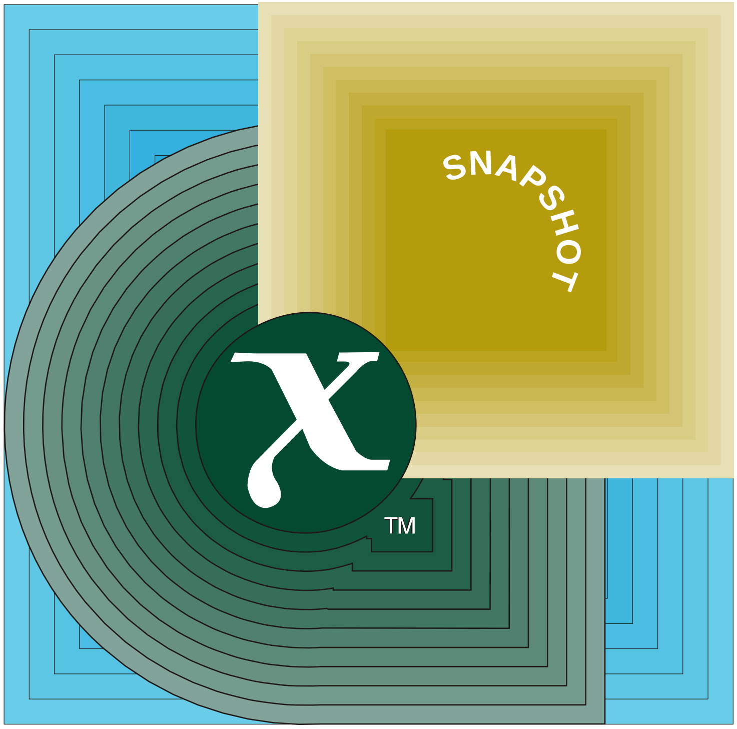
Snapshot

Distributed Transaction Processing: The XA+ Specification Version 2



THE Open GROUP

[This page intentionally left blank]

Snapshot

Distributed Transaction Processing: The XA+ Specification
Version 2

The Open Group

© *June 1994, The Open Group*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Snapshot

Distributed Transaction Processing: The XA+ Specification Version 2

ISBN: 1-85912-046-6
Document Number: S423

Published in the U.K. by The Open Group, June 1994.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is an international open systems organization that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organizations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organizations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This document is a Snapshot (see above), which is an updated, working version of the CAE Specification, **Distributed Transaction Processing: The XA Specification** issued in December, 1991. The XA interface is the bidirectional interface between a transaction manager and resource managers. This specification reflects work in progress to extend the XA interface to support communication resource managers.

The extensions allow a communication resource manager to create transaction branches, to propagate transaction branches to a subordinate component consisting of an application program, a transaction manager, and one or more resource managers, and to propagate the transaction completion protocol to a subordinate transaction manager.

This specification is structured as follows:

- Chapter 1 is an introduction.
- Chapter 2 provides fundamental definitions for the remainder of the document.
- Chapter 3 is an overview of the XA interface.
- Chapter 4 discusses the data structures that are part of the XA interface.
- Chapter 5 contains reference manual pages for each routine in the XA interface.
- Chapter 6 contains state tables.
- Chapter 7 summarises the implementation requirements and identifies optional features.
- Appendix A is the code of the header file required by XA routines.
- Appendix B provides some example scenarios.

There is an index at the end.

Intended Audience

This document is of interest to implementors of the XA interface including transaction manager implementors, resource manager implementors, and communication resource manager implementors.

All readers are expected to be familiar with the X/Open documents **Distributed Transaction Processing: Reference Model, Version 2** and **Distributed Transaction Processing: The TX (Transaction Demarcation) Specification**.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names, for example, substitutable argument prototypes and environment variables
  - — C-language functions; these are shown as follows: *name*( )
- Normal font is used for the names of constants and literals.

- The notation **<file.h>** indicates a C-language header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values, which may be declared in appropriate C-language header files by means of the C **#define** construct.
- The notation [ABCD] is used to identify a coded return value.
- Syntax and code examples are shown in `fixed width` font.
- Variables within syntax statements are shown in *`italic fixed width`* font.

Superseded Documents

This Version 2 Snapshot supersedes the previous version of the X/Open **Distributed Transaction Processing XA+ Specification** published in 1993. Since that version, X/Open has added the Transaction Manager Switch, added facilities to pass timeout values, and aligned the introductory chapters of the specification with the revised guide: **Distributed Transaction Processing: Reference Model, Version 2**

Trademarks

Motif®, OSF/1® and UNIX® are registered trademarks and the ‘‘X Device’’™ and The Open Group™ are trademarks of The Open Group.

Referenced Documents

The following standards are referenced in this specification:

ASN.1
: ISO 8824: 1987, Information Processing Systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).

ASN.1 Extensions
: ISO / IEC 8824 / DAD 1: 1988, Information Processing Systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1), Addendum 1:ASN.1 Extensions.

BER
: ISO 8825: 1987, Information Processing Systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

ISO C
: ISO / IEC 9899: 1990: Programming Languages — C, including:
Amendment 1: 1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

OSI CCR

ISO / IEC 9804
: ISO / IEC 9804: 1990, Information Technology — Open Systems Interconnection — Service Definition for the Commitment, Concurrency, and Recovery Service Element, together with:

Technical Corrigendum 1: 1991 to ISO / IEC 9804: 1990
Amendment 2: 1992 to ISO / IEC 9804: 1990 Session mapping changes.

ISO / IEC 9805
: ISO / IEC 9805: 1990, Information Technology — Open Systems Interconnection — Protocol Specification for the Commitment, Concurrency, and Recovery Service Element, together with:

Technical Corrigendum 1: 1991 to ISO / IEC 9805: 1990
Technical Corrigendum 2: 1992 to ISO / IEC 9805: 1990
Amendment 2: 1992 to ISO / IEC 9805: 1990 Session mapping changes.

OSI TP Model
: ISO / IEC 10026-1: 1992, Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 1: OSI TP Model.

OSI TP Service
: ISO / IEC 10026-2: 1992, Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 2: OSI TP Service.

OSI TP Protocol
: ISO / IEC 10026-3: 1992, Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 3: Protocol Specification.

The following X / Open documents are referenced in this specification:

CPI-C
: Forthcoming X / Open Specification, expected to be published in 1994, Common Programming Interface Communications Specification. This document is expected to have

the same technical content as the CPI-C Implementors' Workshop specification of the same title.

This information is currently documented in the X/Open CAE Specification, February 1992, CPI-C (ISBN: 1-872630-35-9, C210) and the X/Open Snapshot, December 1992, Distributed Transaction Processing: The Peer-to-Peer Specification (ISBN: 1-872630-79-0, S214).

