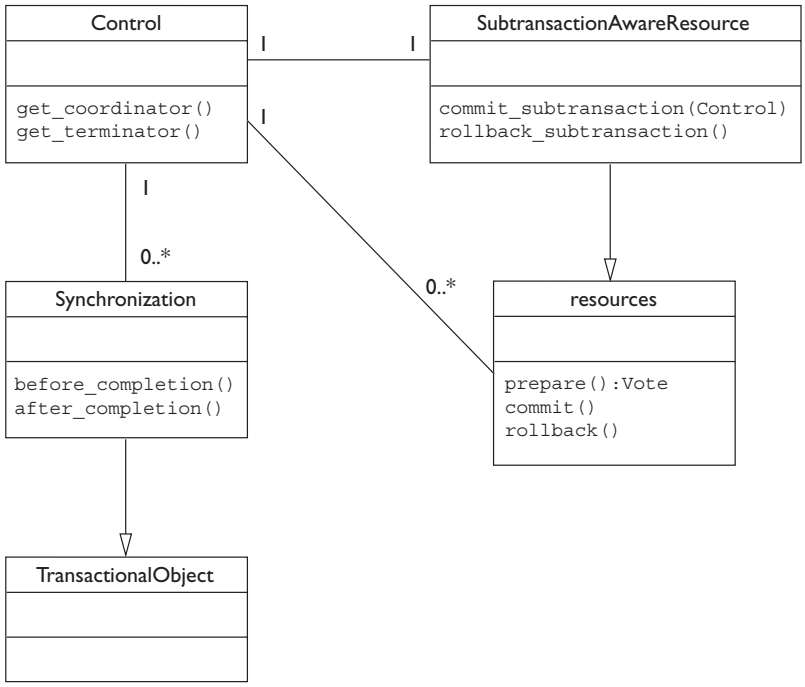

Figure 3-10 The relationships among the various OTS interfaces.

Transactions and Registered Resources

In this section we look at some example interactions between transactions and various types of participants. We also use UML activity diagrams to try to better illustrate exactly what is going on and what we've just described.

Figure 3-11 shows the course of events when committing a subtransaction that has both `Resource` and `SubtransactionAwareResource` objects registered with it; we assume that the `SubtransactionAwareResources` were registered using `register_subtran_aware` so they won't be automatically registered with the parent transaction when it commits. Remember that the `Resources` are not informed of the termination of the subtransaction, whereas the `SubtransactionAwareResources` are. However, only the `Resources` are automatically propagated to the parent transaction.

It should be relatively easy to see what happens when the subtransaction rolls back. Any registered `Resources` are discarded and `SubtransactionAwareResources` are informed of the transaction outcome.

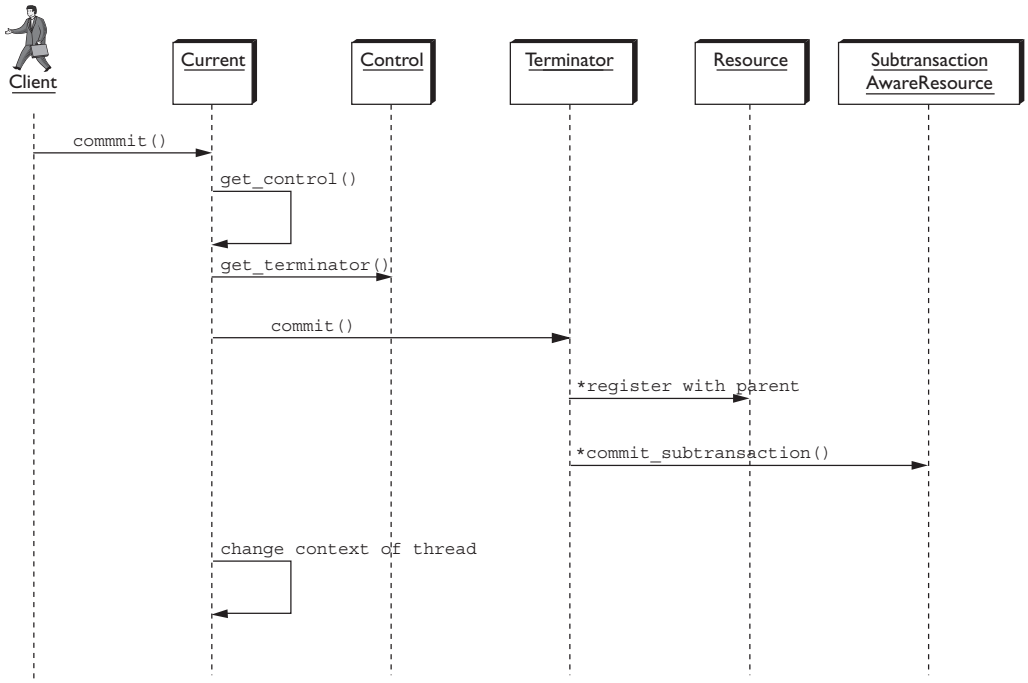

Figure 3-11 Committing a subtransaction.

Figure 3-12 shows the activity diagram for committing a top-level transaction; those subtransactions within the top-level transaction that have also successfully committed will have propagated `SubtransactionAwareResources` to the top-level transaction, and these will then participate within the two-phase commit protocol. As can be seen, however, prior to `prepare` being called, any registered `Synchronizations` are first contacted. Because we are using indirect context management, when the transaction commits, the transaction service changes the invoking thread's transaction context.

