Implementing Atomic Objects with the RelaX Transaction Facility

Michael Mock+, Reinhold Kroeger+, Vinny Cahill‡

+GMD (German National Research Centre for Computer Science)
P.O. Box 1316
D-5205 Sankt Augustin 1, FRG
e-mail: {mock, kroeger}@gmd.de

‡Distributed Systems Group,
O'Reilly Institute
Dept. of Computer Science,
Trinity College,
Dublin 2, Ireland.
e-mail: vjcahill@cs.tcd.ie

Abstract

This paper presents an object-oriented model for distributed transaction processing and discusses the issues which had to be explored in the implementation of this model. The functionality of this transaction model meets the requirements of a range of concurrent, distributed applications. We show that transaction functionality can be implemented in a language independent manner allowing the model to be provided through multiple languages. In particular, this paper presents the design and implementation of the Amadeus/RelaX system. We describe the architecture of the RelaX extensible transaction facility and its integration with the Amadeus distributed, persistent object system. The resulting transaction system can support different languages and allows the use of resource types such as files, and database relations as well as objects, and is portable to various UNIX-like platforms.

1 Introduction

Distributed transaction processing is increasingly recognized as being an essential tool for developing reliable distributed applications. Two distinct streams of activity in this field can currently be observed. Firstly, various experimental distributed systems provide programming models in which atomic objects are manipulated on behalf of transactions [Eppinger & Spector 1989; Liskov & Scheifler 1983; Parrington 1990]. Secondly, on-going standardization efforts are defining interfaces and protocols for distributed transaction processing allowing the interconnection of multi-vendor database systems [X/Open Company 1991]. This paper presents the design and implementation of the Amadeus/RelaX[1] system which combines these two streams. We describe the architecture of the RelaX extensible transaction facility[2] and its integration with the Amadeus distributed, persistent object system[3]. The resulting transaction system is portable to various UNIX-like platforms (including those based on micro-kernels), can support different languages and allows the use of resource types such as files, and database relations as well as objects, and is portable to various UNIX[4]-like platforms.

[1]this work is partly supported by the CEC under ESPRIT contract No. 2071, Comandos II
[2]developed at GMD
[3]developed at Trinity College
[4]UNIX is a registered trademark of UNIX System Laboratories in the USA and other countries

RelaX provides a general purpose, extensible transaction facility [Kröger et al. 1990] which is designed to run on various platforms and to support a broad range of application classes. The transaction model provided by RelaX offers functionality that is flexible enough to support the needs of diverse applications such as Computer Supported Cooperative Work and Computer Aided Design/Computer Aided Manufactering [Kroeger & Nett 1991]. A clear separation between distributed transaction management and management of the resources used by transactions is the basis for extensibility. Following the X/Open model for distributed transaction processing (DTP), distributed transaction management is isolated in a server (the so-called Transaction Manager) on every site which cooperates with an extensible set of resource managers. Resource managers (e.g. object management systems, file systems, database systems, mail systems) may implement different kinds of persistent, shareable entities accessible by transactions. Different kinds of resources may therefore be accessed within the same transaction. Each resource manager cooperates with the transaction manager via a standard interface which extends the X/Open XA interface [X/Open Company 1991]. In order to ease the construction of new resource managers, RelaX also implements generic software components providing concurrency control based on read/write locking of overlapping fragments of resources, recovery control, interfacing to the transaction manager, and a generic logging facility as building blocks.

The Amadeus platform provides support for programming distributed and persistent object oriented applications in a multi-user heterogeneous UNIX environment [Horn & Cahill 1991]. Amadeus aims to support applications written in a variety of object-oriented languages, for example C++ [Ellis & Stroustrop. 1990] and Eiffel [Mayer 1988], as well as inter-working between different languages. The support provided by Amadeus is independent of any particular language. In interfacing a language to the platform, that language's runtime object representation and local object invocation mechanism can be retained. A supported language must however provide a set of upcalls which allow Amadeus to manage the objects implemented using that language.

In existing experimental distributed systems that incorporate transactions and objects, atomicity is usually an intrinsic property of an object's type and typically either implemented in a specific language [Liskov & Scheifler 1983; Parrington 1990] or related to a database style background store [Andrews et al. 1991]. We have followed a different approach. The Amadeus/RelaX platform provides a persistent and distributed object space as a generic, i.e. language independent, system layer in which atomic and non-atomic objects co-exist. To achieve a uniform view, and implementation, of the system the basic mechanisms used to manage objects of both kinds are the same. Of course, management of non-atomic objects must not suffer any overhead resulting from support of atomic objects. Thus, the problem was to enhance the system with transaction functionality for a subset of objects while minimizing the impact on the basic mechanisms. The Amadeus/RelaX system provides complete functionality for object-oriented, distributed programming and is capable of supporting multiple languages and resource types.

This paper presents an object-oriented model for distributed transaction processing and discusses the issues which had to be explored in the integration of the RelaX components with Amadeus to provide this model. We believe that the rich functionality of this transaction model meets the requirements of a range of concurrent, distributed applications. We show that transaction functionality can be implemented in a language independent manner allowing the model to be provided through multiple languages. We demonstrate that only local interfacing to the RelaX components is needed to achieve distributed transaction functionality. Finally, we establish the usefulness of the generic modules to support transaction processing by showing the ease of integration with Amadeus.

Section 2 discusses related work on distributed transaction processing. Section 3 presents an overview of both the Amadeus and RelaX systems. Section 4 describes the transaction model provided by the system. Section 5 gives a small programming example showing how the transaction model could be made available to a C++ programmer. Section 6, the main part of the paper, describes the integration of the two systems in detail. Initial performance data is given in section 7, and section 8 contains some concluding remarks.

2 Related Work

In this section a number of related systems and projects are briefly described and their relationship to the Amadeus/RelaX system discussed. As will be seen these systems fall broadly into two categories: research systems providing transactional access to objects and commercial systems aimed at supporting open transaction processing.

<u>Argus</u>
Argus [Liskov & Scheifler 1983] is an integrated programming language and system aimed at supporting the construction of robust distributed programs. In Argus a *guardian* encapsulates and controls access to a collection of data objects to which access is possible only by invoking the operations exported by the guardian. Argus distinguishes the problem of providing atomicity for a computation from that of supporting resiliency. Actions are units of atomic activity. However the atomicity properties are provided only by *atomic objects* and atomicity is guaranteed only when all the objects shared by actions are atomic. Argus supports nested actions [Moss 1981] as well as nested top-level actions, i.e. actions which are created within another action but whose commit is not dependent on that of their creator. Atomic objects are encapsulated within atomic abstract data types. There are a number of built in atomic types as well as facilities to allow users to define new atomic types [Weihl 1985]. To achieve resiliency, a guardian definition may specify a number of *stable variables*. Atomic objects reachable from a stable variable are *stable objects* and constitute the stable state of the guardian. Only the stable state of a guardian is recovered after a failure. The Argus system has been highly influential and our work incorporates and extends ideas pioneered by Argus - including the basic nesting mechanisms [Moss 1981] and the distinction between atomic and non-atomic objects. However Argus is a closed single language system which includes no support for transactional access to external resources.

<u>Arjuna</u>
Arjuna [Dixon et al. 1989; Parrington 1990], is a C++ based system for constructing robust distributed applications. Arjuna provides persistent objects as instances of C++ classes which may be remotely accessed by RPCs within nested transactions. The Arjuna system has been implemented as a specific C++ class hierarchy providing remote invocation, persistence, state-based recovery, and strict 2-phase read/write locking for concurrency control. Transaction support is made available to the C++ programmer through inheritance. Arjuna is therefore a single language system.

<u>Camelot</u>
Camelot [Eppinger & Spector 1989] is an experimental distributed transaction facility developed at CMU which runs on top of Mach. In Camelot so-called *data servers* encapsulate code and data implementing different abstract data types. Applications act as clients which can begin and end transactions and use RPC to request data servers to carry out operations on the data which they maintain. Data servers can be remote from an application. Accessing such a server results in a distributed transaction. A server may also act as a client and call another server. The transaction model allows limited nesting and parallelism inside a transaction, based on concurrent subtransactions. To the programmer of an application or data server, the Camelot services are generally available as a low level library. Full linguistic support for atomic objects and transaction management is provided by the Avalon language which is based on C++ and runs on top of Camelot [Herlihy & Wing 1987]. Access to resources managed by external servers has not been addressed.

<u>Encina</u>
Encina is a commercial system being developed by Transarc (co-founded by key Camelot designers) to support commercial transaction processing applications in an open

distributed environment [Transarc Corperation 1991]. It is especially targeted to run on top of the Open Software Foundation´s Distributed Computing Environment (DCE). The Encina architecture conforms to the X/Open DTP model. Encina provides a transaction monitor as well as a few specialized transactional resource managers (including a record-oriented transactional file server and a transactional queueing system). Integration of RDBMSs is supported by use of the X/Open XA interface. Interoperability with mainframe systems is provided by incorporating an SNA/LU6.2 protocol interface. At a lower level Encina offers services - via the so-called toolkit - such as strict 2-phase read/write locking, logging and 2-phase commit which can be used in building new transactional resource managers. The Encina transaction model supports full nesting of transactions and internal parallelism within transactions. To the programmer the Encina services are available as augmented C constructs mapped to library calls by a preprocessor. For use in OSF DCE environments a simple extension to the RPC mechanism (Transactional RPC) is provided. The Encina approach is open and compared to other systems most closely relates to the RelaX system, but does not provide the same level of transaction functionality. Furthermore, to our knowledge, there is so far no complete system comparable to the Amadeus/RelaX system which makes use of Encina for supporting global atomic objects in a language independent way.

Ontos
Ontos [Andrews et al. 1991] is an object database system which currently supports C++ and extended SQL interfaces. An Ontos database is implemented by a (possibly distributed) storage server. A process using Ontos must be linked with client code which provides the interface to the server. The main interface to a C++ application program provided by Ontos is a set of functions to explicitly retrieve objects from the database into the processes heap where they can be accessed by virtual address. Ontos provides nested transactions. A top level transaction is normally limited to a single process although it is also possible to start a *shared transaction* which several processes may join. Since there is (as far as we know) no concurrency within a single process, nested transactions are most useful to provide an undo facility. Ontos supports conventional read and write locks as well as dirty read locks (which are compatible with write locks). Ontos also provides additional flexibility by allowing the sophisticated application programmer to control buffering of data, cache management and resolution of lock conflicts by providing functions to be upcalled by the system as appropriate. While Ontos is intended to provide a language independent interface, the degree of transparency provided by the system is poor. Moreover, Ontos provides no support for interaction with other database systems.

Profemo
Profemo [Nett et al. 1985], the predecessor of RelaX, delivered a design for, and a prototype implementation of, an integrated distributed system architecture offering a global hardware-supported object space which could be manipulated by transactions. The transaction model is very flexible and corresponds exactly to the RelaX model (see section 3.1). Emphasis was also placed on protection issues leading to a capability-based, hardware-supported object invocation mechanism [Kaiser 1988]. The system exhibits a clean but closed architecture. The main achievements of Profemo were the provision of a flexible, system level transaction mechanism able to support a broad range of applications and its tight integration with the hardware-supported object model. The transaction model was maintained in RelaX, but its tight integration with the dedicated architecture was given up for the sake of extensibility and portability.