DTP
: Guide, February 1996,, Distributed Transaction Processing: Reference Model, Version 3 (ISBN: 1-85912-170-5, G504).

Peer-to-Peer
: Snapshot, December 1992, Distributed Transaction Processing: The Peer-to-Peer Specification (ISBN: 1-872630-79-0, S214).

SQL
: CAE Specification, August 1992, Structured Query Language (SQL) (ISBN: 1-872630-58-8, C201).

TX
: CAE Specification, April 1995, Distributed Transaction Processing: The TX (Transaction Demarcation) Specification (ISBN: 1-85912-094-6, C504).

TxRPC
: Preliminary Specification, July 1993, Distributed Transaction Processing: The TxRPC Specification (ISBN: 1-85912-000-8, P305).

XA
: CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification (ISBN: 1-872630-24-3, C193).

XAP-TP
: The XAP-TP Transaction Processing API is a forthcoming X/Open Preliminary Specification that is expected to be published in 1993 (ISBN: 1-872630-8-55, P216)

XATMI
: Preliminary Specification, July 1993, Distributed Transaction Processing: The XATMI Specification (ISBN: 1-872630-99-5, P306).

Chapter 1

Introduction

This document represents the interim results of X/Open's technical activity to extend the *XA* interface, principally to address the requirements of communication resource managers. The extended XA interface includes all of the capability described in the referenced **XA** specification and additional functions. X/Open technical activity may result in a new **XA** specification based on this snapshot. In this document, all references to the XA interface mean the extended interface described herein.

1.1 X/Open DTP Model

The X/Open Distributed Transaction Processing (DTP) model is a software architecture that allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions.

The X/Open DTP model comprises five basic functional components:

- an Application Program (AP), which defines transaction boundaries and specifies actions that constitute a transaction
- Resource Managers (RMs) such as databases or file access systems, which provide access to resources
- a Transaction Manager (TM), which assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for coordinating failure recovery
- Communication Resource Managers (CRMs), which control communication between distributed applications within or across TM domains
- a communication protocol, which provides the underlying communication services used by distributed applications and supported by CRMs.

X/Open DTP publications based on this model specify portable Application Programming Interfaces (APIs) and system-level interfaces that facilitate:

- portability of application program source code to any X/Open environment that offers those APIs
- interchangeability of TMs, RMs and CRMs from various sources
- interoperability of diverse TMs, RMs and CRMs in the same global transaction.

Chapter 2 defines each component in more detail and illustrates the flow of control.

1.2 XA Interface

The XA interface is the bidirectional interface between a transaction manager and a resource manager. The XA interface is not an ordinary Application Programming Interface (API). It is a system-level interface between DTP software components. This specification addresses the model presented in Section 2.1 on page 3, and discusses aspects of the model that pertain to a resource manager, such as a communication resource manager, acting as a superior as well as a subordinate to the transaction manager. X/Open anticipates that heterogeneous TMs will use a communication resource manager (by means of the extended XA specification) for communication of DTP information and application data. Such communication involves extensions to the **XA** specification and involves a more detailed explanation of the DTP model.

Other DTP interfaces for direct use by an application program are the subject of other publications (see Section 2.1 on page 3 for an overview).

Relevant definitions and other important concepts are discussed in Chapter 2. This chapter also defines the AP, TM and RM in more detail, and describes their interaction. For an overview of the XA interface, describing the situations in which each of the services is used, refer to Chapter 3. The data structures that are part of the XA interface are discussed in Chapter 4. Reference manual pages for each function in the XA interface are presented in Chapter 5; state tables follow in Chapter 6. For information about the implications of this specification on the implementors of RMs and TMs refer to Chapter 7, which also identifies features that are optional. The contents of an **<xa.h>** header file, in both ISO C and Common Usage C, are given in Appendix A. There are some example scenarios in Appendix B.

Chapter 2

Model and Definitions

This chapter discusses the XA interface in general terms and provides necessary background material for the rest of the specification. The chapter shows the relationship of the interface to the X/Open DTP model. The chapter also states the design assumptions that the interface uses and shows how the interface addresses common DTP concepts.

2.1 X/Open DTP Model

The boxes in the figure below are the functional components and the connecting lines are the interfaces between them. The arrows indicate the directions in which control may flow.

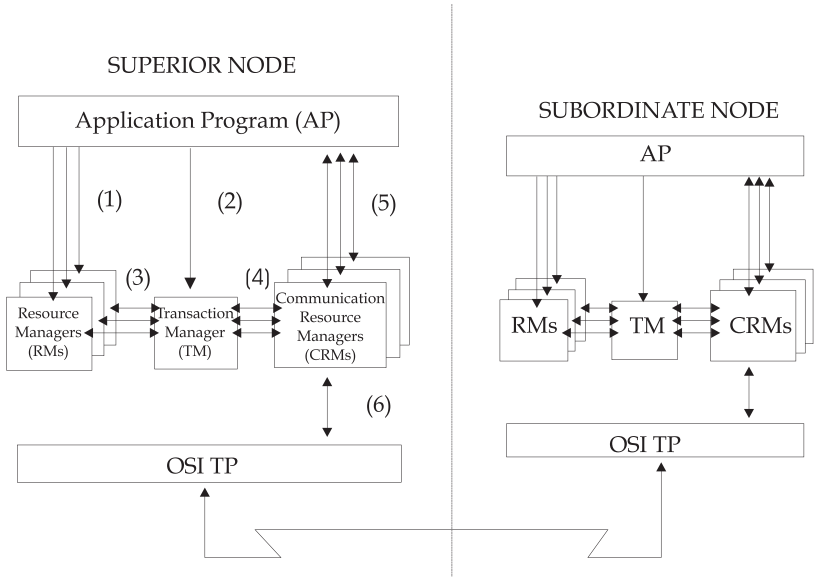


Figure 2-1 Functional Components and Interfaces

Descriptions of the functional components shown can be found in Section 2.1.1 on page 4. The numbers in brackets in the above figure represent the different X/Open interfaces that are used in the model. They are described in Section 2.1.2 on page 5.

For more details on this model and diagram, including detailed definitions of each component, see the referenced **DTP** guide.

2.1.1 Functional Components

Application Program (AP)

The application program (AP) implements the desired function of the end-user enterprise. Each AP specifies a sequence of operations that involves resources such as databases. An AP defines the start and end of global transactions, accesses resources within transaction boundaries, and normally makes the decision whether to commit or roll back each transaction.

Where two or more APs cooperate within a global transaction, the X/Open DTP model supports three *paradigms* for AP to AP communication. These are the TxRPC, XATMI and Peer-to-Peer interfaces.