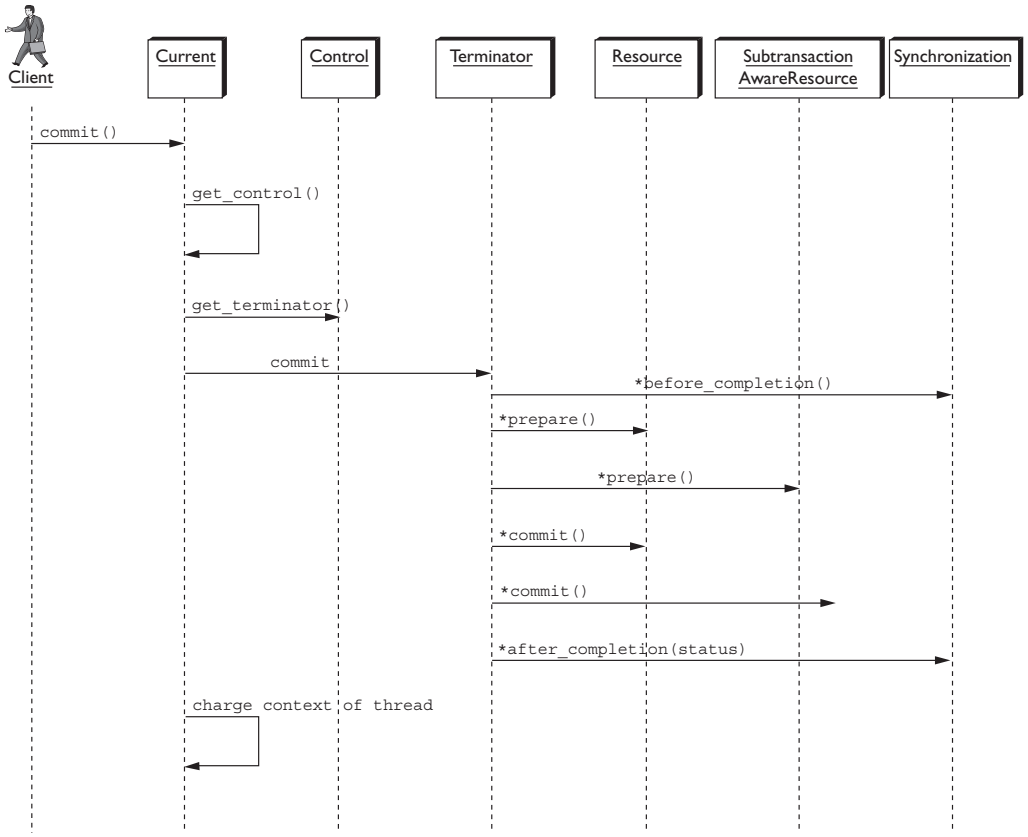

Figure 3-12 Top-level transaction commit.

The `RecoveryCoordinator`

We saw in Chapter 1, "Transaction Fundamentals" how failure recovery is extremely important to transaction systems (and distributed systems in general) and also how it is typically overlooked or neglected by most public domain and some commercial implementations. In the OTS specification recovery is described in some detail and although not optional in the way nested transactions are, you'll still find supposedly "OTS compliant" implementations that don't implement it. Beware!

In order to drive recovery, a participant needs a reference to the transaction (or transaction system) it is registered with. You might think that the `Coordinator` would play that role but it can't for a number of reasons including:

- The CORBA object that supports the `Coordinator` interface doesn't need to be durable; the entity that the `Coordinator` represents may well be durable, however. So, a reference to the `Coordinator` doesn't have to remain valid after recovery has occurred and in most cases it won't be valid.
- We saw in Chapter 1, "Transaction Fundamentals," how a transaction coordinator maintains an intentions list of participants that is made durable after the `prepare` phase succeeds so that failure recovery at the coordinator can be driven if necessary. If a participant fails and recovers and then enquires about the transaction status, the transaction coordinator should be able to tie up the enquiry with the specific participant so that it can begin to prune the intentions list. You might think that the `Resource` could be passed as a parameter (e.g., to `get_status`). Unfortunately, due to the CORBA architecture the same object may be represented by multiple interfaces and there is no guaranteed equality operator in CORBA. The only sure way to do this when OTS was originally defined is to tie a specific participant to a specific coordinator interface: then the coordinator knows implicitly which participant is enquiring about the status since only one participant can use that interface. Unfortunately, the general `Coordinator` interface does not have this property, since the same interface can be propagated to any number of participants.

As a result, the OTS specification introduced a very special interface on the transaction coordinator: the `RecoveryCoordinator`. A reference to a `RecoveryCoordinator` is returned as a result of successfully calling `register_resource` on the transaction `Coordinator`. This object is recoverable so that it remains valid after the transaction it represents fails and recovers and is implicitly associated with a single `Resource`. It is intended to be used to drive the `Resource` through recovery procedures in the event of a failure occurring during the transaction.

As you can see from Figure 3-13, the `RecoveryCoordinator` has a single method `replay_completion` that takes a reference to a `Resource` and returns the current transaction status. The operation is supposed to be non-blocking, which means that although the reception of `replay_completion` may cause the transaction coordinator to start recovery procedures, it must return the status immediately and do any other work afterwards. Each `Resource` is tied to a specific `RecoveryCoordinator`.

Given that each `RecoveryCoordinator` is tied to a specific `Resource`, you may be wondering why `replay_completion` takes a `Resource` parameter. The reason is that participants need not recover at the same location making the original corresponding `Resource` invalid; the coordinator can thus use the parameter provided to redirect invocations meant for the old `Resource`.

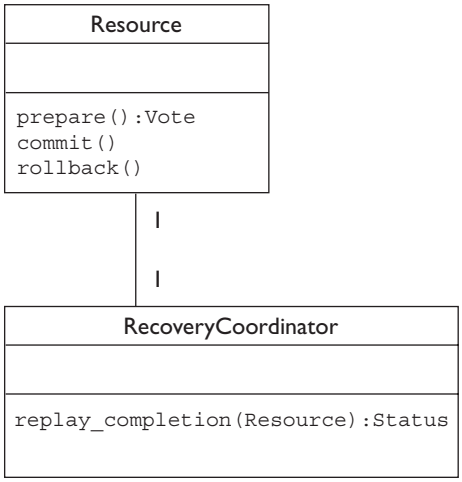

Figure 3-13 Resource and RecoveryCoordinator relationship.