Tuxedo
Tuxedo [UNIX System Laboratories 1991] is a widely accepted commercial system supporting distributed transaction processing applications, which access multi-vendor database systems primarily in a networked UNIX environment. (Access to mainframe systems like MVS/CICS is also provided through support for the SNA/LU6.2 protocol). Tuxedo has been the prototype for the X/Open DTP model [X/Open Company 1991]. The main component of Tuxedo is a transaction monitor which must reside at every site. Several (local) relational database management systems (RDBMS) may be connected to each transaction monitor using the X/Open XA resource manager interface. The

transaction model supported allows only simple flat transactions. All resources are database tuples and are accessed via the standard SQL query interface. Internal parallelism within a transaction can be achieved by explicitly breaking down an application into a client (frontend) module and several server modules which can be invoked concurrently by the frontend. Tuxedo is restricted to supporting simple flat transactions involving multiple database systems and does not support distributed, object-oriented programming.

<u>Quicksilver</u>
Quicksilver [Haskin et al. 1988; Schmuck & Wyllie 1991] is an experimental operating system which includes support for distributed transactions. Applications are structured as sets of cooperating processes, including an extensible set of server processes, that communicate via Quicksilver IPC. Applications can define sets of potentially concurrent server invocations to be grouped into flat transactions. In addition, existing applications can be transparently encapsulated to run as transactions. The transaction manager (TM) is a special server which is notified by the operating system of on-going transactional IPC messages. Servers can join transactions by registering with the TM. Emphasis is put on supporting various modes of Server/TM collaboration optimizing the performance of the commit and abort protocols for read only-servers or for servers that only maintain volatile state. The main result of Quicksilver is a demonstration that distributed applications really can take advantage of a low-level integrated transaction service supporting different resource types. The main disadvantages are the flat transaction model and the tight integration with a specific operating system which is an obstacle for reuse of existing applications. Object orientation, although not excluded by the approach, was not addressed.

3 Background

This section gives an overview of the Amadeus and RelaX systems, providing the necessary background for understanding the remainder of the paper.

3.1 Amadeus

The fundamental goal of Amadeus was to provide a level of support which would allow a range of object-oriented languages to be used to write distributed and persistent applications without requiring each language to adopt a common object model. In order to achieve this goal the platform consists of two main components: the Generic Runtime (GRT) which provides generic support for the management of distributed and persistent objects, and the Kernel which provides the underlying support for distribution, persistence, sharing and integrity. Language Specific Runtimes (LSR) can then be implemented for each supported language which adapt the support provided by Amadeus to that particular language's object model (see Figure 1).

Physically, Amadeus consists of a library - incorporating the GRT and part of the kernel - with which programs written in supported languages must be linked, and the Amadeus server daemon - implementing the remainder of the kernel - which must be run on each site participating in an Amadeus system.

3.1.1 Amadeus Concepts

Amadeus directly supports a model based on passive objects being accessed by distributed processes. At any time an Amadeus system may include applications running on behalf of many different users sharing objects in a controlled way. The following sections give an overview of the functionality provided by Amadeus.

<u>Global and Persistent Objects</u>
Current object-oriented languages typically manipulate local volatile objects, i.e. an application can only use objects which are present in the address space in which the application is running. Moreover, the lifetime of any objects created by the application is, by default, bounded by the lifetime of the process in which the application is running.

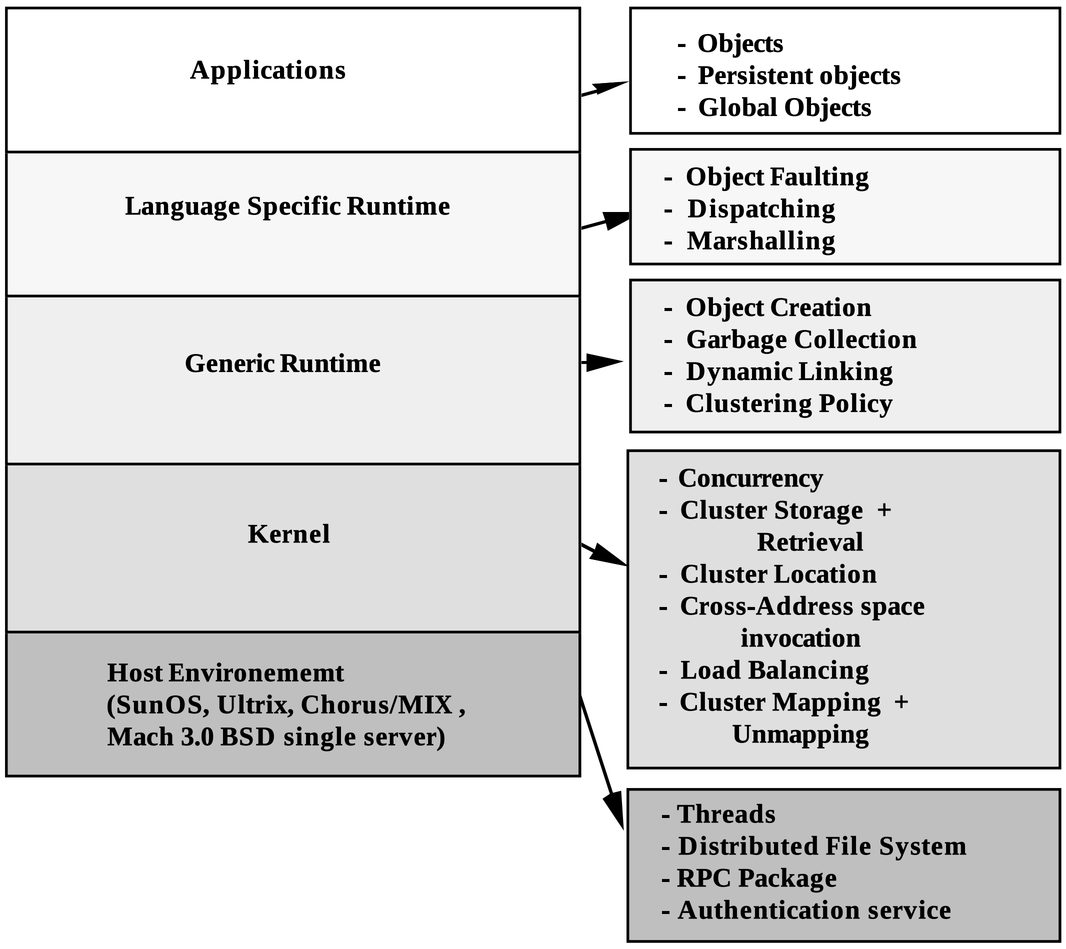


Figure 1: Structure of the Amadeus Platform.

In Amadeus, an object may be created as a *persistent object*, meaning that the object exhibits the property of *potential persistence*. Such an object is guaranteed, in the absence of failures, to exist for as long as it is transitively reachable from some designated root object(s), independent of the lifetime of the process that created the object. When the object is no longer reachable from any root, it is garbage, and may be deleted from the system. Also in Amadeus, an object may be created as a *global object*, meaning that it is possible to access the object, in particular to invoke operations on the object, even if it is currently located in a different address space, possibly on a remote site. Hence, an Amadeus application may use either local volatile objects, global volatile objects, or, local or global persistent objects as required. The way in which global and persistent objects are made visible, if at all, to the application programmer in a particular language depends on the language in question [Cahill et al. 1991].

Jobs, Activities, Processes and Contexts

A *job* is a distributed process consisting of a set of *activities*. Activities are distributed threads of control, i.e. analogous to lightweight processes but with the possibility of executing in several address spaces or *contexts* at the same or different sites at different times. An activity may be active in at most one context at any time. Each activity is implemented by one or more lightweight *processes* in each visited context.

A job initially consists of a single activity executing in a single context. The activity executes by invoking operations on objects. When invoked a given object may be located either in the activity's current context, in another context at the current site, in some context at a remote site or in secondary storage. If located in other than the current context then the current activity takes an *object fault*, analogous to a page fault in a conventional operating system, which will be resolved either by performing a cross-context invocation, i.e. between contexts potentially located on different sites, or by mapping the object into the faulting context. In the case where the invocation is carried out in a different context, the invoking activity is transparently extended to the target context.

A new job is created for each application run by a user. An activity can also create a totally independent job to carry out an operation invocation. In either case, the job terminates when all of the activities created within the job have terminated. A new activity can be created at any time to carry out an invocation asynchronously.

Clusters and Containers

Clusters are used to store groups of related objects. Every global or persistent object must belong to some cluster. Clusters are the units of storage and mapping supported by Amadeus. Clusters allow Amadeus to exploit the locality of reference between related objects by allowing them to be grouped in the same cluster.

Clusters are the units of mapping into a context. Each context contains a set of clusters which may vary dynamically as more clusters are mapped into or unmapped from the context. By default, clusters are only unmapped from a context when the context is being deleted i.e. after the last invocation that was active in the context completes. A cluster can also be explicitly unmapped from a context by the application if none of the objects in the cluster are needed by activities running in the context.

In Amadeus, secondary storage is divided into a set of *containers*. There may be zero, one or more containers at a given site. Currently each container is stored entirely at a single site however in the future we envisage supporting replication of containers. Each container stores a subset of the clusters in the system.

3.1.2 Logical Architecture

Logically, jobs - and activities - execute in a distributed virtual memory, known as*Virtual Object Memory* (VOM), which is implemented by a collection of distinct contexts. The state of the VOM is volatile. Part of this state may be lost as a result of a site or context failure. The stable background representation of any object is maintained in the *Storage Subsystem* (SS). In particular, each object is stored as part of a cluster. The site at which a cluster is stored is called the *storage site* for the cluster.

When an object is accessed for the first time its cluster is *activated*, i.e. an image of the cluster is mapped into VOM so that accesses to the required object and to other objects in the cluster can take place possibly resulting in updates to the virtual memory images of the objects concerned. The site at which the cluster is mapped in known as the *activation site* for the cluster. Updates to a cluster while activated are not immediately reflected in the state of the cluster maintained in the SS. Only when the cluster is *passivated*, i.e. unmapped from VOM, are the updates written back to the SS. Updates to activated clusters may therefore potentially be lost as a result of site or context failures.

Central to the operation of Amadeus is that only a single image of any cluster may be activated at any time, i.e. a cluster is either mapped in exactly one context or is stored in the SS. The *Location Service* (LS) is responsible for keeping track of the current location of each mapped cluster. Moreover, the responsibility for ensuring that only a single image of each cluster is mapped, rests with the LS.

Since it would be too expensive for each site to keep exact information about the location of every mapped cluster, information about the locations of mapped clusters is partitioned. A single site is assigned the responsibility of maintaining information about the current location of all the clusters from a given container. That site is known as the *control site* for the container. The LS at a cluster's control site acts as the central authority in determining whether or not a cluster is activated, thereby ensuring that an image of the cluster is only mapped once in virtual memory.

3.2 RelaX

This section presents an overview on the RelaX system and the generic components provided by RelaX.

3.2.1 The Generalized Transaction Concept

The generalized transaction concept supported by RelaX is derived from the Profemo project [Nett et al. 1985; Nett et al. 1986].