Transaction Manager (TM)

TMs manage global transactions, coordinate the decision to commit them or roll them back, and coordinate failure recovery. The AP defines the start and end of a global transaction by calling a TM. The TM assigns an identifier (XID) to the global transaction (see Section 2.2.6 on page 8). The TM manages global transactions and informs each RM of the XID on behalf of which the RM is doing work. Although RMs can manage their own recoverable work units as they see fit, each RM must accept XIDs and associate them with those work units. In this way, an RM knows what recoverable work units to complete when the TM completes a global transaction. If the RM is a Communication Resource Manager (CRM), it passes XIDs to partner, subordinate CRMs when performing communication between two APs.

Resource Manager (RM)

The resource manager (RM) manages a defined part of the computer's shared resources. These may be accessed using services that the RM provides. Examples for RMs are database management systems (DBMSs), a file access method such as X/Open ISAM, and a print server.

In the X/Open DTP model, RMs structure all changes to the resources they manage as recoverable and atomic transactions. They let the TM coordinate completion of these transactions atomically with work done by other RMs.

A single RM may divide its resources into separate resource partitions called *resource domains*. If these resource domains need and support independent transaction completion, the concept of *RM instances* is used. An RM instance is known to the TM by the address of its **xa_switch_t** structure (see Section 4.4 on page 29). Multiple RM instances may share the same **xa_switch_t** structure; but its address is made known to the TM multiple times. (See also Section 3.2 on page 16.) Unless specified otherwise, operations this specification allows on an RM are allowed on each RM instance.

Communication Resource Manager (CRM)

A CRM allows an instance of the model to access another instance either inside or outside the current TM Domain. Within the X/Open DTP model, CRMs use OSI TP services to provide a communication layer across TM Domains. CRMs aid global transactions by supporting the following interfaces:

- the communication paradigm (TxRPC, XATMI or Peer-to-Peer) used between an AP and CRM
- XA+ communication between a TM and CRM
- XAP-TP communication between a CRM and OSI TP.

A CRM may support more than one type of communication paradigm, or a TM Domain may use different CRMs to support different paradigms. The XA+ interface provides global transaction information across different instances and TM Domains. The CRM allows a global transaction to extend to another TM Domain, and allows TMs to coordinate global transaction commit and abort requests from (usually) the superior AP. Using the above interfaces, information flows from superior to subordinate and *vice versa*.

2.1.2 Interfaces between Functional Components

There are six interfaces between software components in the X/Open DTP model. The numbers correspond to the numbers in Figure 2-1 on page 3.

(1) **AP-RM.** The AP-RM interfaces give the AP access to resources. X/Open interfaces, such as SQL and ISAM, provide AP portability. The X/Open DTP model imposes few constraints on native RM APIs. The constraints involve only those native RM interfaces that define transactions. (See the referenced **XA** specification.)

(2) **AP-TM.** The AP-TM interface (the TX interface) provides the AP with an Application Programming Interface (API) by which the AP coordinates global transaction management with the TM. For example, when the AP calls *tx_begin*( ) the TM informs the participating RMs of the start of a global transaction. After each request is completed, the TM provides a return value to the AP reporting back the success or otherwise of the TX call.

For details of the AP-TM interface, see the referenced **TX** specification.

(3) **TM-RM.** The TM-RM interface (the XA interface) lets the TM structure the work of RMs into global transactions and coordinate completion or recovery. The XA interface is the bidirectional interface between the TM and RM.

The functions that each RM provides for the TM are called the *xa_**( ) functions. For example, the TM calls *xa_start*( ) in each participating RM to start an RM-internal transaction as part of a new global transaction. Later, the TM may call in sequence *xa_end*( ), *xa_prepare*( ) and *xa_commit*( ) to coordinate a (successful in this case) two-phase commit protocol. The functions that the TM provides for each RM are called the *ax_**( ) functions. For example, an RM calls *ax_reg*( ) to register dynamically with the TM.

For details of the TM-RM interface, see the referenced **XA** specification.

(4) **TM-CRM.** The TM-CRM interface (the XA+ interface) supports global transaction information flow across TM Domains. In particular TMs can instruct CRMs by use of *xa_**( ) function calls to suspend or complete transaction branches, and to propagate global transaction commitment protocols to other transaction branches. CRMs pass information to TMs in subordinate branches by use of *ax_**( ) function calls. CRMs also use *ax_**( ) function calls to request the TM to create subordinate transaction branches, to save and retrieve recovery information, and to inform the TM of the start and end of blocking conditions.

The XA+ interface is a superset of the XA interface and supersedes its purpose. Since the XA+ interface is invisible to the AP, the TM and CRM may use other methods to interconnect without affecting application portability.

(5) **AP-CRM.** X/Open provides portable APIs for DTP communication between APs within a global transaction. The API chosen can significantly influence (and may indeed be fundamental to) the whole architecture of the application. For this reason, these APIs are frequently referred to in this document and elsewhere as *communication paradigms*. In practice, each paradigm has unique strengths, so X/Open offers the following popular paradigms:

- the TxRPC interface (see the referenced **TxRPC** specification)
- the XATMI interface (see the referenced **XATMI** specification)
- the Peer-to-Peer interface (see the referenced **Peer-to-Peer** specification).

Note: The Peer-to-Peer interface is expected to be aligned with the referenced **CPI-C** specification.

X/Open interfaces, such as the CRM APIs listed above, provide application portability. The X/Open DTP model imposes few constraints on native CRM APIs.

(6) **CRM-OSI TP.** This interface (the XAP-TP interface) provides a programming interface between a CRM and Open Systems Interconnection Distributed Transaction Processing (OSI TP) services. XAP-TP interfaces with the OSI TP Service and the Presentation Layer of the seven-layer OSI model. X/Open has defined this interface to support portable implementations of application-specific OSI services. The use of OSI TP is mandatory for communication between heterogeneous TM domains. For details of this interface, see the referenced **XAP-TP** specification and the OSI TP standards.

2.2 Definitions

For additional definitions see the **DTP** guide.

2.2.1 Transaction

A transaction is a complete unit of work. It may comprise many computational tasks, which may include user interface, data retrieval and communication. A typical transaction modifies shared resources. (The OSI TP standard (model) defines transactions more precisely.)

Transactions must be able to be *rolled back*. A human user may roll back the transaction in response to a real-world event, such as a customer decision. A program can elect to roll back a transaction. For example, account number verification may fail or the account may fail a test of its balance. Transactions also roll back if a component of the system fails, keeping it from retrieving, communicating or storing data. Every DTP software component subject to transaction control must be able to undo its work in a transaction that is rolled back at any time.

When the system determines that a transaction can complete without failure of any kind, it *commits* the transaction. This means that changes to shared resources take permanent effect. Either commitment or rollback results in a consistent state. *Completion* means either commitment or rollback.

2.2.2 Transaction Properties

Transactions typically exhibit the following properties:

Atomicity This means that the results of the transaction's execution are either all committed or all rolled back.