Interposition

In Chapter 1, "Transaction Fundamentals," we saw how interposition of subordinate coordinators can be used in distributed transaction systems to improve performance. OTS objects supporting the interfaces such as the `Control` interface are simply standard CORBA objects. This implies that when an interface is passed as a parameter in some operation call to a remote server only an object reference is passed. This ensures that any operations that the remote server performs on the interface are correctly performed on the real object. However, this can have substantial penalties for the application due to the overheads of remote invocation.

To avoid this overhead, an implementation of OTS *may* support interposition. This permits a server to create a local control object, which acts as a local coordinator fielding registration requests that would normally have been passed back to the originator. This surrogate must register itself with the original coordinator to enable it to correctly participate in the commit protocol.

Unfortunately as with nested transactions, interposition isn't mandated by the specification. You may need to check with your friendly OTS vendor to determine whether or not performance optimizations are possible when using distributed transactions.

As we mentioned earlier, the `PropagationContext` contains the timeout for the transaction. An interposed coordinator can use this value to optimize the termination protocol and (essentially) start a local timer so that if it goes off, the interposed coordinator can roll back its portion of the transaction hierarchy. Some transaction service implementations use this fact to optimize the entire transaction termination protocol: the root coordinator may not even send a roll back message since it knows that subordinates have already rolled back (e.g., the Hewlett-Packard Transaction Service (now marketed by Arjuna Technologies).

Shared and Unshared Transactions

The introduction of asynchronous messaging into CORBA was a relatively late development and required a new form of transaction model similar to that we considered in Chapter 1, "Transaction Fundamentals," in the section on Transactions and Queues. The model supported by OTS specification 1.0, 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 was introduced (the *unshared transaction model*) in version 1.2, which uses a store and forward transport between the client and server. In this new model, the communication between client and server is broken into separate requests, separated by a reliable transmission between routers (queues). When transactions are used, this model uses multiple shared transactions, each executed to completion before the next one begins.

Checked Transaction Behavior

In Chapter 1, "Transaction Fundamentals," we saw how in a distributed, multi-threaded environment checked transactions are essential to ensure integrity of your applications. So it shouldn't come as a surprise to learn that OTS supports both checked and unchecked transaction behavior. If you remember, checked transactions have a number of integrity constraints including:

- Ensuring that a transaction will not commit until all transactional objects involved in the transaction have completed their transactional requests.
- Ensuring that only the transaction originator can commit the transaction.

Checked behavior is only possible if implicit propagation is used since the use of explicit propagation prevents OTS from tracking which objects are involved in the transaction with any certainty: the propagation of the transaction context happens at the application level out of the control of the transaction service to monitor.

In contrast, unchecked behavior allows relaxed models of atomicity to be implemented. Any use of explicit propagation implies the possibility of unchecked behavior since it is the application programmer's responsibility to ensure the correct behavior.

There are many possible implementations of checking in a transaction service. One provides equivalent function to that provided by the request/response inter-process communication models defined by X/Open. The X/Open transaction service model of checking is particularly important because it is widely implemented and this is the one favored by OTS. In X/Open, completion of the processing of a request means that the object has completed execution of its method and replied to the request. X/Open DTP Transaction Managers are examples of transaction management functions that implement checked transaction behavior.

When implicit propagation is used, the objects involved in a transaction at any given time may be represented as a tree: the request tree for the transaction. The beginner of the transaction is the root of the tree. Requests add nodes to the tree; replies remove the replying node from the

tree. Synchronous requests, or the checks described below for deferred synchronous requests, ensure that the tree collapses to a single node before commit is issued.

Applications that use synchronous requests implicitly exhibit checked behavior. For applications that use deferred synchronous requests, in a transaction where all clients and objects are in the domain of a checking transaction service, the transaction service can enforce this property by applying a *reply check* and a *commit check*. The transaction service must also apply a *resume check* to ensure that the transaction is only resumed by application programs in the correct part of the request tree.

1. *Reply check*: Before allowing an object to reply to a transactional request, a check is made to ensure that the object has received replies to all its deferred synchronous requests that propagated the transaction in the original request. If this condition is not met, an exception is raised and the transaction is marked as rollback only, that is, it cannot be successfully committed. A transaction service may check that a reply is issued within the context of the transaction associated with the request.
2. *Commit check*: Before allowing commit to proceed, a check is made to ensure that (1) the commit request for the transaction is being issued from the same execution environment that created the transaction, and (2) the client issuing commit has received replies to all the deferred synchronous requests it made that caused the propagation of the transaction.
3. *Resume check*: Before allowing a client or object to associate a transaction context with its thread of control, a check is made to ensure that this transaction context was previously associated with the execution environment of the thread. This would be true if the thread either created the transaction or received it in a transactional operation.

Transaction Interoperability

We mentioned at the start of this chapter that the original intention of the OTS specification was that it would be the next generation of transaction service standard. However, interoperability with existing implementations and standards was a major requirement. As such, OTS defines possible interactions with the following:

- X/Open TX interface for transaction demarcation. There is a lot of overlap between these interfaces and the OTS `Current` interface.
- X/Open XA interface for resource manager interactions. Given the discussions in this chapter and Chapter 2, you might be able to see the possible mappings between the XA resource manager and the OTS `Resource` interface. However, full XA interoperability for OTS goes beyond this mapping.
- OSI TP protocol is the transaction protocol defined by ISO and selected by X/Open to allow the distribution of transactions by one of the communication interfaces (remote procedure call, client-server, or peer-to-peer).

- SNA LU 6.2 protocol is a transactional protocol defined by IBM and widely used for transaction distribution.
- ODMG standard is defined by the Object Database Management Group and described a portable interface to access Object Database Management Systems.

It's outside the scope of this book to describe how interoperability between these various protocols and OTS is achieved, but it is important that you realize it is possible. Although interoperability between OTS implementations may still be less than ideal, interoperability with protocols such as those mentioned above has always had a higher priority. The sheer number of transaction systems and applications that use one or more of these protocols made this an obvious necessity.