RelaX guarantees the usual transaction properties of serializability, all-or-nothing, and permanence of effect. In contrast to conventional transaction systems, isolation of transactions is not mandatory. RelaX provides a means for the controlled use of uncommitted data in order to increase the concurrency and efficiency of the system[1]. A transaction that uses uncommitted data *depends* on the transaction that produced that data. Such a transaction cannot commit or abort independently and may, once terminated, be required to wait for the commitment of any transaction on which it depends before committing. The system checks for and keeps track of dependencies between transactions. These are then taken into account during the execution of the commit protocol in order to achieve a transaction consistent system state. For a more detailed description the reader is referred to [Schumann & Mock 1989].

Since the system can distinguish between the successful termination of a transaction and its commitment, transactions can terminate in an additional state other than the committed or aborted states, i.e. the *completed* state. A completed transaction may be committed later in a (potentially distributed) group commit. Note that, in contrast to the database notion of group commit, delaying commitment does not imply preventing access to the results of the transaction.

Concurrency control between transactions is implemented using non-strict two-phase read/write locking [Eswaran et al. 1976]. The lockpoint indicating the beginning of the shrinking phase of the transaction is not necessarily combined with its commit point. Premature release of locks allows the results of the transaction to be made available before its commit. Note that this feature does not result in dirty reads since serializability and the all-or-nothing property are still guaranteed.

Concurrency within transactions is supported in two possible ways. Full nesting with potentially concurrent subtransactions and concurrent processes inside each individual (sub)transaction are both supported. A model based on single writer/multiple reader locks is used for the synchronization of concurrent processes inside of a single transaction. By combining nesting with support for concurrent processes, concurrent processing at different nesting levels is possible. In addition, nesting of transactions for recovery purposes only is possible. In this case the synchronization level of a parent and its subtransaction are the same but the subtransaction is able to abort independently from its parent.

3.2.2 Overall Architecture

This section describes the architecture of the RelaX system which provides the transaction model described above. The main goal of this architecture is to provide the facilities for distributed transaction management in an extensible and portable way, allowing multiple resource types to be supported. An additional goal was conformance with the X/Open model and, in particular, the XA interface. Figure 2 depicts the resulting overall architecture.

1A transaction can however specify that it requires the use of committed data only, thus achieving traditional isolation as a special case

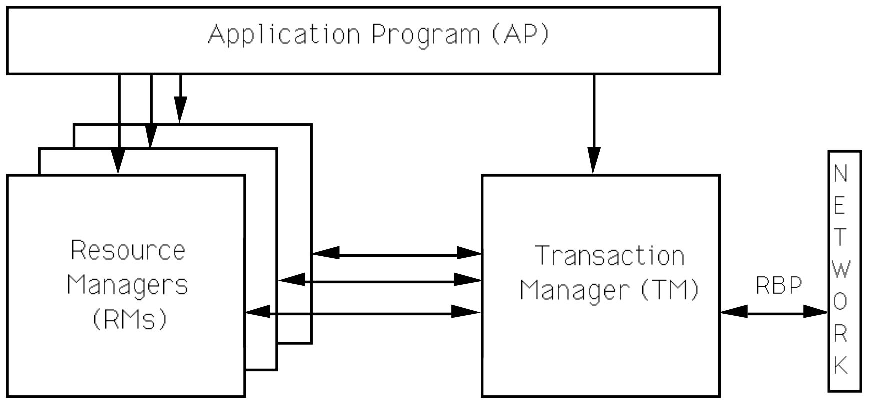


Figure 2: The RelaX architecture.

Every site runs a separate server known as the Transaction Manager (TM) which is responsible for transaction control and which provides, in collaboration with the TMs at all other sites, a consistent system wide view of the outcomes of distributed transactions. In particular, the TMs execute the distributed commit and abort protocols implementing distributed transaction processing. TMs communicate via a reliable broadcast protocol (RBP) [Vonthin 1987] which extends the protocol proposed in [Chang & Maxemchuk 1984] with enhanced facilities for site failure and restart detection. The RBP reports changes in the system configuration to each TM. TMs use this information to decide whether to abort on-going transactions affected by a failure or to reschedule interrupted commit protocol executions after a restart.

From the TM's point of view, the active entity responsible for computational progress is the "application program" (AP). A "resource manager" (RM) is responsible for managing persistent data. Of course, there may be more than one AP or RM in any system. Different RMs may provide different kinds of data with different interfaces to the AP. The distinction between AP and RM is a logical one and does not prevent them being represented by the same entity.

An AP defines transaction boundaries, aborts a transaction or sets the lockpoint of a transaction by calling the TM, which propagates these events to all relevant RMs on every affected site. Every AP must be linked with the TM-library which logically connects it to the TM. The TM-library caches some information concerning each transaction (such as the transaction identifier or the set of sites involved in the transaction). Calls to the TM go through this library but can be short-circuited based on cached information.

Inside of a transaction, an AP uses the resources of one or more RMs. If computations go remote to a new site, the TMs on both sites are informed in order to keep track of distribution. Some transaction information that is transparent to the RPC package is piggy-backed on each RPC message. Note that remote communication is required to be RPC structured to avoid the need for additional termination detection mechanisms.

If a transaction aborts[1], the TM notifies the corresponding AP. To support consistent roll-back of the computational state of the AP (which should not be confused with the state of data maintained by a resource manager), the TM supports a model in which an AP is structured as a collection of potentially concurrent *recovery participants* each of which is a sequential thread of control executing on a single site. The actual representation of a participant is invisible to the TM, and could be either a thread or a process. The TM is informed as participants join and leave transactions so that the TMs always know which participants are running in a given transaction. This information is

[1]note that the abort might have been initiated remotely, e.g. by a site failure

used to notify all the participants in case of an abort and to map failure of a participant to abort of its transaction (if so desired).

Every RM collaborates with the TM via the TM-RM control interface to provide the transaction properties for its resources. Again, the actual representation of a RM is transparent to the TM. The RM coordinates state changes of transactions (commit/abort) with the corresponding state changes of its resources. This is basically a local extension of the distributed protocols executed by the TM. The key point here is that the TM maintains a so-called outcome-log to keep track of the progress of commit protocol executions while the RM provides an interface which allows its resources to be updated according to a two-phase-update protocol. This interface may be realized internally by use of a data-log. For a more detailed description, see [Schumann et al. 1990]. Accesses to atomic resources are tagged with the identifier of the current transaction taken from the TM-library. When a RM becomes involved in a transaction for the first time, it informs the TM that it is joining the transaction. Therefore, the TMs have knowledge of the RMs involved in every transaction. The TM also supports the notion of *static* RMs [X/Open Company 1991] that are involved in every transaction by default and do not have to explicitly join a transaction. RM failures are mapped by the TM to aborts of those transactions in which the RM was involved. RM restarts trigger the resumption of commit protocol executions which were interrupted by the failure of the RM.

3.2.3 Generic Components

Any RM conforming to the interface provided by the TM can be integrated into the architecture irrespective of how the RM implements transaction functionality internally. The implementor of an RM can, but is not forced to, make use of generic software components provided by RelaX to support RM internal transaction management. The integration of Amadeus and RelaX was considerably simplified by using these components. Figure 3 shows the logical structure of a RM and outlines those parts (concurrency control, recovery control, TM-RM interface, and logging) which are supported by the generic components.

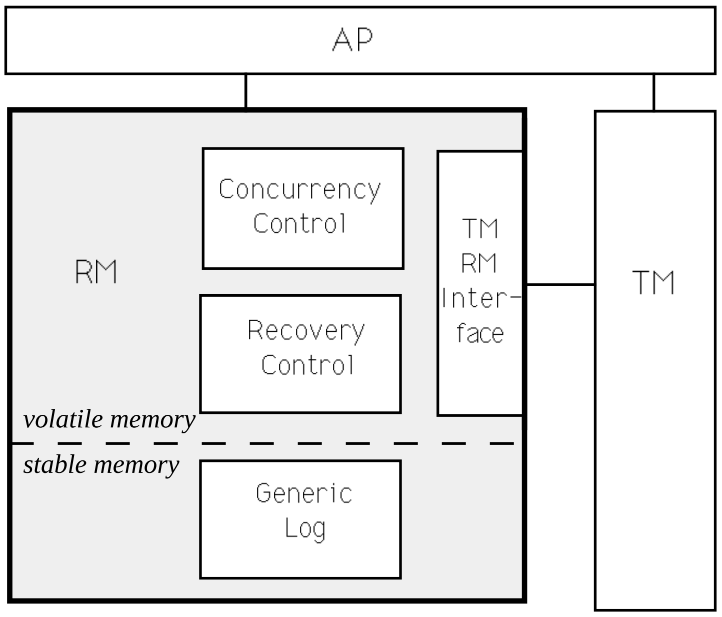


Figure 3: Logical Structure of a Resource Manager.

The resources provided by the RM are stored in stable and volatile memory. Stable memory contains committed and prepared versions of resources. Usually, access to a resource requires a volatile memory representation of the resource. Note that stable memory does not necessarily imply that the RM maintains a version of the resource on a local disk - a distributed implementation of the RM's store is possible.

The RM-library consists of three modules: the TM-RM control module implementing the interface between the RM and the TM, the concurrency control module implementing locking and the recovery control module supporting the commit and abort of transactions. The recovery control module maintains, for each transaction, the set of recovery points needed, in case of an abort, to restore the before image of each resource used by the transaction. In case of a (group) commit, it computes the after images of the resources modified by the transactions which are committing.

The key to the genericity of these modules lies in the abstraction of the actual resource type making it opaque to the RM-library. A resource is identified by an (opaque) identifier type and is expected to implement some low-level operations (described below) which can be called by the RM-library. Accesses to resources are trapped and reported to the RM-library for concurrency and recovery control checking based on the identifiers of the current transaction and the resource, and the mode (read/write) of the attempted access. The RM-library supports the notion of fragmented resources allowing for access to possibly overlapping fragments of a resource.

The concurrency control module can handle different locking modes (with corresponding lock compatibility matrices) for different resources. The default strategy is non-strict two-phase read/write locking. Other compatibility matrices specifying rules for granting locks can be linked to the concurrency control module at run-time on a per resource basis. Concurrency control is otherwise handled completely inside the library, i.e. no additional operations from the RM are required. This is due to the fact that any decision on the granting of a lock is independent of the representation of the resource. For internal synchronization, i.e. synchronization of concurrent accesses to a resource within a transaction, the RM-library implements a simple single-writer/multiple-reader algorithm. The scheduling units that compete for the resources are represented within the RM-library by the abstract type *synchronization participant* whose actual representation is again opaque to the RM-library. For blocking and unblocking of requests the RM-library assumes basic suspend and resume operations on synchronization participants to be implemented by the RM.

The recovery control module assumes that every resource has a committed state which is stable and persistent and which will only be updated under control of the commit protocol. All accesses to a resource are directed to its actual state which corresponds initially to the committed state and is modified to reflect all updates to the resource as they occur. When a transaction modifies a resource for the first time, the RM-library calls a *save* operation, which must be provided by the RM, to create a recovery point for the resource. This recovery point is supposed to contain enough information to restore the actual state of the resource to its state before the transaction in case of an abort. The save operation returns a handle identifying the recovery point that is passed by the RM-library to a *restore* operation if the transaction aborts. Note that, as the transaction model allows nesting and the use of uncommitted data, there may be a set of recovery points associated with a resource. This set does not necessarily grow and shrink in a stack-like manner since the transaction model exceeds pure nesting. The actual state of the resource is accessible to all active transactions, each saving its individual before image into a recovery point. This before image corresponds to the after image of the preceeding transaction. Recovery points are, in general, expected to be value-based although other approaches are not precluded. Their representation is hidden to the RM-library. If a transaction reads uncommitted data from another transaction, the recovery control module also informs the TM about the dependency between these transactions (if so desired).