Consistency This means that a completed transaction transforms a shared resource from one valid state to another valid state.

Isolation This means that changes to shared resources that a transaction effects do not become visible outside the transaction until the transaction commits.

Durability This means the changes that result from transaction commitment survive subsequent system or media failures.

These properties are known by their initials as the **ACID** properties. In the X/Open DTP model, the TM coordinates Atomicity at global level whilst each RM is responsible for the Atomicity, Consistency, Isolation and Durability of its resources.

2.2.3 Distributed Transaction Processing

Within the scope of this document, DTP systems are those where work in support of a single transaction may occur across RMs. This has several implications:

- The system must have a way to refer to a transaction that encompasses all work done anywhere in the system.
- The decision to commit or roll back a transaction must consider the status of work done anywhere on behalf of the transaction. The decision must have uniform effect throughout the DTP system.

Even though an RM may have an X/Open-compliant interface such as Structured Query Language (SQL), it must also address these two items to be useful in the DTP environment.

2.2.4 Global Transactions

Every RM in the DTP environment must support transactions as described in Section 2.2.1 on page 7. Many RMs already structure their work into recoverable units.

In the DTP environment, many RMs may operate in support of the same unit of work. This unit of work is a *global transaction*. For example, an AP might request updates to several different databases. Work occurring anywhere in the system must be committed atomically. Each RM must let the TM coordinate the RM's recoverable units of work that are part of a global transaction.

Commitment of an RM's internal work depends not only on whether its own operations can succeed, but also on operations occurring at other RMs, perhaps remotely. If any operation fails anywhere, every participating RM must roll back all operations it did on behalf of the global transaction. A given RM is typically unaware of the work that other RMs are doing. A TM informs each RM of the existence, and directs the completion, of global transactions. An RM is responsible for mapping its recoverable units of work to the global transaction.

2.2.5 Transaction Branches

A global transaction has one or more *transaction branches* (or *branches*). A branch is a part of the work in support of a global transaction for which the TM and the RM engage in a separate but coordinated transaction commitment protocol (see Section 2.3 on page 10). Each of the RM's internal units of work in support of a global transaction is part of exactly one branch.

A global transaction might have more than one branch when, for example, the AP uses a CRM to communicate with remote applications. The CRM asks the TM to create a new transaction branch prior to accessing a remote AP for the first time. Subsequent accesses to the same remote AP are typically done within the same transaction branch. Accesses to different remote APs are typically done in separate transaction branches.

After the TM begins the transaction commitment protocol, the RM receives no additional work to do on that transaction branch. The RM may receive additional work on behalf of the same transaction, from different branches. The different branches are related in that they must be completed atomically. However, the TM directs the commitment protocol for each branch separately. That is, an RM receives a separate commitment request for each branch.

Each transaction branch identifier (or XID — see Section 2.2.6) that the TM gives the RM identifies both a global transaction and a specific branch. The RM may use this information to optimise its use of shared resources and locks.

2.2.6 Data Interfaces

Transaction Identifier

A transaction identifier (XID) is a data structure that a TM assigns. It represents the unique relationship between an AP, the work it issues to RMs, and the global transaction which the TM manages on behalf of the AP.

The XID lets the TM track and coordinate all of the work associated with a global transaction. Each RM maps the XID to the RM-internal work it does for the global transaction. To ensure global uniqueness, the XID should contain *atomic action identifiers* as specified in the referenced OSI CCR standard. For more information on XIDs, see Section 3.4 on page 20. For details of the structure itself, see Section 4.2 on page 28.

XA Switch Structure

Each RM provides a set of pointers to the functions that the TM calls, in a data structure known as the XA Switch structure.

2.2.7 Thread of Control

A thread of control (or a *thread*) is the entity, with all its context, that is currently in control of a processor. The context may include locks on shared resources and open files. For portability reasons, the notion of thread of control must be common among the AP, TM and RM.

The thread concept is central to the TM's coordination of RMs. APs call RMs to request work, while TMs call RMs to delineate transaction branches. The way the RM knows that a given work request pertains to a given branch is that the AP and the TM both call it from *the same thread of control*. For example, an AP thread calls the TM to declare the start of a global transaction. The TM records this fact and informs RMs. After the AP regains control, it uses the native interface of one or more RMs to do work. The RM receives the calls from the AP and TM in the same thread of control.

Certain XA functions, therefore, must be called from a particular thread. The reference manual pages in Chapter 5 indicate which functions require this.

2.2.8 Tightly- and Loosely-coupled Threads

Many application threads of control can participate in a single global transaction. All the work done in these threads is atomically completed. Within a single global transaction, the relationship between any pair of participating threads is either *tightly-coupled* or *loosely-coupled*:

- A tightly-coupled relationship is one where a pair of threads are designed to share resources. In addition, with respect to an RM's isolation policies, the pair are treated as a single entity because they share a common transaction branch. Thus, for a pair of tightly-coupled threads, the RM must guarantee that resource deadlock does not occur within the transaction branch.
- A loosely-coupled relationship provides no such guarantee since the threads are assigned to separate transaction branches. With respect to an RM's isolation policies, the pair may be treated as if they were in separate global transactions even though the work is atomically completed.

Within a single global transaction, a set of tightly-coupled threads may consist of more than just a pair. Moreover, many sets of tightly-coupled threads may exist within the same global transaction and each set is loosely coupled with respect to the others. The reference manual pages in Chapter 5 indicate how a TM communicates these relationships to an RM.

2.3 Transaction Completion and Recovery

TMs and RMs use two-phase commit with presumed rollback, as defined by the OSI TP standard (model).

In Phase 1, the TM asks all RMs to *prepare to commit* (or *prepare*) transaction branches. This asks whether the RM can guarantee its ability to commit the transaction branch. An RM may have to query other entities internal to that RM. CRMs are asked to prepare transaction branches that they created. (See Section 2.2.5 on page 8.) This involves sending the prepare request to the remote site and receiving the outcome.

If an RM can commit its work, it records stably the information it needs to do so, then replies affirmatively. CRMs usually do not need to stably record the results of their operations because of the way the two-phase commit protocol with presumed rollback works. The TM must stably record information about transaction branches created by CRMs when it decides to commit after Phase 1.

A negative reply from an RM reports failure for any reason. After making a negative reply and rolling back its work, the RM can discard any knowledge it has of the transaction branch.

In Phase 2, the TM issues all RMs an actual request to commit or roll back the transaction branch, as the case may be. CRMs are asked to commit or roll back transaction branches that they created. (See Section 2.2.5 on page 8.) This involves sending the commit or rollback request to the remote site and receiving the outcome. (Before issuing requests to commit, the TM stably records the fact that it decided to commit, as well as a list of all involved RMs and any subordinate transaction branches that they created.) All RMs commit or roll back changes to shared resources and then return status to the TM. The TM can then discard its knowledge of the global transaction.