However, although interoperability with these protocols is possible, it is not a mandated part of the OTS specification. Therefore, a compliant OTS implementation need not provide interoperability at all, or may only be interoperable with a subset of the protocols. It is worth noting that all of the commercial implementations of OTS that we know about support interoperability with some of these protocols (usually at least X/Open).

Writing Applications Using OTS Interfaces

So far we've looked at how the OTS specification supports the transaction fundamentals we discussed earlier in this book. However, we haven't seen how you could actually use the specification to implement a transactional application. In the following sections we look at this aspect.

The first things to note are that to develop a transactional application that has transactional clients and objects, a programmer must be concerned with:

- Creating `Resource` and `SubtransactionAwareResource` objects for each object that will participate within the transaction/subtransaction. These resources are responsible for the persistence, concurrency control and recovery for the object. The OTS will invoke these objects during the prepare/commit/rollback phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.
- Registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.
- Ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions is correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.
- In the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction using the `RecoveryCoordinator`.

- The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. The interfaces defined within the OTS specification are too low-level for most application programmers.

As you can see, there's quite a lot that involved and most of it is error-prone, which is one reason why most implementations provide high-level interfaces to abstract away the complexity. The most common high-level API is the Java Transaction API that we saw in Chapter 2. However, this can have its disadvantages, such as the fact that your participants are limited to being XA-aware. So, just in case you do want to program at the level of OTS, we look at the issues involved in more detail in the following sections.

Transaction Context Management

If implicit transaction propagation is being used, the programmer should ensure that appropriate objects are located within a POA with the correct transaction policies specified; otherwise, the transaction contexts must be explicitly passed as parameters to the relevant operations.

A Transaction Originator: Indirect Context Management and Implicit Propagation

In the code fragments below, a transaction originator uses indirect context management and implicit transaction propagation; `txn_crt` is the `Current` interface; the client uses the `begin` operation to start the transaction which becomes implicitly associated with the originator's thread:

```
. . .
txn_crt.begin();
// should test the exceptions that might be raised
...
// the client issues requests, some of which involve
// transactional objects;
BankAccount1.makeDeposit(deposit);
. . .
```

The program commits the transaction associated with the client thread. The `report_heuristics` argument is set to `false` so no report will be made by the transaction service about possible heuristic decisions; not necessarily a great idea, but all right for the purposes of our example.

```
. . .
txn_crt.commit(false);
. . .
```

Transaction Originator: Direct Context Management and Explicit Propagation

In the following example, a transaction originator uses direct context management and explicit transaction propagation. The client uses a transaction factory to create a new transaction and uses the returned `Control` object to retrieve the `Terminator` and `Coordinator` objects.

```
. . .
org.omg.CosTransactions.Control c;
org.omg.CosTransactions.Terminator t;
org.omg.CosTransactions.Coordinator co;
org.omg.CosTransactions.PropagationContext pgtx;

c = TFactory.create(0);
t = c.get_terminator();
pgtx = c.get_coordinator().get_txcontext();
. . .
```

The client then issues requests, some of which involve transactional objects. When invoking transactional objects the client explicitly propagates the context as an extra parameter in each operation. The `Control` object reference is passed as an explicit parameter of the request:

```
transactional_object.do_operation(arg, pgtx);
```

The transaction originator uses the `Terminator` object to commit the transaction; the `report_heuristics` argument is again set to `false`:

```
t.commit(false);
```

Implementing a Recoverable Server

Hopefully by now you have some idea of what a recoverable server is, but let's spend a bit of time just going over what we mean by this term. In OTS a recoverable server is essentially a deployment environment for at least one transactional object and one `Resource` (participant). For example, if you remember the bank account example we first examined back in Chapter 1, "Transaction Fundamentals," we might imagine a recoverable server for each bank account, as shown in Figure 3-14.

In that recoverable server we assume that there is a transactional object representing the actual account and containing operations to add, remove and inspect operations (there could well be others, but those will be sufficient for this example). Whenever an operation is performed on the account object within the scope of a transaction (e.g., removing money), the transactional object will be responsible for registering a participant with the transaction so that it can ultimately control the outcome of the work. In OTS terms, this participant is a `Resource`.

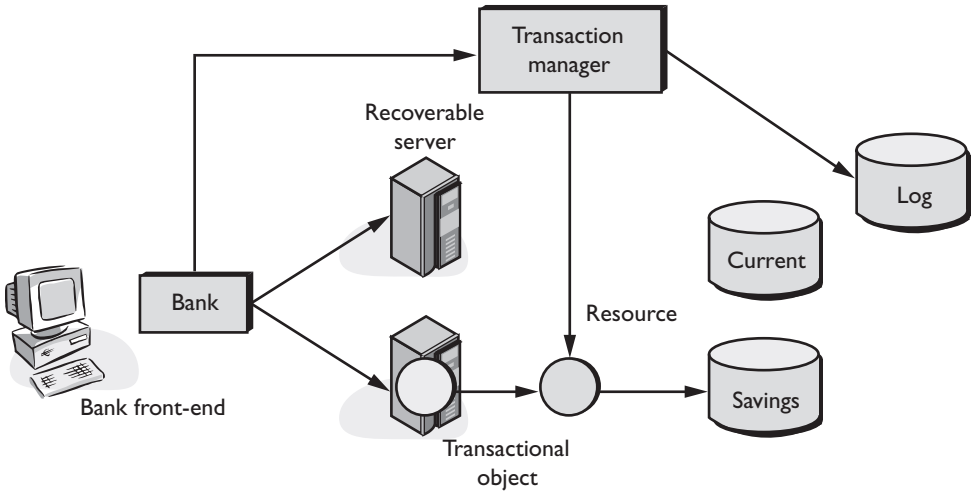

Figure 3-14 Recoverable servers and the bank account.