For transaction commit, the recovery control module requires the RM to implement operations to support two-phase update of resources. In the prepare phase, the RM must save the resource state stably in an intermediate store without deleting the committed representation of the resource. Depending on the final commit decision, the RM must then commit or discard the intermediate version of the resource. The implementation of the two-phase-update operations may make use of the generic log component discussed below.

The generic log component implements a generic logging model supporting protocols whose behaviours are specified by finite state machines. Several finite state machines realizing different protocols can be interpreted concurrently. A finite state machine describes the states of the protocol (for instance, collect_data, prepared, committed, and propagated in case of the two phase commit protocol), possible state transitions, and associated operations. Events that are reported via the external interface to the log trigger state transitions. Each event is stored on the log. In addition, the generic log component maintains a volatile data structure - the log-table - which contains the current state of all on-going protocols. Every state machine includes a specified final state (e.g. propagated in the example above), which, when reached, denotes the fact that the corresponding protocol has reached its end and can be deleted from the log-table. Compaction of the log is realized by checkpointing the log-table. On restart, the log is read from its beginning, replaying all events found and thereby rebuilding the log-table.

A possible, efficient implementation of the required recovery point and two-phase-update operations is based on maintaining recovery points as copies in volatile memory. During commitment after images have to be stored stably. This approach affects normal processing less than stable logging of before images and after images on every access. Furthermore, the performance gains made possible by access to uncommitted data and group commit can be fully exploited. The resulting activation cycle for a resource is shown in Figure 4:

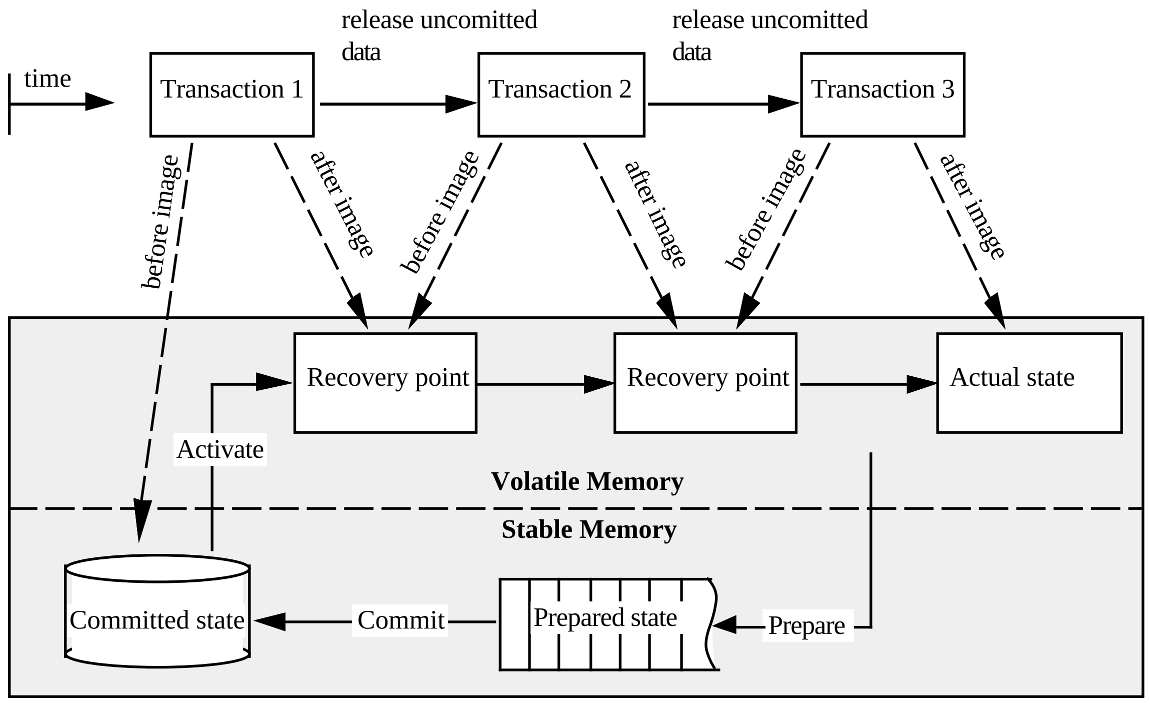


Figure 4: Activation cycle of a resource.

The passive state of the resource corresponding to its last committed state is stored in stable storage. The resource becomes active, i.e. is brought into volatile memory, when it is accessed by a transaction for the first time. All modifications to the actual state are performed in volatile memory. Once activated, the resource can be used by several transactions, each of them copying its before image into a recovery point. Only committing a transaction forces a version of the resource (which might be stored in a recovery point due to further modifying transactions) back to the stable storage. This can happen concurrently with on-going transactions that use the actual, uncommitted state of the resource.

4 Transaction Model

This section describes the transaction model provided in Amadeus by use of the RelaX components.

4.1 Atomic Objects

The unit of concurrency and recovery control is the object. Objects may be classified as being either *atomic* or *non-atomic*. The transaction properties only apply to operations on atomic objects carried out within a transaction. No guarantees are made concerning operations on non-atomic objects. Both types of objects may be accessed within a transaction, but only atomic objects are guaranteed to be consistent in spite of failures. Outside of transactions, atomic objects may only be read, thus providing a mechanism for programmers to by-pass the restrictions imposed by transactions without violating the consistency rules. Attempts to modify atomic objects outside of a transaction result in an exception being raised.

It should be noted that the atomicity of an object is orthogonal to its class i.e. there may potentially be both atomic and non-atomic instances of the same class. The system includes a mechanism for creating atomic objects and for promoting non-atomic objects to be atomic objects. Once an object is atomic it cannot be changed to become non-atomic thus ensuring time-invariant consistency of the collection of all atomic objects. The motivation for supporting both categories of objects is primarily to allow application designers to avoid the overheads associated with accessing atomic objects when using objects which have no strong requirements for consistency.

Although normally hidden from the application programmer, it is also possible to allow access to the basic concurrency control operations implemented in the system so that the application may perform its own locking and bypass the default locking provided by the system.

4.2 Transactions

An invocation on an object can be carried out as a transaction - note however that, by default, invocations on atomic objects do not cause new transactions to be created. Transactional invocations are trapped by the system and a transaction created before the invocation is executed as normal. When the invocation finishes successfully, the transaction ends and normal execution is resumed. If the transaction aborts, an exception is raised which may be caught by the activity that created the transaction. When a transaction is created it is possible to provide some options specifying the nature of the new transaction (i.e. whether the transaction should be run at the same synchronization level as its parent transaction or whether the transaction should end in a committed or completed state).

Note that there is no explicit operation to end a transaction, however an operation is provided to explicitly abort the current transaction which may be called by any participant of the transaction and which results in an exception being raised in the activity which created the transaction. A transaction may be started by an activity at any time. If the activity is not currently part of a transaction then the transaction is created as a top-level transaction. Transactions created within a transaction are created as nested transactions. An escape facility is provided by creating a job. New jobs are not considered as part of the current transaction and a transaction created within a new job is created as new top-level transaction. Concurrent activities may however be created within a transaction.

4.3 Impact on Amadeus Model

Adding atomic objects and transactions to Amadeus as described in the previous sections has a number of ramifications for the Amadeus model. In particular, the execution of activities running within transactions must be restricted so that when an activity visits a different context it cannot be allowed to return until all processing initiated by the activity from that context is complete. In particular, any activities created as a result of the call must have completed before the original activity returns. Moreover, by the time the invocation which initiated a transaction has completed, all computation on behalf of the transaction must have terminated.

Supporting both atomic and non-atomic objects in the same system may lead to inconsistencies which would not arise if all objects were atomic. For example, a dangling reference can arise if a non-atomic object holds a reference to an atomic object whose creation has been undone due to the abort of the creating transaction. However it is worth noting that his particular situation can also arise, in the absence of atomic objects, if an object holds a reference to another object which is lost due to a site failure.

4.4 Fault Model

The fault model assumes three types of faults:

1. Transaction faults: which occur when a transaction is explicitly aborted or when an unhandled exception is raised inside of a transaction;
2. Context faults: which occur when a single context fails resulting in the loss of a single address space.
3. Site faults: which occur when a site or its operating system fails, resulting in the loss of the entire virtual memory of the site.

All faults are mapped to the abort of those transactions that depended on the lost state and which were not already prepared or committed at the time of the fault.

Objects may also become temporarily unavailable for several reasons, e.g. because the object's storage and/or control site is down. If the control site for some container is down then it may not be possible to access objects from the container even if they are mapped at some other site. It will certainly not be possible to activate or passivate objects from the container effected, even if their storage site is up. Likewise, if the storage site for a container is down, it will not be possible to activate or passivate objects from the effected container. Any attempt to access an object in these circumstances will lead to an exception which, depending on the application's semantics, may be mapped to a transaction abort.

5 C** Example

The following simple program illustrates the use of transactions in Amadeus. The program is a simple interactive calendar written in C** - a version of C++ extended with distribution, persistence and, now, atomicity, supported by Amadeus [Horn & Cahill 1991].

The program (Figure 5) defines three classes - `day` (line 2), `calendar` (line 9) and `menu` (line 18). Since `day` and `calendar` are defined to be 'permclasses', instances of these classes are persistent objects. In addition `day` and `calendar` are 'global classes' - since they have 'global' methods (lines 6 and 14) - so that instances of these classes are also global objects. The class `menu` has an 'active' method which can be the initial operation of a job, activity or transaction (line 22). Finally note that the `display` method of both `day` and `calendar` is non-modifying (lines 7 and 16).

A menu object provides an interactive interface to a single calendar which in turn contains entries for a number of days. Each day is represented by an atomic object created when its containing calendar is created (line 13). Note however that instances of `day` are, by default, non-atomic as are instances of `calendar`.