2.3.1 Rolling Back the Global Transaction

The TM rolls back the global transaction if any RM responds negatively to the Phase 1 request, or if the AP directs the TM to roll back the global transaction. Therefore, any negative response vetoes the global transaction. A negative response concludes an RM's involvement in the global transaction.

The TM effects Phase 2 by telling all RMs to roll back transaction branches. They must not let any changes to shared resources become permanent. The TM does not issue Phase 2 requests to RMs that responded negatively in Phase 1. The TM does not need to record stably the decision to roll back nor the participants in a rolled back global transaction.

2.3.2 Protocol Optimisations

- **Read-only**
  An RM can respond to the TM's prepare request by asserting that the RM was not asked to update shared resources in this transaction branch. This response concludes the RM's involvement in the transaction; the Phase 2 dialogue between the TM and this RM does not occur. The TM need not stably record, in its list of participating RMs, an RM that asserts a read-only role in the global transaction.

  However, if the RM returns the read-only optimisation before all work on the global transaction is prepared, global *serialisability*[1] cannot be guaranteed. This is because the RM

[1] Serialisability is a property of a set of concurrent transactions. For a serialisable set of transactions, at least one serial sequence of the transactions exists that produces identical results, with respect to shared resources, as does concurrent execution of the transaction.

may release transaction context, such as read locks, before all application activity for that global transaction is finished.

A CRM can assert that it is not a participant in the transaction branch active in a particular thread when the TM suspends or ends the thread's association with the transaction branch. The CRM is not considered a participant unless it becomes reassociated with the transaction branch. This assertion may allow a TM to make a One-Phase Commit optimisation described below.

- **One-Phase Commit**
  A TM can use one-phase commit if it knows that there is only one RM anywhere in the DTP system that is making changes to shared resources. In this optimisation, the TM makes its Phase 2 commit request without having made a Phase 1 prepare request. Since the RM decides the outcome of the transaction branch and forgets about the transaction branch before returning to the TM, there is no need for the TM to record stably these global transactions and, in some failure cases, the TM may not know the outcome.

2.3.3 Heuristic Branch Completion

Some RMs may employ heuristic decision-making: an RM that has prepared to commit a transaction branch may decide to commit or roll back its work independently of the TM. It could then unlock shared resources. This may leave them in an inconsistent state. When the TM ultimately directs an RM to complete the branch, the RM may respond that it has already done so. The RM reports whether it committed the branch, rolled it back, or completed it with mixed results (committed some work and rolled back other work).

An RM that reports heuristic completion to the TM must not discard its knowledge of the transaction branch. The TM calls the RM once more to authorise it to forget the branch. This requirement means that the RM must notify the TM of all heuristic decisions, even those that match the decision the TM requested.

The OSI TP standards (Model) and (Service) define heuristics more precisely.

2.3.4 Failures and Recovery

A useful DTP system must be able to recover from a variety of failures. A storage device or medium, a communication path, a node or a program could fail.

Failures that a node can correct internally may not affect a global transaction.

Failures that do not disrupt the commitment protocol let the DTP system respond by rolling back appropriate global transactions. For example, an RM recovering from a failure responds negatively to a prepare request based on the fact that it does not recognise the XID.

More significant failures may disrupt the commitment protocol. The TM typically senses the failure when an expected reply does not arrive.

Failure and recovery processing in an X/Open DTP system is compatible with the OSI TP standards, which define the presumed-rollback protocol. The X/Open DTP model makes these assumptions:

- TMs and RMs (except for CRMs) have access to stable storage.
- TMs coordinate and control recovery.
- RMs (except for CRMs) provide for their own restart and recovery of their own state. On request, an RM must give a TM a list of XIDs that the RM has prepared for commitment or has heuristically completed. Also on request, a TM must give a CRM a list of XIDs for which the TM has information logged on behalf of the CRM.

Chapter 3

Interface Overview

This chapter gives an overview of the XA and XA+ interfaces. In an X/Open DTP system, XA is the interface between a TM and an RM, and XA+ is the interface between a TM and a CRM. Chapter 5 contains reference manual pages for each function in alphabetical order. These pages contain C-language function prototypes.

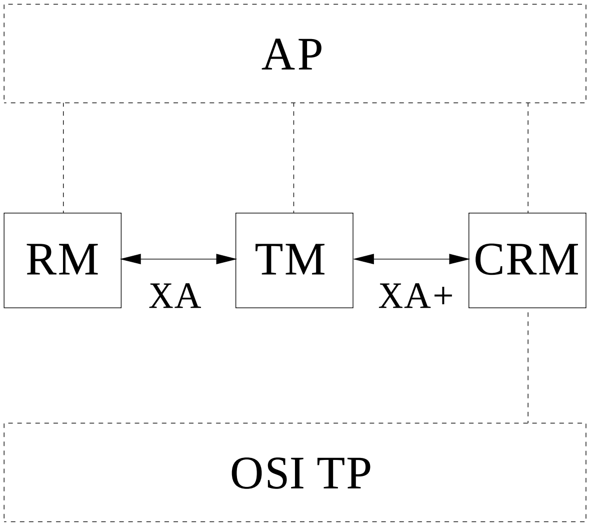


Figure 3-1 The XA and XA+ Interfaces

3.1 Index to Services in the XA and XA+ Interfaces

The *ax_**( ) functions let an RM call a TM. All TMs must provide these functions. These functions let an RM dynamically control its participation in a transaction branch. Additionally, CRMs use the *ax_* interface to create transaction branches, to suspend or complete transaction branches, and to propagate the commitment protocol to transaction branches. The *ax_*( ) functions are listed in the following table:

Name	Description	See
ax_add_branch†	Generate a new branch for an existing global transaction.	Section 3.4.1 on page 20
ax_commit†	Propagate transaction branch commitment to a transaction manager.	Section 3.5 on page 22
ax_done†	Report that recovery associated with a transaction branch is complete.	Section 3.7 on page 25
ax_end†	Notify the transaction manager to end work performed on behalf of a transaction branch.	Section 3.3 on page 18
ax_forget_branch†	Remove a branch from an existing global transaction.	Section 3.4.1 on page 20
ax_get_branch_info†	Access information for a transaction branch.	Section 3.4 on page 20
ax_prepare†	Propagate transaction prepare to commit to a transaction branch.	Section 3.5 on page 22
ax_ready†	Find out if a subordinate should commit or roll back.	Section 3.7 on page 25
ax_recover†	Get a list of XIDs for which the CRM has logged information with the TM.	Section 3.4.1 on page 20
ax_reg	Register an RM with a TM.	Section 3.3.2 on page 19
ax_reg_2	Register an RM with a TM, and pass options information to the TM.	Section 3.3.2 on page 19
ax_rollback†	Propagate transaction rollback to a transaction manager.	Section 3.5 on page 22
ax_set_branch_info†	Save information for a transaction branch.	Section 3.4 on page 20
ax_start†	Notify the transaction manager to propagate or resume a transaction branch association with this thread of control.	Section 3.3 on page 17
ax_start_2†	Notify the transaction manager to propagate or resume a transaction branch association with this thread of control, and pass options information to the TM.	Section 3.3 on page 18
ax_unreg	Unregister an RM with a TM.	Section 3.3.2 on page 19

† The functions marked with a dagger sign are applicable to the XA+ interface only. Unmarked functions apply to both XA+ and XA.