Hopefully, you can now see that the concept of a recoverable server is fundamental to transactional applications. Whether you know it or not, if you've written server-side objects that use transactions, such as EJBs residing in an application server, this combination (of application server and EJB) is the equivalent of the OTS recoverable server. As we've seen in earlier chapters, if that EJB uses a JDBC driver, the application server will be getting `XAResources` and registering them with the transaction at appropriate times: the equivalent of the OTS `Resource`. So, with that in mind, let's look at the responsibilities of each of these objects in more detail in the following sections.

Transactional Object

The responsibilities of the transactional object are to implement the object's operations and to register a `Resource` object with the `Coordinator` so commitment of the resources, including any necessary recovery, can be completed.

Remember that the `Resource` object identifies the involvement of the recoverable server in a particular transaction so it can only be registered in one transaction at a time. A different resource object *must* be registered for each transaction in which the server is involved. Because a transactional object may receive multiple requests within the scope of a single transaction it should only register its involvement once per transaction. In order to ensure that this is the case, you can use the `is_same_transaction` operation of the `Coordinator` to do transaction comparison. The `hash_transaction` operation allows you to reduce the number of transaction comparisons it has to make: the `is_same_transaction` operation need only be done on `Coordinators` which have the same hash code as the `Coordinator` of the current request.

Resource Object

We saw earlier that the responsibilities of a `Resource` object are to participate in the completion of the transaction, to update the recoverable server's resources in accordance with the transaction outcome, and ensure termination of the transaction, including across failures.

> **NOTE** In the following examples of transactional objects some form of concurrency control implementation will be required to ensure isolation. Typically this will depend upon the deployment environment and is beyond the scope of OTS to mandate. As such, we simply assume concurrency control is available.

An Example of a Recoverable Server

In this example, `BankAccount1` is an object that is directly responsible for its own state, so it is a transactional object and explicitly uses a `Resource`. It inherits from both the `TransactionalObject` and the `Resource` interfaces (for the purposes of this example it is easier to revert to the 1.1 way of doing things rather than show you how to initialize a POA):

```
interface BankAccount1 : CosTransactions::TransactionalObject,
CosTransactions::Resource
{
    ...
    void makeDeposit (in float amt);
    ...
};
```

The corresponding Java class is:

```
public class BankAccount1 extends
org.omg.CosTransactions.TransactionalObjectImplBase, extends
org.omg.CosTransactions.ResourceImplBase
{
public void makeDeposit(float amt);
public float get_balance ();
public void set_balance (float amt);
    ...
};
```

When an invocation on `makeDeposit` is made, the context of the transaction is implicitly associated with the thread doing the work by virtue of the fact that the `BankAccount1` interface is derived from the `CosTransactions::TransactionalObject` interface. From the previous discussion of `TransactionalObject`, you'll remember that this is works as a flag to the transaction service and ORB to ensure that any transaction context present at the client is implicitly propagated to the server and made available to the object doing the work (via `Current`). You don't have to worry about how this happens—that's the responsibility of the

OTS implementation you're using to figure out. All you have to know is that it will happen and you can program accordingly.

For those readers who are interested in how this context propagation and thread association occurs, we give a brief overview. Anyone who's not interested can skip the next paragraph.

When a thread with a transaction associated with it makes a call on a remote object, the transaction service and ORB cooperate to ensure that the transaction context is propagated with the invocation. At the receiver, this additional context information is plucked off the incoming invocation and associated with the thread that will do the work—which, just to add more complexity, may not actually be the thread that received the invocation! This transaction association occurs in collaboration with `Current` such that if that thread were then to use `Current` to get the transaction, `Current` would return a reference to the imported transaction (or to a local subordinate coordinator if interposition is being used).

`Current` is used to retrieve the `Coordinator` object associated with the transaction. In this example, the variable `txn_crt` is assumed to point to an instance of `Current` – we won't show how this variable is set up simply because that can depend upon the implementation of the transaction service and the version of the OTS specification you're using.

```
void makeDeposit (float amt)
{
    org.omg.CosTransactions.Control c;
    org.omg.CosTransactions.Coordinator co;
    c = txn_crt.get_control();
    co = c.get_coordinator();
    . . .
```

As we saw earlier, because `Current` was developed with client interactions in mind, it is not suitable for all server-side requirements. That's the reason that the above code fragment has to use `Current` to get the `Coordinator` via the `Control` so that it can then register the `Resource`.

NOTE In this example the object registers itself as a `Resource`. This imposes the restriction that the object may only be involved in one transaction at a time. For obvious reasons this is not the recommended way for recoverable objects to participate within transactions and is only used as an example. If more parallelism is required, separate resource objects should be registered for involvement in the same transaction.

```
    RecoveryCoordinator r;
    r = co.register_resource(this);

    // get a WRITE lock on the object state; only proceed if
```

```
// that lock is granted.

    balance = balance + f;
    num_transactions++;
    ...
    // end of transactional operation
};
```

It's typical to register the `Resource` before any work is performed by the transactional object. The reason for this is that the resource may need to perform some initialization on the state that the transactional object is about to manipulate. For example, if the state is obtained from a database, then the `Resource` implementation may need to associate the transaction with the database connection prior to it being used.

What we haven't shown in the above fragment is what happens with the `RecoveryCoordinator` returned by the `Coordinator` when the object registers the `Resource`. As has been mentioned previously, the `RecoveryCoordinator` is the hook into the transaction recovery subsystem that the recovering object has (recovering `BankAccount1` object in our case). So, making sure this hook (reference) is saved away successfully is very important, because without it, recovery may take a long time to complete or in some circumstances may never be able to happen.