The first time the program is run (lines 48-49) an atomic instance of `calendar` is created (line 48) and a reference to it stored into the Amadeus name server (line 49) thereby designating the calendar as a root object and ensuring that it persists.

```
#include <amadeus.h>

permclass day {
 char text[128];
public:
 day() { text[0] = '\0'; };
 global void write(const char *txt) { strcpy(text, txt); }
 global void display() const {printf("\n\t<< %s >>\n\n",text); }
};

permclass calendar {
 day  *days[356];
public:
 calendar() { for (int i = 0; i <= 355; i++)
              days[i] = new {ATOMIC} day(); }
 global void write(const char *txt, int day_no)
                  { days[day_no]->write(txt); }
 global void display(int day_no) const {days[day_no]->display();}
};

class menu {
 calendar *cal;
public:
 menu(calendar *c) { cal = c; }
 active void start(int);        // Can be run as a transaction
};

void menu::start(int i) {
  char c[5], text[30];
  int day_no;
  for (;;) {
  printf(
     "\n\tE <day> <text>  - enter text <text> for day <day>\n"
     "\tD <day>           - display text for day <day>\n"
     "\tX                 - exit (saving changes)\n"
     "\tQ                 - quit (without saving changes)\n"
     "\n\t Enter your choice ? ");
  scanf("%s",c);
  switch (toupper(c[0])) {
          case 'E' :   scanf("%d%s", &day_no, text);
                       cal->write(text, day_no);
                       break;
          case 'D' :   scanf("%d", &day_no);
                       cal->display(day_no);
                       break;
         case 'X' :    return;
         case 'Q' :    amadeus.transaction_abort(); }
  }
}

main()
{ if (amadeus.reset()) {
     calendar *cal = new {ATOMIC} calendar();
     amadeus.record("CALENDAR", cal );
    } else {
     calendar *cal = amadeus.lookup("CALENDAR" );
     menu *m = new menu( cal ) ;
     ts_options options;    // Use default transaction options
                            // Use top-level transaction
     m->start( TRANSACTION, options, 1 );
   }
}
```

Figure 5: C** programming example.

Whenever the program is subsequently run (lines 51-55), the reference to the calendar is retrieved from the name server (line 51) and used to create a new volatile instance of `menu` which provides the user interface to the calendar (line 52). The interactive menu is then launched as a transaction (line 55) by calling an overloaded version of the `start` operation - generated automatically by the C** compiler - which also takes as parameter a set of options specifying various attributes of the transaction which in this case are empty.

From the menu a user can display or update the calendar. Invoking the first operation on the calendar will cause the `calendar` object to be activated at some site. Displaying a day will result in a read lock on both the `calendar` and the appropriate `day` object being acquired while updating a day will result in write locks being acquired. Note that several instances of the program may be active simultaneously, possibly at different nodes, but sharing the same calendar.

When a user has completed his/her session he/she can choose either to save any changes made - by returning from the menu operation causing an attempt to commit the transaction to be made (line 42) - or can discard any changes by explicitly aborting the transaction (line 43).

6 Amadeus-RelaX Integration

This section describes in detail the integration of the Amadeus and the RelaX systems.

6.1 Integration Architecture

In Amadeus, each context acts as the Resource Manager (RM) for the objects that are currently mapped into the context and as the Application Program (AP) with respect to all object invocations executed in the context.

The Transaction Manager (TM) is implemented as a UNIX process which is spawned by the Amadeus server on system (re)start. The resulting concrete architecture is depicted in figure 6:

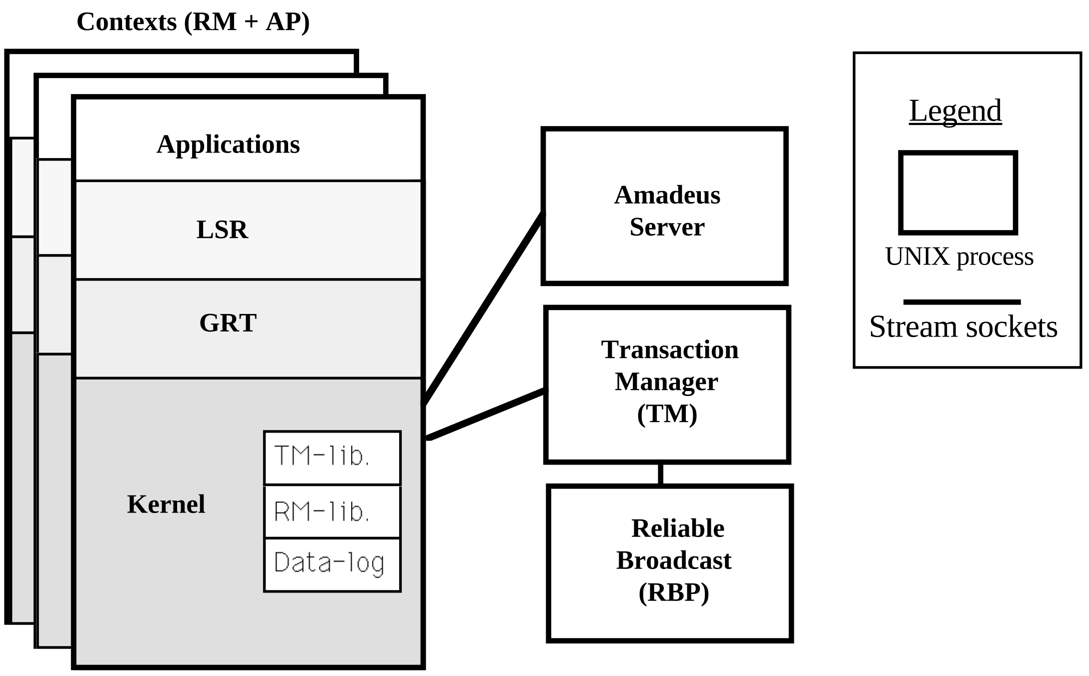


Figure 6: Amadeus-RelaX Integration Architecture.

Every context includes the TM and RM libraries which logically link the context to the local TM. In addition, every context maintains a data-log to support commit processing and restart handling. The TM and RM libraries and the data-log are encapsulated within the Transaction Subsystem (TS) of the Amadeus kernel.

6.1.1 The Generic Runtime Interface

One of the most important goals of integrating Amadeus and RelaX was to allow transaction support to be added to a range of (object-oriented) languages with minimum effort. Transaction support therefore had to be added below the Amadeus GRT interface. However, the addition of transaction support to Amadeus required a number of extensions to the GRT interface in order to allow the support provided by the system to be exploited from a language. The necessary extensions to the GRT interface are outlined in this section.

Essentially the GRT interface provides routines to allow a language specific runtime to create and control transactions and to report attempts to access atomic objects. The routines that have been added to the GRT can be described as follows (Figure 7):

```
Outcome CreateTA(OpDesc *Op, Bool NewSyncLevel, Bool CommitReq,
                 Bool CommittedOnly, Bool WaitOnLock, Bool
AnnounceDep);
void make_atomic(void * object);
void AbortTA();
void SetLockPoint();
void PreAccess(void *object, AccessMode mode);
void PostAccess(void *object, AccessMode mode);
void SetLock(void *object, AccessMode mode);
void ReleaseLock(void *object, AccessMode mode);
void UseObject(void *object, AccessMode mode);
void FreeObject(void *object, AccessMode mode);
```

Figure 7: Generic Runtime Interface for Transactions.

`CreateTA` creates a new transaction returning only when processing on behalf of the transaction is terminated. The return value describes the outcome of the transaction which can be committed, aborted or just completed. The `Op` parameter describes the operation to be carried out as a transaction and contains the name of the target object, the operation to be carried out and the parameters to the operation in a standard format. The remaining parameters specify some options for the new transaction. When a nested transaction is created, `NewSyncLevel` specifies whether or not the inner transaction is created only for recovery purposes. `CommitReq` specifies whether or not the transaction should be committed immediately when completed. `CommittedOnly` allows the caller to specify that the transaction should only use committed data. `WaitOnLock` specifies whether or not to block when a lock cannot be acquired immediately and, finally, `AnnounceDep` specifies whether or not to inform the TM of the transaction's dependencies on other transactions.

With the `make_atomic` call, an object is promoted to become atomic. `AbortTA` simply aborts the current transaction, if any. `SetLockPoint` sets the lockpoint of the current transaction.

`PreAccess` and `PostAccess` are used to report an attempt to access an atomic object to the system. Note that it is the responsibility of the language (specific runtime) to detect attempts to access atomic objects - the interface provided by the GRT allows the attempt to be reported to Amadeus. In particular `PreAccess` should be called before the access takes place and `PostAccess` immediately afterwards.

Finally, in addition to implicit concurrency control provided by `PreAccess` and `PostAccess`, explicit concurrency control is possible through `SetLock` and `ReleaseLock`. In addition, `UseObject` and `FreeObject` can be used to synchronize concurrent participants of transactions.

6.1.2 Mappings

The following paragraphs describe the mappings between the abstract entities of the RelaX transaction system and the entities implemented by Amadeus in more detail.

Participants

The RelaX concept of recovery participant is mapped to the Amadeus concept of process. This is the most natural mapping since both recovery participants and processes are local to a single site. Creation of a transaction blocks the current process and creates a new process for the transaction in the context thus providing a way of cleanly rolling back to the previous state of the computation, in the case of a transaction abort, by simply killing the process(es) involved.

Since an activity may be represented by more than one process in a given context and since each of these processes should be able to use the locks held by other processes belonging to the same activity without blocking, a RelaX synchronization participant is represented by an Amadeus activity rather than a single process.

Resources

In Amadeus the resources (i.e. the units of recoverable state) handled by the RM are objects. Objects are non-fragmentable resources, so only whole objects may be specified for recovery and concurrency control.

Resources in RelaX are identified by the opaque type `RES_ID`. In Amadeus, a `RES_ID` corresponds to the virtual memory address of the corresponding object as this is sufficient to uniquely identify the object within the context. Naturally, this means that an atomic object cannot be moved once mapped into a context (e.g. by the garbage collector).

Recovery Points

In Amadeus a recovery point is simply a (virtual memory) copy of an object. Recovery points in RelaX are identified by the opaque type `REC_HANDLE`. Similar to resources, a `REC_HANDLE` in Amadeus is the virtual address of the recovery point.

6.1.3 RelaX Requirements on Amadeus

Invocations on atomic objects must be trapped and reported to the RM-library, which performs local concurrency and recovery control based on the identifiers of the current transaction and the target object, and the type of access (either read or write) that is being attempted.

For recovery control, Amadeus must provide supporting operations which are called by the RM-library. To handle transaction aborts, operations to save and restore object states to and from recovery points are required. Of course, the Amadeus garbage collector must take references stored in these recovery points into account in case the recovery point is subsequently restored. For commitment, Amadeus provides operations to prepare an image of an object to be written to the log and is responsible for calling the data-log to store the object. On the restart of a context, Amadeus uses the log-table to rebuild object states and to resume commit protocol executions. New objects can also be created in a context. If the object is atomic, the RM-library must be informed of its creation so that it can undo the creation or initiate a stable save for the object as required.

If an invocation leaves the current context, the target context must notify the local TM in order to join the transaction. If an invocation goes remote, the TMs on both sites must be informed. Information used by the TMs at the calling and called sites is piggybacked on each remote invocation call and return message sent by Amadeus.

If a transaction aborts, the local TM calls the TM-library in each participating context which must be able to reset the local state of the aborted transaction. In the current implementation this is achieved by simply aborting the local process(es) belonging to the transaction.

6.2 Distributed Algorithms

Having each context act as a RM for those objects that are currently mapped into the context requires distributed algorithms to manage atomic objects given the possibility of remote activation of objects from remote storage sites. These algorithms are complicated by the fact that clusters are the units of transfer from/to the storage site and may contain objects which may be accessed individually by distinct transactions. This section describes the algorithms that implement distributed object management to realize the RM model.

6.2.1 Activation, Passivation and Location of Clusters

A cluster can contain both atomic and non-atomic objects. However, the activation and passivation of clusters is orthogonal to transaction management, i.e. many transactions which modify (the atomic objects stored in) a cluster may commit while the cluster is activated. Note, however, that a cluster containing atomic objects may only be passivated if all commit protocol executions involving objects belonging to the cluster have terminated.