Note: The *ax_**( ) function names are only templates.

The actual names of the functions are internal to the TM. The TM inserts a pointer to the TM switch structure into the location pointed to by a field in the RM's switch structure prior to making any calls to the RM.

The *xa_*()* functions are supplied by RMs operating in the DTP environment and called by TMs. When an AP calls a TM to start a global transaction, the TM may use the *xa_* interface to inform RMs of the transaction branch. After the AP uses the RM's native interface to do work in support of the global transaction, the TM calls *xa_*() functions to commit or roll back branches. One other *xa_*() function helps the TM coordinate failure recovery. The *xa_*() functions are listed in the following table:

Name	Description	See
xa_close	Terminate the AP's use of an RM.	Section 3.2 on page 16
xa_commit	Tell the RM to commit a transaction branch.	Section 3.5 on page 22
xa_complete	Test an asynchronous *xa_* operation for completion.	Section 3.6.3 on page 24
xa_done†	Report that a transaction branch has been committed.	Section 3.7 on page 25
xa_end	Dissociate the thread from a transaction branch.	Section 3.3 on page 17
xa_forget	Permit the RM to discard its knowledge of a heuristically-completed transaction branch.	Section 3.7 on page 25
xa_open	Initialise an RM for use by an AP.	Section 3.2 on page 16
xa_prepare	Ask the RM to prepare to commit a transaction branch.	Section 3.5 on page 22
xa_ready†	Report that a transaction branch has been prepared.	Section 3.7 on page 25
xa_recover	Get a list of XIDs the RM has prepared or heuristically completed.	Section 3.7 on page 25
xa_rollback	Tell the RM to roll back a transaction branch.	Section 3.5 on page 22
xa_start	Start or resume a transaction branch — associate an XID with future work that the thread requests of the RM.	Section 3.3 on page 17
xa_start_2	Start or resume a transaction branch — associate an XID with future work that the thread requests of the RM, and return options status to the RM.	Section 3.3 on page 18
xa_wait†	Give control to a CRM so that it can report status from superiors.	Section 3.8 on page 25
xa_wait_recovery†	Give control to a CRM so that it can report status from subordinates.	Section 3.7 on page 25

† The functions marked with a dagger sign are applicable to the XA+ interface only. Unmarked functions apply to both XA+ and XA.

A TM must call the *xa_*()* functions in a particular sequence (see the state tables in Chapter 6). When a TM invokes more than one RM with the same *xa_*()* function, it can do so in an arbitrary sequence.

Note: The *xa_*()* function names are only templates.

The actual names of these functions are internal to the RM. The RM publishes the name of a structure (see Section 4.4 on page 29) that specifies the entry points to the RM.

3.2 Opening and Closing Resource Managers

In each thread of control, the TM must call *xa_open*( ) for each RM directly accessible by that thread before calling any other *xa_**( ) function. The TM must eventually call *xa_close*( ) to dissociate the AP from the RM.

If an RM needs to take start-up actions (such as opening files, opening paths to a server, or resynchronising a node on the network), then it could do so when called by *xa_open*( ). X/Open does not specify the actual meaning of *xa_open*( ) and *xa_close*( ) to an RM, but the effect must be internal to the RM and must not affect transaction processing in either the calling TM or in other RMs.

If an RM requires or accepts parameters to govern its operation (for example, a directive to open files for reading only), or to identify a target resource domain, then a string argument to *xa_open*( ) conveys this information. If the RM does not require initialisation parameters, the string is typically an empty string. The *xa_close*( ) call likewise takes a string.

TMs typically read the initialisation string from a configuration file. The *xa_open*( ) function, and the string form of its argument, support portability. A TM can give the administrator control over every run-time option that any RM provides through *xa_open*( ) with no reprogramming or relinking. The administrator must only edit a configuration file or perform a comparable, system-specific procedure.

The TM calls *xa_open*( ) with an identifier that the TM uses subsequently to identify the RM instance. A single RM may service multiple resource domains using multiple RM instances, if each instance supports independent transaction completion. For example, a single database system might access several data domains, or a single printer spooler might service multiple printers. The TM calls such an RM's *xa_open*( ) function several times, once for each instance, using string parameters that identify the respective resource. It must generate a different RM identifier for each call.

To enhance portability, RMs in the DTP environment should rely on use of *xa_open*( ) in place of any non-standard *open* service the RM may provide in its native interface. If an RM lets DTP applications call the native *open* function, the effect must not conflict with the TM's use of *xa_open*( ).

3.3 Association of Threads with Transaction Branches

Several threads may participate in a single transaction branch, some more than once. The *xa_start*( ) and *xa_end*( ) functions pass an XID to an RM to associate or dissociate the calling thread with a branch. The association is not necessarily the thread's initial association with the branch; its dissociation is not necessarily the final one.

A thread's association with a transaction branch can be active or suspended:

- A thread is actively associated with a transaction branch if it has called *xa_start*( ) and has not made a corresponding call to *xa_end*( ). A thread is allowed only one active association with each RM at a time.
- Certain calls to *xa_end*( ) suspend the thread's association (see **Suspend** below). The call may indicate that the association can *migrate*; that is, that any thread may resume the association. In this case, the calling thread is no longer associated. (An RM may indicate that it does not support association migration.)

If a thread calls *xa_end*( ) to suspend its association but the association cannot migrate to another thread, the calling thread retains a suspended association with the transaction branch.

Several uses of *xa_start*( ) and *xa_end*( ) are considered below:

- **Start**
  The primary use of *xa_start*( ) is to register a new transaction branch with the RM. This marks the start of the branch. Subsequently, the AP, *using the same thread of control*, uses the RM's native interface to do useful work. All requests for service made by the same thread are part of the same branch until the thread dissociates from the branch (see below).

  The return code from *xa_start*( ) may indicate that the RM has already vetoed commitment of the transaction branch. This return code is not an error; rolled back global transactions may be function, while actual errors deserve the administrator's attention.

  A CRM primarily uses *ax_start*( ) to register a propagated transaction branch with the TM. The TM, in turn, issues *xa_start*( ) calls to the applicable local RMs (that is, those that do not use dynamic registration). The return code from *ax_start*( ) reflects those returned to the TM from the RMs on *xa_start*( ).