We now show the other two important operations on the bank account. They are surprisingly similar to `makeDeposit`.

```
float get_balance ()
{
    org.omg.CosTransactions.Control c;
    org.omg.CosTransactions.Coordinator co;
    c = txn_crt.get_control();
    co = c.get_coordinator();

    RecoveryCoordinator r = co.register_resource(this);

    // get a READ lock on the object state; only proceed if
// that lock is granted.

    num_transactions++;

return balance;    // end of transactional operation
};

void set_balance (float amt)
{
    org.omg.CosTransactions.Control c;
    org.omg.CosTransactions.Coordinator co;
    c = txn_crt.get_control();
    co = c.get_coordinator();
```

```
    RecoveryCoordinator r = co.register_resource(this);

    // get a WRITE lock on the object state; only proceed if
// that lock is granted.

    num_transactions++;

balance = amt;    // end of transactional operation
};
```

Example of a Transactional Object

In this example, `BankAccount2` is an object that delegates all state manipulation and control to other objects (perhaps implementations of `BankAccount1`). Therefore, it's not a recoverable server, but simply a "pure" transactional object (it does not use `Resources`). However, in order to ensure that any transaction context present on client invocations is implicitly propagated, the `BankAccount2` interface derives from `TransactionalObject` interface:

```
interface BankAccount2 : CosTransactions::TransactionalObject
{
    ...
    void makeDeposit(in float amt);
    ...
};

public class BankAccount2 extends
org.omg.CosTransactions.TransactionalObjectImplBase
{
public void makeDeposit(float amt);
    ...

private BankAccount1 res1;
private BankAccount1 res2;
};
```

The `makeDeposit` operation performs some transactional requests on external, recoverable servers which, as you can see above, we assume are implementations of `BankAccount1`. The objects `res1` and `res2` are recoverable objects, but the `BankAccount2` object isn't—it does not maintain any state that is affected by the transaction, so it doesn't need to save or restore anything in the event of a failure (or recovery). The current transaction context is implicitly propagated to these other recoverable objects because we remembered to make sure that `BankAccount1` derives from `TransactionalObject`.

```
void makeDeposit (float amt)
{
    balance = res1.get_balance(amt);
    balance = balance + amt;
    res1.set_balance(balance);
    res2.increment_num_transactions();
} // end of transactional operation
```

So the `makeDeposit` implementation looks a lot simpler in this case than previously. However, if you think about it, all we've done is passed the responsibility of recoverability to something else (in our case `BankAccount1`). Someone must be responsible for ensuring `Resources` are enlisted with transactions at the appropriate time, `RecoveryCoordinators` are saved away and that recovery procedures are instigated when required. There is no such thing as a free lunch. However, what this example does show is that it is entirely possible to construct one transactional application using objects from another.

Worked Example

The following example illustrates the concepts and the implementation details for a simple client/server example that uses implicit context propagation and indirect context management. It is relatively simplistic in that only a single unit of work is included within the scope of the transaction; consequently, a two phase commit is not required, but rather a one phase commit.

The IDL interface for this example is as follows. For the purposes of this worked example, we have defined a single method (see line 9 in Code Listing 3-7) for the `DemoInterface` interface. We use this method in the `DemoClient` program.

Code Listing 3-7 The Demo IDL

```
1    #include <idl/CosTransactions.idl>
2    #pragma javaPackage ""
3
4
5    module Demo
6    {
7        exception DemoException {};
8
9        interface DemoInterface :
CosTransactions::TransactionalObject
10       {
11           void work() raises (DemoException);
12       };
13   };
```

Resource

Here, we have overridden the methods of the `Resource` implementation class; the `DemoResource` implementation includes the placement of `System.out.println` statements at judicious points, to highlight when a particular method has been invoked.

As mentioned previously, only a single unit of work is included within the scope of the transaction; consequently, we should not expect the `prepare()` at line 6 in Code Listing 3-8, or the `commit()` at line 20 to be invoked. However, we should expect the `commit_one_phase()` method at line 26 to be called.

Code Listing 3-8 Example OTS Resource implementation

```
1    import org.omg.CosTransactions.*;
2    import org.omg.CORBA.SystemException;
3
4    public class DemoResource extends  org.omg.CosTransactions
._ResourceImplBase
5    {
6        public Vote prepare() throws HeuristicMixed, HeuristicHazard,
7        SystemException
8        {
9            System.out.println("prepare called");
10
11           return Vote.VoteCommit;
12       }
13
14       public void rollback() throws HeuristicCommit, HeuristicMixed,
15       HeuristicHazard, SystemException
16       {
17           System.out.println("rollback called");
18       }
19
20       public void commit() throws NotPrepared, HeuristicRollback,
21       HeuristicMixed, HeuristicHazard, SystemException
22       {
23           System.out.println("commit called");
24       }
25
26       public void commit_one_phase() throws HeuristicHazard,
SystemException
27       {
28           System.out.println("commit_one_phase called");
29       }
30
31       public void forget() throws SystemException
32       {
33           System.out.println("forget called");
34       }
35   }
```

Transactional Implementation

At this stage, let's assume that the `Demo.idl` has been processed by the ORB's idl compiler to generate the necessary client/server package.

Line 13 in Code Listing 3-9 returns the transactional context for the `Current` object. Once we have a `Control` object, we can derive the `Coordinator` object (line 15).

Line 16 creates a resource for the transaction.

Line 18 uses the `Coordinator` to register a `DemoResource` object as a participant in the transaction. When the transaction is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction.

Code Listing 3-9 Example transactional object implementation

```
1    import Demo.*;
2    import org.omg.CosTransactions.*;
3    import org.omg.*;
4
5
6    public class DemoImplementation extends
Demo._DemoInterfaceImplBase
7    {
8        public void work() throws DemoException
9        {
10           try
11           {
12
13               Control control = get_current().get_control();
14
15               Coordinator  coordinator = control.get_coordinator();
16               DemoResource resource    = new DemoResource();
17
18               coordinator.register_resource(resource);
19
20           }
21           catch (Exception e)
22           {
23               throw new DemoException();
24           }
25       }
26
27   }
```

Server Implementation

It is the servant class `DemoImplementation` that contains the implementation code for the `DemoInterface` interface. Furthermore, it is ultimately the responsibility of the servant to

service a particular client request. Line 13 in Code Listing 3-10 instantiates a transactional object (servant) for the subsequent servicing of client requests.