Since the contents of virtual memory may be lost as a result of a fault, the main responsibility of the Transaction Subsystem (TS) can be seen as ensuring that committed updates to atomic objects are not lost and that an image of each activated cluster, containing the youngest committed version of each atomic object belonging to the cluster, can be recreated in virtual memory on restart. In particular, restart handling (see section 6.2.4.) ensures that every activated cluster, to which changes were committed or prepared[1], is restored in virtual memory at its activation site. Activated clusters, to which no changes were committed or prepared since activation[2], are lost from virtual memory if the activation site fails. In this case, the SS then holds the up to date state of the cluster.

Since the activation site interacts with the SS only when activating and passivating whole clusters, and since activation and passivation are orthogonal to transaction management, the SS is not involved in committing updates to objects. A failure of the SS while an object is activated does not cause transactions to abort nor prevent them from committing. However, as explained later, the SS may be involved in the restart of the activation site of a cluster.

A central premise of the system is that each object is mapped into only one context. Hence, it must be ensured that attempts to activate an object are directed to its youngest committed state. This is not always trivial, since the youngest committed state of an object may, in the event of a failure of the activation site, can be found only on a data-log once modifications to the object have been committed. The object's representation in the SS is obsolete at this stage. In this case any attempt to access, and in particular, to activate the object must be delayed until the activation site recovers.

Avoiding multiple simultaneous activations of a cluster is achieved by registering the activation site for the cluster with the Location Service (LS). Thus the design of the LS must cater for the possibility that, due to a site crash, an activated cluster may be lost from VOM or if not lost, because it will be recovered by the TS, may be unavailable while the site at which it was mapped is down. Moreover, the information maintained by the LS must be resilient to site failures. Thus, even if the activation site fails during an activation period while possibly holding new committed or prepared object states on its log, multiple activation is prevented by the LS. Moreover, if an activation site is registered in the LS, but does not hold the cluster after restart, it follows that all changes to the cluster have been aborted as a result of the site failure and that the SS holds the valid state of the cluster which can be safely reactivated. The distributed algorithm used to locate a cluster is, therefore, as follows (Figure 8). Note that an exception reported by the location algorithm may be mapped to a transaction abort.

[1] but not propagated (see section 6.2.3)
[2] or propagation

```
Assume the following functions are defined:

SiteState (site_name) -> (Up, Down)
ObjectState (object_name) -> (Activated, Passivated)
                     - as determined by the location service
MappingState (object_name) -> (Absent, Present)
                     - at a given site

The algorithm is as follows:

CASE SiteState (Control Site) OF
       Down:    either wait or report exception;
       Up:      CASE ObjectState (Object) OF
                Activated:    go to activation site;
                     CASE SiteState (Activation Site) OF
                          Down:    either wait or report exception;
                          Up:      CASE ObjectState (Object) OF
                                   Present: access object;
                                   Absent:  /*all changes aborted*/
                                        reactivate object;
                                        access object;
                Passivated:   activate object;
                     access object;
```

Figure 8: Location algorithm.

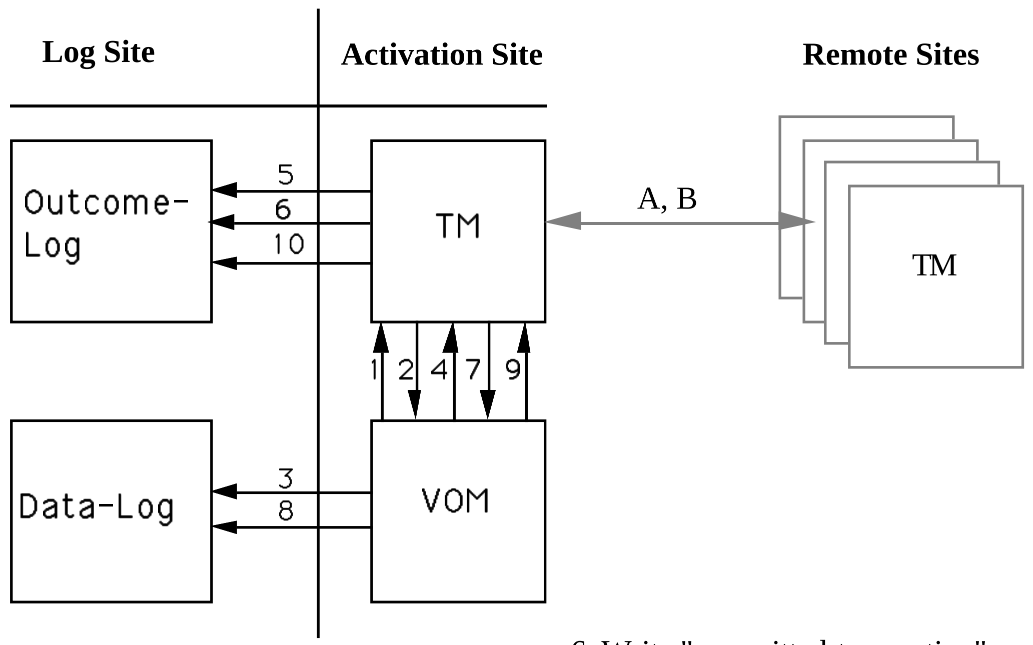
When activating a cluster it is necessary to reliably register the activation of the cluster in the LS before retrieving the cluster from the SS. If the cluster cannot be retrieved, because of a failure of the SS, then the activation of the cluster must be unregistered from the LS. When passivating a cluster, the cluster must be written to the SS before the activation is unregistered from the LS.

6.2.2 Committing Transactions

As described, every context maintains a data-log to store the after images of modified atomic objects in the first phase of the commit protocol. A context located on a diskless site is statically one-to-one associated with a log on a site acting as a log-server. In addition, the TM at each site maintains an outcome-log to keep track of the progress of commit protocol executions.

The operation of the commit protocol is described by figure 9, in which 'VOM' stands for the collection of contexts involved in a given commit request, and 'Data-Log' means the corresponding collection of data-logs, each of them associated with a particular context.

In the first phase of the commit protocol, the RM writes the after images of individual atomic objects which have been modified by the committing transaction to the data-log, together with a commit request identifier. Note that only modified atomic objects and not their containing clusters are written to the data-log. In addition, the context log-table describes on-going commit requests and the objects/clusters involved. In the second phase, a corresponding "commit-data" record is written to the data-log if the transaction commits, or, if the transaction aborts, a "discard-data" record is written.



1. Initiate commit;
2. Request after images be prepared;
3. Record prepared after images;
4. Announce that transaction is prepared;
5. Write "prepared-transaction" record;

A. execute distributed commit protocol

6. Write "committed-transaction" record;
7. Announce that transaction is committed;
8. Write "committed-data" record;
9. Acknowledge commitment;

B. execute distributed erasure protocol

10. Write "done" record;

Figure 9: Committing a Transaction.

6.2.3 Propagating clusters

Propagating a cluster entails writing its youngest committed state to the SS (without necessarily passivating the cluster). Propagation updates the stable state of the cluster in the SS thereby reducing the amount of recovery information that must be maintained in each context to recreate the youngest committed state from the stable state. Thus, propagating a cluster without passivating it, is solely for the purpose of releasing log records for storage reclamation reasons and reducing restart time. A cluster may be propagated several times during one activation period and each propagation may include the effects of several transactions.

To propagate a cluster the contiguous committed image of the cluster has to be constructed in the context by assembling the youngest committed state of each of the cluster's constituent objects. In general, this need not be the object's actual state if the object has been modified by an on-going transaction, but can be taken from a recovery point.

After writing back a cluster to the SS, a "propagated" record is forced to the log, which gives the identifier of the last commit request which committed changes to the cluster. This information is used during restart to determine which log-entry information for this cluster may be discarded, thus allowing commit processing to proceed concurrently with propagation. If writing the cluster back fails then no record is written to the log. The committed cluster (or one of its derivatives) will eventually be propagated.

6.2.4 Site Failure and Restart

A failure of an activation site will abort all transactions running at that site and will interrupt all on-going commit protocol executions at that site. Furthermore, clusters that were active at the site prior to the failure will be unavailable while the site is down.

On restart of the activation site, the Amadeus server restarts the TM which in turn restarts each context that existed before the crash. For this purpose the TM must maintain a list of the contexts that are active at the site as well as sufficient information to locate the data-log for each context. In addition, a mapping from context to text image must be maintained so that each context knows which text image to 'exec' when restarting. This mapping is maintained in the context data-log (i.e. the name of the context's text image is recorded in the context's data-log). If the activation site is diskless, restart will be delayed until its log-server site is available. Commit protocol executions interrupted by a site failure are then rescheduled as described in [Schumann et al. 1989].

During restart each context scans its data-log and rebuilds its log-table in virtual memory. Clusters for which no committed entries are found or whose last valid entry is "propagated" are ignored. Thus, after restart, the log-table indicates all clusters to which changes were committed but not propagated. These (and possibly more) clusters are already registered in the LS as being active in the restarting site. The required clusters are then retrieved from the SS. If a cluster is unavailable (because its storage site is down) then restart must be delayed until the cluster is again available.

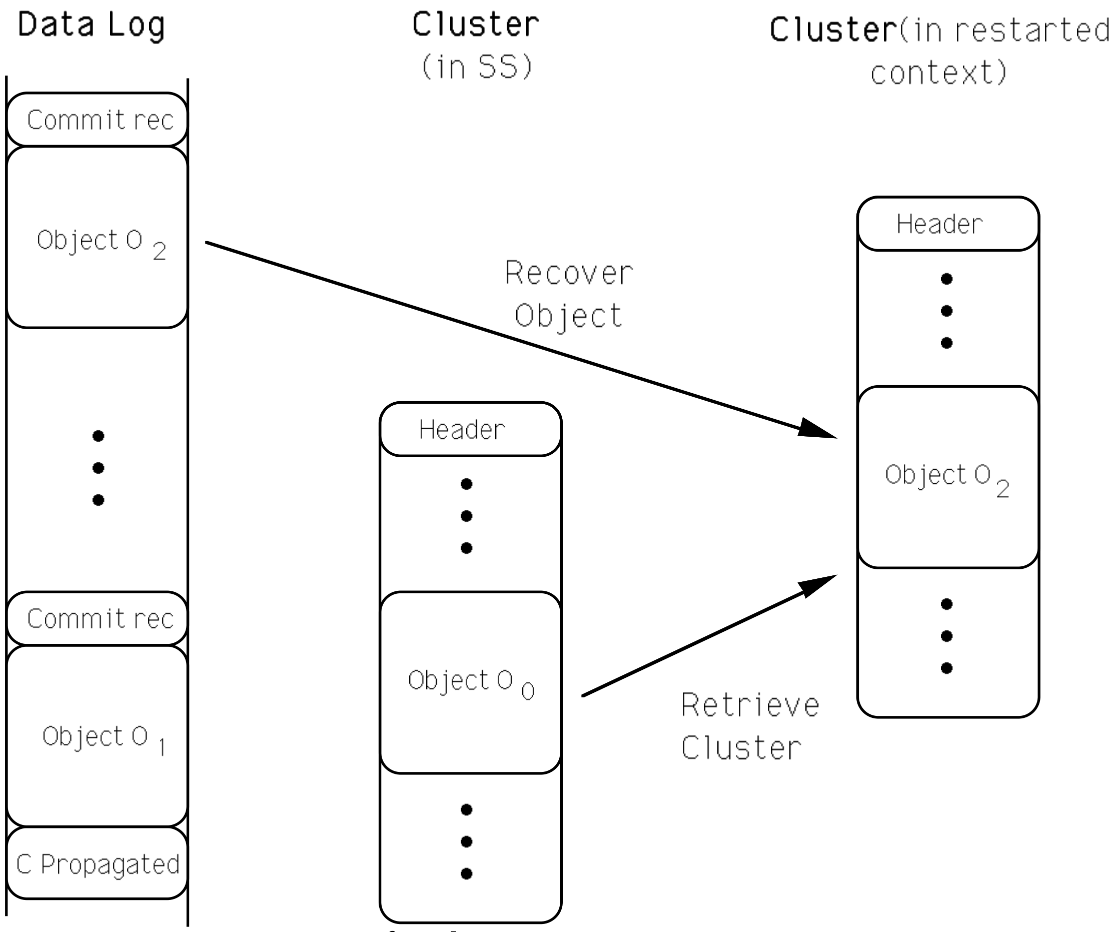


Figure 10: Restart Processing for Clusters.