- **Join**
  Another use of *xa_start*( ) is to join an existing transaction branch. TMs must use a certain form of *xa_start*( ) so that RMs can validate that they recognise the passed XID.

  RMs in the DTP environment should anticipate that many threads try to use them concurrently. If multiple threads use an RM on behalf of the same XID, the RM is free to serialise the threads' work in any way it sees fit. For example, an RM may block a second or subsequent thread while one is active.

  Another use of *ax_start*( ) by a CRM is to inform a TM to join or reuse an existing transaction branch. CRMs must use a certain form of *ax_start*( ) so that the TM can validate that it recognises the passed XID.

- **Resume**
  A special form of *xa_start*( ) is used by a TM to inform an RM to associate a thread with an existing transaction branch that has been suspended (see below).

  A special form of *ax_start*( ) is used by a CRM to inform a TM to associate a thread with an existing transaction branch that has been suspended (see below).

- **End**
  A typical call to *xa_end*( ) dissociates the calling thread from the transaction branch and lets the branch be completed (see Section 3.5 on page 22). Alternatively, a thread may use *xa_start*( ) to rejoin the branch.

  A CRM typically uses *ax_end*( ) to inform a TM to dissociate the calling thread from the transaction branch to let the branch be completed (see Section 3.5 on page 22). Alternatively, a CRM may use *ax_start*( ) to rejoin the branch.

- **Suspend**
  A form of *xa_end*( ) suspends, instead of ending, a thread's association with the transaction branch. This indicates that the thread has left the branch in an incomplete state. By using the **resume** form of *xa_start*( ), it or another thread resumes its association with the branch. Another thread may resume a suspended association only if the RM has indicated that it supports association migration. Instead of resuming, the TM may completely end the suspended association by using *xa_end*( ).

  A form of *ax_end*( ) requests a TM to suspend, instead of ending, a thread's association with the transaction branch. This indicates that the thread has left the branch in an incomplete state. By using the **resume** form of *ax_start*( ), the CRM can resume the thread's association with the branch. A CRM in another thread may resume a suspended association only if the TM has indicated that it supports association migration. Instead of resuming, the CRM may completely end the suspended association by using *ax_end*( ).

- **Rollback-only**
  An RM need not wait for global transaction completion to report an error. The RM can return **rollback-only** as the result of any *xa_start*( ) or *xa_end*( ) call. The TM can use this knowledge to avoid starting additional work on behalf of the global transaction. An RM can also unilaterally roll back and forget a transaction branch any time before it prepares it. A TM detects this when an RM subsequently indicates that it does not recognise the XID.

  A TM need not wait for global transaction completion to report an error. The TM can return **rollback-only** as the result of any *ax_start*( ) or *ax_end*( ) call if an RM indicates **rollback-only** on the corresponding *xa_**( ) call. A TM can also unilaterally roll back and forget a transaction branch any time before it prepares it. A CRM detects this when a TM subsequently indicates that it does not recognise the XID.

- **Transaction branch states**
  Several state tables appear in Chapter 6. Each call to *xa_start*( ) or *xa_end*( ) may affect the status of the thread's association with a transaction branch (see Table 6-2 on page 105 and Table 6-3 on page 106) and the status of the branch itself (see Table 6-4 on page 108). A TM must use these functions so that each thread of control makes calls in a sequence that complies with both tables.

Note: The functions *ax_start_2*( ) and *xa_start_2*( ) are upward-compatible versions of *ax_start*( ) and *xa_start*( ) respectively. Besides acting as described above, they also allow the passing of transaction-timeout information between a CRM and RM (respectively) and the TM. In future versions of this specification, *ax_start_2*( ) and *xa_start_2*( ) may be extended to allow other options to be passed as well.

3.3.1 Transaction Context

Transaction context is RM-specific information visible to the AP. The RM should preserve certain transaction context on *xa_end*( ) so that the RM can restore context in the join or resume cases (defined above). In the join case, the RM should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the transaction branch. In the resume case, the RM should make available at least that RM-specific transaction context present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

3.3.2 Registration of Resource Managers

Normally, a TM involves all associated RMs in a transaction branch. (The TM's set of RM switches, described in Section 4.4 on page 29, tells the TM which RMs are associated with it.) The TM calls all these RMs with *xa_start*( ), *xa_end*( ) and *xa_prepare*( ), although an RM that is not active in a branch need not participate further (see Section 2.3.2 on page 10). A technique to reduce overhead for infrequently-used RMs is discussed below.

Dynamic Registration

Certain RMs, especially those involved in relatively few global transactions, may ask the TM to assume they are *not* involved in a transaction. These RMs must register with the TM before they do application work, to see whether the work is part of a global transaction. The TM never calls these RMs with any form of *xa_start*( ). An RM declares dynamic registration in its switch (see Section 4.4 on page 29). An RM can make this declaration only on its own behalf, and doing so does not change the TM's behaviour with respect to other RMs.

When an AP requests work from such an RM then, before doing any work, the RM contacts the TM by calling *ax_reg*( ). The RM must call *ax_reg*( ) from the same thread of control that the AP would use if it called *ax_reg*( ) directly. The TM returns to the RM the appropriate XID if the AP is in a global transaction.

If the thread ends its involvement in the transaction branch (using *xa_end*( )), then the RM must re-register (using *ax_reg*( )) with the TM if the AP calls it for additional work in the global transaction. If the RM does not resume its participation, then the TM does not call the RM again for that branch until the TM completes the branch.

If the RM calls *ax_reg*( ) and the AP is not in a global transaction, the TM informs the RM, and remembers that the RM is doing work outside any global transaction. In this case, when the AP completes its work with the RM, the RM must notify the TM by calling *ax_unreg*( ). The RM must call *ax_unreg*( ) from the same thread of control from which it called *ax_reg*( ). Until then — that is, as long as the AP thread involves the RM outside a global transaction — the TM neither lets the AP start a global transaction, nor lets any RM register through the same thread to participate in one.

Note: The function *ax_reg_2*( ) is an upward-compatible version of *ax_reg*( ). Besides acting as described above, it also allows an RM to pass transaction-timeout information to the TM. In future versions of this specification, *ax_reg_2*( ) may be extended to allow other options to be passed as well.

3.4 Branch Creation

When an AP requests a CRM to send a message to a remote AP, the CRM has to create a subordinate transaction branch if one does not already exist for the dialogue with that remote AP. There are two methods of creating and managing transaction branches. One is called transaction manager managed transaction branches, and the other is called communication resource manager managed transaction branches.

3.4.1 Transaction Manager Managed Transaction Branches

The transaction manager can create and manage transaction branches. The CRM requests the TM to create a branch by using the *ax_add_branch*( ) call. The TM returns an XID containing the global transaction identifier and a new branch qualifier. The TM also retains knowledge of the branch so that it is included in the transaction completion procedure. The CRM passes the branch's XID to its counterpart at the remote site. That CRM informs the TM that a new branch is being propagated to it by using the *ax_start*( ) call.