Lines 15 through to 19 take the servant and obtain a CORBA reference (IOR) for it. This is a string representation of the reference that can be passed between users via a file or name service, for example, and allow multiple users to share the same object implementation. In our example, this IOR is written to a temporary file. This IOR will be subsequently used to construct the object in the `DemoClient` program.

Finally, line 23 places the server process into a state where it can begin to accept requests from client processes.

Code Listing 3-10 Example server implementation

```
1    import java.io.*;
2    import com.arjuna.OrbCommon.*;
3
4    public class DemoServer
5    {
6        public static void main (String[] args)
7        {
8            try
9            {
10               // initialise the ORB using args from command line
11               // initialise the POA
12
13               DemoImplementation obj = new DemoImplementation();
14
15               String ref  = orb.object_to_string(obj);
16               BufferedWriter file =
17               new BufferedWriter(new
FileWriter("DemoObjReference.tmp"));
18               file.write(ref);
19               file.close();
20
21               System.out.println("Object reference written to
file");
22
23               for (;;);
24           }
25           catch (Exception e)
26           {
27               System.err.println(e);
28           }
29       }
30   }
```

Client Implementation

Once a server process has been started and it has written the IOR of the servant to a temporary (and shareable) file, our client can read that IOR (lines 17 to 22 in Code Listing 3-11).

Once we have the IOR, we can reconstruct the servant object. Initially, this string to object conversion returns an instance of `org.omg.CORBA.Object` (see line 24). However, if we want to invoke a method on the servant object, it is necessary for us to narrow this instance to an instance of the `DemoInterface` interface (line 26).

Once we have a reference to this servant object, we can start a transaction (line 28), perform a unit of work (line 30) and commit the transaction (line 32).

Code Listing 3-11 Example client implementation

```
1    import Demo.*;
2    import java.io.*;
3    import org.omg.*;
4    import org.omg.CosTransactions.*;
5
6
7
8    public class DemoClient
9    {
10       public static void main(String[] args)
11       {
12           try
13           {
14               // initialise the ORB using args
15               // initialise the root POA
16
17               String ref = new String();
18               BufferedReader File =
19               new BufferedReader(new
FileReader("DemoObjReference.tmp"));
20
21               ref = file.readLine();
22               File.close();
23
24               org.omg.CORBA.Object obj =
25               orb.string_to_object(ref);
26               DemoInterface d = (DemoInterface)
DemoInterfaceHelper.narrow(obj);
27
28               get_current().begin();
29
30               d.work();
31
32               get_current().commit(true);
33           }
34           catch (Exception e)
```

```
35            {
36                System.err.println(e);
37            }
38        }
39    }
```

Sequence Diagram

The activity diagram in Figure 3-15 illustrates the method invocations that occur between the client and server.

The following aspects are worthy of further discussion:

1. The transactional context does not need to be explicitly passed as a parameter (as we are using implicit context propagation since `DemoInterface` is a `TransactionalObject`) in the `work()` method.
2. We assume the use of interposition when the client and server processes are started. The interposed coordinator is automatically registered with the root coordinator at the client (indicated by the `Control` object at the client).

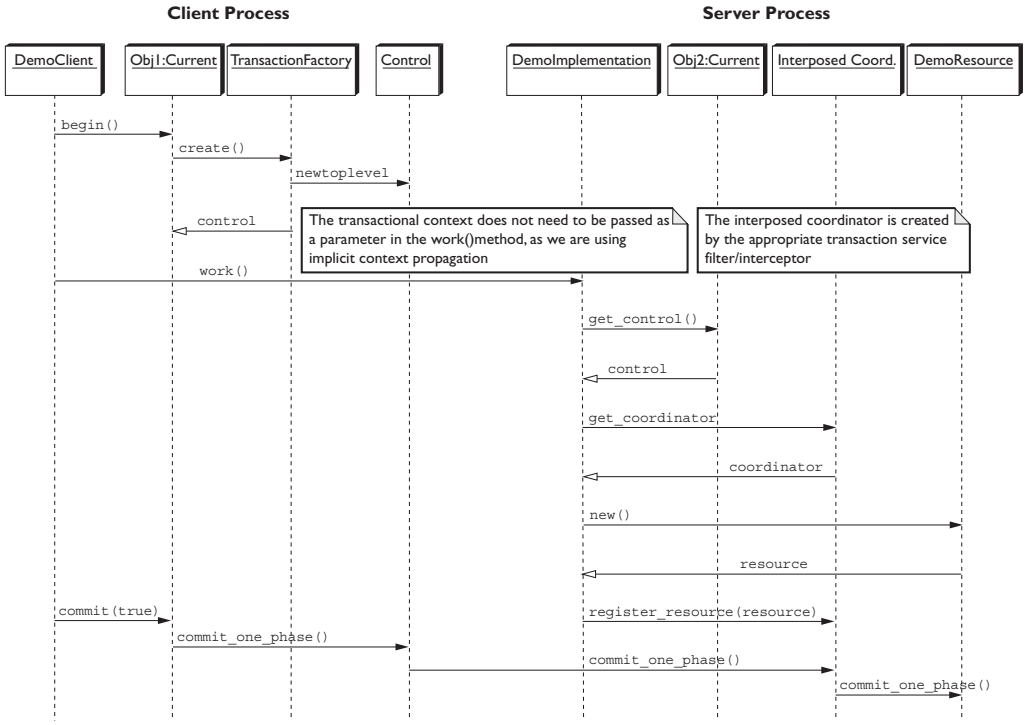

Figure 3-15 Example activity diagram.

3. The resource that is responsible for committing or rolling back modifications made to the transactional object is associated (“registered”) with the interposed coordinator.

The `commit()` invocation in the client process calls the root coordinator. The root coordinator calls the interposed coordinator, which in turn calls the `commit_one_phase()` method for the resource.