The youngest committed state of each cluster is constructed from the stable state obtained from the SS by overwriting the actual state of each object, to which changes were committed since the cluster was last propagated, with the youngest committed state of the object found on the data-log. Once a context has been completely reconstructed, each cluster is propagated and a "propagated" record written to the data-log indicating that all objects corresponding to the commit requests being processed have been propagated. These objects may now be deleted from the log in order to reduce its the length. The clusters mapped into the context can now be made available to other applications.

Figure 10 shows restart processing for object O from cluster C which has been modified by two committed transactions since C was last propagated (O_0, O_1 and O_2 denote sucessive states of O).

6.3 Local Algorithms

This section discusses the role of Amadeus in supporting recovery control and commitment for atomic objects.

6.3.1 Basic Recovery Point Operations

A recovery point must be created before the first operation that modifies an atomic object's state within any transaction so that, in the event of a transaction abort, the previous state of the object can be restored. A recovery point may also be used during the prepare phase of the two phase commit protocol.

Every object that is mapped into virtual memory is represented by its actual state and associated recovery points (if any). All accesses are directed to the actual state of the object which always remains in the same place and never changes size. Each object consists of a header (maintained by the GRT) followed immediately by the object's data. The object's header contains information such as the object's size and the cluster to which the object belongs.

When the RM-library is informed about an attempt to invoke an operation on an object that will modify the object's state it calls the Amadeus `VOM_Save` routine which creates a recovery point for the object and returns the identifier (address) of the new recovery point. The parameters to `VOM_Save` include the resource identifier (address) of the object and a flag indicating how to create the recovery point. Normally this flag indicates that a value based recovery point should be made. Making such a recovery point involves copying the target object's actual state. To support garbage collection, the entire object including its header must be saved in the recovery point. This method of creating recovery points assumes that objects are contiguous in memory and prohibits saving fragments of objects.

A transaction abort means that all objects modified by a transaction must revert to their original state. Here, the Amadeus `VOM_Restore` routine is called with a list of all objects to be restored and their associated recovery points. The recovery points are copied back to overwrite the actual state of the modified objects and then the recovery points deleted. When copying a recovery point back to overwrite the actual object state, care must be taken to ensure that runtime data structures are kept consistent. Thus, it may be necessary to update some fields in the object's header after the recovery point has been copied back.

After a transaction has committed, all recovery points for that transaction may be deleted. Here, the Amadeus `VOM_Delete` routine is called with a list of recovery points that are no longer required. Each recovery point in the list is deleted by freeing the memory allocated for the recovery point.

6.3.2 Object Creation and Garbage Collection

If an atomic object is created during a transaction, a recovery point for the object must be made so that if the creating transaction aborts the object can be deleted. In this case, `VOM_Save` is called with an indication that a creation operation is taking place. Since the only state to be recorded is the fact that the object exists, recovery points for newly created atomic objects are empty.

On a transaction abort, `VOM_Restore` will be called for the new object. The recovery point for the object will identify it as newly created and the object will simply be deleted.

Note that if a reference to a newly created atomic object is stored in a non-atomic object and the atomic object is deleted as the result of a transaction abort, then the reference in the non-atomic object will refer to a nonexistent object. Attempts to access such nonexistent objects are trapped by the GRT and an exception raised.

The runtime garbage collector works by scanning all objects mapped in the current context. To do so, it maintains a table of all mapped objects. Since recovery points need to be scanned, they must also be represented in the table. It is for this reason that the complete object must be saved when making a recovery point (i.e. a recovery point must appear as a normal object to the runtime). However, since a recovery point has the same object identifier as the original object, they cannot be placed in the same table. So a new table which lists all recovery point objects is required by the runtime. Thus, `VOM_Save` must be extended to insert the object represented by the recovery point into the table and `VOM_Restore` must be extended to remove the entry from the table.

An object can only be deleted by the runtime garbage collector when it is found that there are no references to the object, either on the stack or in other objects or recovery points. If an atomic object is not referenced, then it is safe to delete the object without having to make a recovery point for the deletion. That is, once all references to an atomic object have been lost, there is no possibility of any of those references coming back into existence as the result of a transaction abort. Furthermore, on a restart there is no necessity to redo the deletion since the object is garbage and will be deleted later. Therefore, there is no need to make recovery points for deleted objects.

6.3.3 Two-Phase-Update Support

During the prepare phase of the commit protocol modified atomic objects are written to the data-log. Since this log is used to rebuild the context on a restart, its entries cannot contain any virtual memory addresses since this would require that the context is reconstructed with exactly the same layout as existed before the crash. Normally, when an object is stored in a cluster in the SS, each object reference within the object is converted to an offset within the cluster (This process is called *folding*; the opposite process of converting from offset pointers to language specific pointers is called *unfolding*.). This offset either points directly to the referenced object if it is located in the same cluster or to a *stub* for the referenced object if it is not located in the same cluster. A stub contains enough information to locate the referenced object. Each object in a cluster has enough space allocated immediately after its data for all the stubs that the object might require. When an object is written to the data-log, each reference in the object is converted to an offset. However, since just objects and not clusters are saved to the log, a stub is required for each reference, whether or not the object referenced is in the same cluster as the object being written to the log.

Writing the object to the data log is done in response to a call to `VOM_StableSave` which is passed a list of all the objects to be saved and associates an identifier with the commit request. Each object in the list is copied to a spare region of memory where all references to other objects are folded and any necessary stubs are generated and appended to the object. Then the object is forced to the data-log. In order to reduce the number of writes to the log, a number of objects may be buffered together and written in a single operation. Note that when, during restart, an object is restored from the log any references in the object do not need to be unfolded. This will be done automatically by the runtime if necessary when the object is first used.

During the second phase of the commit protocol, the commit request is either committed or discarded. If the commit request succeeds a call to `VOM_Commit` writes a record to the data log indicating that all objects associated with the commit request are now committed. Later, once the corresponding clusters have been propagated, these entries in the log can be deleted. If the commit request fails a call to `VOM_Discard` writes a corresponding record to the data-log. All entries in the log belonging to this commit request may be deleted (either immediately or at some later stage).

Figure 11 summarizes the recovery point management and logging operations provided by Amadeus.

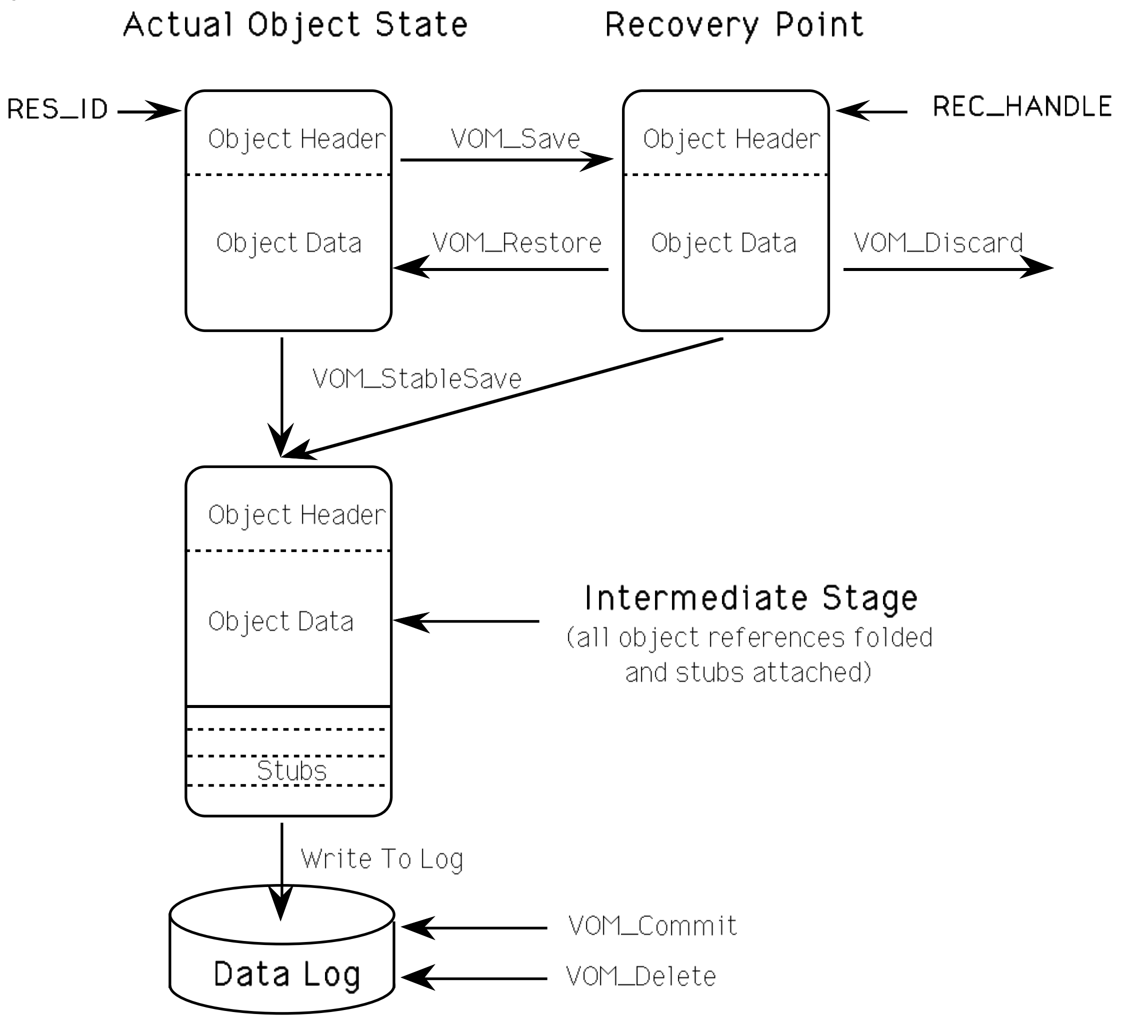


Figure 11: Amadeus Recovery Point and Logging Operations.

6.4 Access Trapping

One of the main requirements on Amadeus is that all attempts to access atomic objects must be trapped and reported to the recovery and concurrency control module so that the appropriate locks can be acquired and a recovery point made if necessary. As described previously, trapping accesses to atomic objects is a function of the language (specific runtime) which can report such accesses via the extended GRT interface described in section 6.1.1. In this section we describe the mechanism used to trap accesses to atomic objects in our current C** implementation. Different mechanisms may be appropriate for other languages.

In designing a method for trapping access to atomic objects we were concerned to minimize the performance impact for access to non-atomic objects. Moreover since, in the general case, the client of an object does not know in advance whether that object is atomic or not, the possibility of inserting the necessary tests into client code was ruled out.

The mechanism chosen is based on the use of different versions of the class code for atomic and non-atomic instances of a class, i.e. while both atomic and non-atomic instances of the same class may exist they are bound to different versions of the class code. The atomic version of each operation calls the TS to perform the necessary checks before calling the operation. Using this mechanism the client is not aware of the kind of object to which it is talking. When the target object is non-atomic no overhead is incurred either in testing for locks/recovery points nor even in testing for atomicity.

Currently, in the absence of a general dynamic linking facility, our mechanism requires that all invocations are indirect via a function table (i.e. similar to virtual functions in C++ [Ellis & Stroustrop. 1990]). Moreover, this approach restricts access to objects to being through functions, i.e. it is not possible to read or write the instance data of an object directly. Of course this mechanism also requires that two versions of each class are produced by the compiler/preprocessor.

When a non-atomic object is promoted to being an atomic, the function table pointer used by the object is modified to point to the corresponding function table for the atomic class code. In the atomic class code, the operation is forwarded to the real code using the original function table pointer which is saved within the object.

A class may also contain embedded objects and accesses to these objects must also be trapped if the enclosing object is atomic. However this can only be done if the embedded object is also atomic. During the production of the atomic class code, any embedded class is replaced by an atomic version of that class. When an object is made atomic, all embedded objects are also made atomic.

Since embedded objects do not have a runtime header, when an access to an embedded object is trapped, the header of the enclosing object must first be located in order to lock the object and to perform recovery control operations. This means that accessing an embedded object results in the complete enclosing object being locked and if necessary a recovery point for the enclosing object being made. This is a consequence of the fact that information about the size of the embedded object is not available at runtime.

7 Implementation and Performance

The RelaX transaction facility and the Amadeus system currently run on networks of DECStations, Sun-3 and Sun-4 workstations running Ultrix or SunOS. In addition, prototype implementations of Amadeus are currently running on Chorus/MIX and the Mach 3.0 BSD single server. An implementation of Amadeus on the Chorus microkernel is underway. Both systems are written in C and C++, comprising approximately 180,000 lines of code each. The integration of the systems, i.e. the implementation of the algorithms described in section 6, took approximately 6 person-months of effort.

The Amadeus server and all contexts, the TM and the RBP are currently implemented as separate UNIX processes that communicate locally over TCP/IP stream connections (refer to Figure 6). The commit protocol executed by the TM relies on the RBP which uses UDP/IP datagram broadcasts.

The following table (Figure 12) give some initial performance data for the system running on three DECStation 2100s connected by a 10Mbit/sec Ethernet.

Read + Write one object per site	**1 Site**	**2 Sites**	**3 Sites**
Normal Invocations	0,8 ms	25 ms	49 ms
Completing Transactions	185 ms	288 ms	566 ms
Committing Transactions	308 ms	489 ms	826 ms
Group Committment	224 ms	327 ms	734 ms

Figure 12: Elapsed execution time in milliseconds.

The table illustrate the average execution time (elapsed time) of ten sequential transactions each reading and wrtiting a single atomic object per site. The columms vary in the number of sites involved, i.e., columm one describes the performance of local transactions, columm two and three that of distributed transactions running on two and three sites respectively. Row one describes the performance of non-transactional reads, thus giving the lower bound for transactions. Note that non-transaction write access to atomic objects is excluded by the transactional model (see section 4). Row two describes complete-only transactions that delivered their results on termination without waiting for a commit protocol execution. These transactions are committed eventually in a background commit. In row three, the transactions committed individually meaning that their elapsed time includes a (distributed) commit protocol execution. Finally, row four gives the average of nine comlpete-only transactions followed by a tenth transaction which initiated the group commitment of all ten.

The difference between the time for a complete-only transaction (row two) and its corresponding committed transaction (row three, individual commit) shows the overhead of the commit protocol execution for the local case (approxiamately 123 msecs) and the distributed case (approxiamately 201 msecs for two sites, approxiamately 260 msecs for three sites). The group commitment of row four further reduces the average overhead for the commit protocol execution compared to the complete-only case (row two).

8 Summary and Conclusion

The Amadeus/RelaX system shows that it is possible to support atomic objects in a language independent manner. Moreover such a system can give full support for atomic objects while being open to supporting additional resource types. Use of the RelaX transaction facility has enabled the provision of a flexible transaction model and support for the integration of multiple resource types. The ease of integration of Amadeus and RelaX shows the utility of breaking transaction functionality down into an extensible architecture. Furthermore, the generic transaction support components have proven to be very useful and time-saving for the implementation of atomic resources.

The overall functionality of the integrated system is comprehensive and its performance, given an underlying UNIX kernel and no attempt for optimization, is acceptable. System performance and behaviour will be evaluated in greater detail using a distributed measurement environment [Lange et al. 1992]. Further work is on-going to support more languages and to integrate existing relational database systems. Functional extensions towards general purpose fault tolerance schemes [Nett 1991] including support for replicated objects are envisaged.

Acknowledgements

The authors gratefully acknowledge the input of all those who have contributed to the Amadeus and RelaX systems and their integration: Sean Baker, Neville Harris, Chris Horn, Andre Kramer, John Moreau, Gradimir Starovic and Bridget Walsh at TCD; Friedhelm Baer, Rainer Frings, Rolf Heinl, Susanne Kerp, Frank Lange, Edgar Nett, Gerd Rademann, Andreas Schneider, Ralf Schumann, Rosemarie Vonthin and Beatrice Wirz at GMD. The work of Paul Taylor at TCD and GMD is particularly acknowledged.

We are also gratefull to our many colleages with whom we have collaborated in the course of the Comandos project including those at Bull, Chorus, IMAG and the OSF in France, INESC in Portugal, the Universtiy of Glasgow in the United Kingdom, UPC in Spain and the Fraunhofer Institute in Germany.

References

T. Andrews, C. Harris and K. Sinkel. *ONTOS: A Persistent Database for C++.* in Object-Oriented Databases with Applications to CASE, Networks and VLSI CAD. Prentice Hall, 1991

V. Cahill, C. Horn and G. Starovic. Towards Generic Support for Distributed Information Systems. *International Workshop on Object Orientation in Operating Systems*, 104-107, Palo Alto, Cal., 1991

J. Chang and N. Maxemchuk. “Reliable Broadcast Protocols.” *ACM Transactions on Computer Systems* **2**(3): 251-273, 1984

G. N. Dixon, G. D. Parrington, S. K. Shrivastava and S. M. Wheater:. “The Treatment of Persistent Objects in Arjuna.” *The Computer Journal* **32**(4): 1989

M. A. Ellis and B. Stroustrop. *The Annotated C++ Reference Manual*. Addison Wesley, 1990

J. L. Eppinger and A. Z. Spector. A Camelot Perspective. UNIX Review. 1989

K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger:. “On the Notions of Consistency and Predicate Locks.” *Communications of the ACM* **19**(11): 1976

R. Haskin, Y. Malachi, W. Sawdon and F. Chan. “Recovery Management in Quicksilver.” *ACM Transactions on Computer Systems* **6**(1): 82-108, 1988

M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. *Fault-Tolerant Computing Systems-17,* 1987

C. Horn and V. Cahill. “Supporting Distributed Applications in the Amadeus Environment.” *Computer Communications Review* **14**(6): 1991

J. Kaiser. “MUTABOR, A Coprocessor Supporting Memory Management in an Object-Oriented Architecture.” *IEEE Micro* **8**(5): 30-46, 1988

R. Kroeger and E. Nett. System-Level Support for Dependable Distributed Applications. *Int. Workshop on Operating Systems of the 90s and Beyond*, Schloß Dagstuhl, Germany, 1991

R. Kröger, M. Mock, R. Schumann and F. Lange. RelaX - An Extensible Architecture Supporting Reliable Distributed Applications. *9th Symposium on Reliable Distributed Systems*, 156-164, Huntsville, Alabama, 1990

F. Lange, R. Kroeger and M. Gergeleit. “JEWEL: Design and Implementation of a Distributed Measurment System.” *IEEE Transactions on Parallel and Distributed Systems* (Spelial Issue on Measurement and Evaluations of Parallel and Distributed Systems): 1992

B. Liskov and R. Scheifler. “Guardians and Actions: Linguistic Support for Robust Distributed Programs.” *ACM Transactions on Programming Languages and Systems* **5**(3): 381-404, 1983

B. Mayer. *Object Oriented Software Construction*. Prentice-Hall, 1988

J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Programming*. Ph.D. Thesis, MIT/LCS/TR-260, 1981

E. Nett. *Supporting Fault Tolerant Distributed Computations*. Habil. Thesis, Bonn, 1991

E. Nett, K.-E. Grosspietsch, A. Jungblut, J. Kaiser, R. Kröger, W. Lux, M. Speicher and H.-W. Winnebeck. *PROFEMO - Design and Implementation of a Fault Tolerant Distributed System Architecture*. GMD-Studie 100, GMD, 1985

E. Nett, J. Kaiser and R. Kröger. Providing Recoverability in a Transaction Oriented Distributed System. *6th Int. Conf. on Distributed Computing Systems*, Cambridge, Mass., 1986

G. D. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. *2nd Usenix C++ Conference*, San Francisco, 1990

F. Schmuck and J. Wyllie. Experiences with Transactions in Quicksilver. *13th ACM Symposium on Operating System Principles*, 239-253, Pacific Grove, CA, 1991

R. Schumann, R. Kröger and M. Mock. The Decentralized Non-Blocking RelaX Commit Protocol. *11th ITG/GI-Fachtagung Architektur von Rechensystemen*, 403-413, Muenchen, 1990

R. Schumann, R. Kröger, M. Mock and E. Nett. Recovery Management in the RelaX Distributed Transaction Layer. *8th Symp. on Reliable Distributed Systems*, Seattle, 1989

R. Schumann and M. Mock. Efficient Commit/Abort Procedures in the RelaX Distributed Transaction Layer. *Fault-Tolerant Computing Systems*, 209-220, Baden-Baden, 1989

Transarc Corperation. Encina - Enterprice Computing in a New Age, Product Overview. 1991

UNIX System Laboratories. “The Tuxedo System, Product Overview.” : 1991

R. Vonthin. *Spezifikation des PROFEMO-Reliable Broadcast Protokolls in Unix 4.2 BSD*. GMD-Studie 127, GMD, 1987

W. Weihl. “Implementation of Resilient Atomic Data Types.” *ACM Transactions on Programming Languages and Systems* **7**(2): 1985

X/Open Company. Distributed Transaction Processing Reference Model: The XA Specification. *Berkshire*, X/Open Company Limited. 1991