Choosing an OTS Implementation

We’ve looked at what’s involved in providing an implementation of the OMG’s OTS specification. In this section, we try to distill this information and our own experiences into some important questions to consider and ask if you are looking at an OTS implementation. This list is by no means exhaustive and you may come up with your own questions after re-reading this chapter.

- Has the implementation been written from scratch, and if so, what are the credentials of the implementers? This may seem like a strange question, but let’s look at an analogy. If you were going to buy a car would you get one from a reputable dealer or from someone who’s only just started in the business? Now the answer is not straightforward, as it will depend upon a number of factors. But, if you do go to the newbie car manufacturer, you’d be sensible to determine before hand what their track record is in the area of building cars. Unfortunately, this analogy doesn’t quite hold up because there are laws governing who can build and market cars and all cars have to go through stringent safety tests. The same cannot be said of OTS implementations. So, doing your homework beforehand is your first and best line of defense against shoddy implementations.
- Has the implementation been deployed elsewhere? Although there has always got to be a first customer for an OTS, it’s worth asking whether or not the implementation has been used before and if this was successful. Systems that have been used before and have survived are often more reliable than those that have only been tested by the developers.
- Which version of the specification does it support? As we said at the start of this chapter, most implementations are OTS 1.1 compliant, but the current version of the specification is 1.4. Backward support for interoperability between 1.4 and 1.1 is available, but upward interoperability is not possible.
- How compliant is it with the specification? It may be surprising to learn that there are implementations of OTS that do not support failure recovery, or only support it in very limited situations.
- Does it support the optional parts of the specification, such as nested transactions and interposition? Although you may never need nested transactions, interposition can be extremely important in any large-scale deployment in improving the performance of your applications. If the implementation does not support interposition then it may be

an indication that performance considerations haven't been high on the developer's agendas.

- The OTS specification is written in such a way as to provide flexibility in implementations. This is primarily so that it can be layered on existing transaction processing systems that may have taken slightly different implementation choices. For example, OTS allows an implementation to always have a separate transaction factory process/server, or the factory can be co-located within the process of the transaction creator. The former implementation choice can affect performance adversely, but may allow for better management. It is worthwhile considering what design-time choices the OTS implementers have taken in these, and other areas. At the moment we know of only one implementation that supports *all* of the flexible choices at runtime, the Hewlett-Packard/Arjuna Technologies Transaction Service.
- The OTS isn't perfect and there are a number of places where there are obvious errors, as we've already seen (remember the issue of subtransactions and heuristics?). However, workarounds and non-standard solutions are possible. You should check to see if the developers have considered these issues and what solutions they have proposed for the implementation. If they do not know about any problems with OTS then we would recommend that you look elsewhere.
- As we saw at the start of this chapter, OTS simply defines the raw two-phase protocol engine. It doesn't specify how to implement or use concurrency control or persistence, for example. So, if you have to implement transactional applications at the level of OTS you're going to need more than just an OTS: at the very least you'll need some kind of `Resource` implementation, and possibly a concurrency control service. Does the implementation you're considering provide any support in these areas? Most assume that you'll be using OTS to control transactions operating on a database and *may* therefore provide a `Resource` that can drive a specific type of database (though even that's not guaranteed); they assume that the concurrency control is then provided by that database. If you're not interested databases, then you may have to worry about implementing your own `Resources`. And if you want nested transactions, you'll have to look at implementing `SubtransactionAwareResources`. There are several commercial implementations that offer good support to developers working at the level of OTS, including IBM and Hewlett-Packard. However, we know of no open source implementations that compare favorably.
- Performance is always an issue with transaction systems, and particularly with OTS. This is because the CORBA specification has evolved over time to provide as much distribution transparency to applications as possible. In the early days of CORBA, it was possible to have different invocation behavior if the object you were calling resided in the same process as opposed to if it resided in a separate process. The latest versions of CORBA solve this issue, but it often results in purely local implementations performing as though all invocations (including those on local

objects) are remote. There are ways around this that can be provided by the ORB or the application (in our case, OTS). So you would be wise to determine beforehand whether or not this issue has been solved by the implementation you are considering.

Summary

In this chapter we looked at probably the most important distributed transaction specification for J2EE: the Object Transaction Service. This is the specification that J2EE recommends for interoperability of its JTA implementations, so you may already by using an OTS implementation whether you know it or not. We looked at how the principles of transaction processing we discussed earlier in the book have been applied in a real-world specification, including failure recovery, heuristics and distribution. In Chapter 1, "Transaction Fundamentals," we discussed the nested transaction model and in this chapter we saw how OTS was the first industrial strength specification to support them, albeit optionally. The fact that nested transactions are supported at all means that you may well find implementations that provide them, which could be important for you.

Through many example scenarios and UML activity diagrams we've seen how much work is actually involved in turning theory into practice for an implementer of OTS, and why issues such as performance and failure resiliency are important. As we also saw, actually using an OTS to implement a transactional application is not trivial, as there is much work involved in writing clients and transactional services. This is perhaps the main reason why most commercial implementations provide higher-level APIs (such as the JTA) to isolate users from the programming intricacies.

Finally, we reiterate something that we said at the start of this chapter: OTS does not define how to *implement* a transaction service. It will not teach you the fundamental principles of transaction processing or how to create an implementation that scales, that performs or is fault tolerant. As someone looking to purchase or use an OTS you should be extremely cautious of implementations that have been written from scratch by people relatively new to the field of transaction processing; this is not to say that transaction processing is an elitist field and only those who have been involved in it for decades can ever be right, because that is obviously wrong as well: it is simply unlikely that an implementation based on the details given in the OTS specification will be best-of-breed. You should be able to use the contents of this book to determine the right questions to ask and what answers to expect.

Likewise, if you are thinking of implementing a transaction service you can do a lot worse than starting with OTS. It is a good specification to conform to, but it is not sufficient to help you through all of the many pitfalls that may occur. We would strongly recommend that you not only use the contents of this book but other texts such as *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter and *Principles of Transaction Processing* by Philip Bernstein and Eric Newcomer before implementing a single line of code: forewarned is forearmed.