In some cases, the CRM at the remote site may realise that it can join an existing transaction branch rather than propagate the new one. It informs its counterpart at the superior site of this fact. The CRM at the superior site can use the *ax_forget_branch*( ) call to inform the TM to discard knowledge of the transaction branch so that it is not included in the transaction completion procedure.

When a CRM creates a transaction branch, it can request the TM to save information about the branch using the *ax_set_branch_info*( ) call. The TM logs this information when it records the positive (*ready*) votes after the first phase of the two-phase commitment procedure. When a branch is propagated to a remote site, the CRM can use the *ax_set_branch_info*( ) call to save information about the superior. The TM logs this information during the commitment procedure also. This logging service of the TM can relieve a CRM of any logging responsibility of its own. This provides for higher performance by combining logging steps.

A CRM can retrieve information that it has saved using *ax_set_branch_info*( ) at any time using the *ax_get_branch_info*( ) call. A CRM can retrieve a list of XIDs for which it has saved information by means of *ax_set_branch_info*( ) at any time using the *ax_recover*( ) call.

3.4.2 Communication Resource Manager Managed Transaction Branches

If the communication resource manager creates and manages its transaction branches without the aid of the transaction manager, it does not involve the transaction manager in the branch creation process. Such a communication resource manager appears to be an ordinary resource manager to the transaction manager in the process that creates the transaction branch.

The CRM assigns an XID for the transaction branch (see Section 4.2 on page 28). The CRM must follow the rules for assigning a unique branch qualifier. The CRM passes the branch's XID to its counterpart at the remote site. That CRM informs the TM that a new branch is being propagated by using the *ax_start*( ) call.

The CRM can request the TM to save information for it using the *ax_set_branch_info*( ) call. Information is saved using the local XID known to the TM. Unlike the transaction manager managed transaction branches, the TM does not know about the existence of transaction branches. Therefore, the TM can only save one record of information associated with the local XID for the CRM. The TM logs this information when it records the positive (*ready*) votes after the first phase of the two-phase commitment procedure. When a branch is propagated to a remote site, the CRM can use *ax_set_branch_info*( ) to save information about the superior. The TM logs this information during the commitment procedure also. This logging service of the TM can relieve a CRM of any logging responsibility of its own. This provides for higher performance by combining logging steps.

The CRM can retrieve information that it has saved via *ax_set_branch_info*( ) at any time using the *ax_get_branch_info*( ) call.

3.5 Branch Completion

A TM calls *xa_prepare*( ) to ask the RM to prepare to commit a transaction branch. The RM places any resources it holds in a state such that it can either make any changes permanent if the TM subsequently calls *xa_commit*( ), or nullify any changes if the TM calls *xa_rollback*( ). An affirmative return from *xa_prepare*( ) guarantees that a subsequent *xa_commit*( ) or *xa_rollback*( ) succeeds, even if the RM experiences a failure after responding to *xa_prepare*( ).

A CRM propagates the prepare command to its counterpart at a remote site. The CRM at the remote site uses *ax_prepare*( ) to inform the TM that the transaction branch is being prepared. The TM prepares local RMs and subordinate branches and returns the outcome to the CRM. The CRM reports the outcome to the CRM at the superior site which, in turn, reports the outcome to the TM.

A TM calls *xa_commit*( ) to direct the RM to commit a transaction branch. The RM applies permanently any changes it has made to shared resources, and releases any resources it held on behalf of the branch. A TM calls *xa_rollback*( ) to ask the RM to roll back a branch. The RM undoes any changes that it applied to shared resources, and releases any resources it held.

The CRM propagates the commit or rollback command to its counterpart at a remote site. The CRM at the remote site uses *ax_commit*( ) to inform the TM that the transaction branch is being committed or *ax_rollback*( ) to inform the TM that the transaction branch is being rolled back. The TM commits or rolls back local RMs and subordinate branches as indicated and returns the outcome to the CRM. The CRM reports the outcome to the CRM at the superior site which, in turn, reports the outcome to the TM.

Before a TM can call *xa_prepare*( ) for a transaction branch, all associations must be completely ended with *xa_end*( ) (see Section 3.3 on page 17). Any thread can then initiate branch completion. That is, the TM may supervise branch completion with a separate thread from the AP threads that did work on behalf of the global transaction.

A CRM may issue *ax_prepare*( ) to a TM for an association that is suspended. The TM ensures that the association with local RMs is completed by using a form of *xa_end*( ) before issuing *xa_prepare*( ). If the suspended association cannot be migrated, the CRM must issue the *ax_prepare*( ) call in the thread of control that was suspended and indicated it could not migrate. Alternatively, the CRM can dissociate a suspended non-migratable thread by issuing a form of the *ax_end*( ) call in that thread of control. The CRM is then free to issue *ax_prepare*( ) in any thread.

Optimisations

This section describes the use of *xa_**( ) functions in the standard two-phase commit protocol. See Section 2.3.2 on page 10 for other permissible sequences of these calls.

Heuristic Decision

The X/Open DTP model lets RMs complete transaction branches heuristically (see Section 2.3.3 on page 11). The RM cannot discard its knowledge of such a branch until the TM permits this by calling *xa_forget*( ) for each branch.

3.5.1 Transaction Chaining

Transaction chaining can be selected by the application program using the TX interface. Transaction chaining means that the application is always in transaction mode. That is, the completion of one transaction, commit or rollback, automatically creates a new one.

CRMs that support chaining need to keep subordinate branches in transaction mode too. An option of *xa_commit*( ) and *xa_rollback*( ) allow the TM to pass along the identification of the new transaction branch to the CRM. Similarly, an option of *ax_commit*( ) and *ax_rollback*( ) allow the CRM to propagate the identification of the new transaction branch to the TM at the remote site.

The completion of one transaction branch and the creation of a new one are handled in one function so that the CRM can propagate transaction completion and the beginning of a new one in one logical communications operation (or PDU: protocol data unit). Creation of a new transaction branch does not actually start processing for it. Processing for the new transaction starts in the transaction originator when the TM calls RMs to start processing in the application thread of control. Processing for new transaction branches starts in subordinate application threads of control when the CRM calls the TM to start processing.

3.5.2 Communication Resource Manager Managed Transaction Branches

Since the TM does not know about subordinate transaction branches created by the communication resource manager when the CRM is managing its own branches, the TM issues *xa_prepare*( ), *xa_commit*( ), or *xa_rollback*( ) only once for the local transaction branch to the CRM using the local XID known to the TM. It is up to the CRM to propagate the correct corresponding protocol to each subordinate branch created by the CRM. It is also up to the CRM to direct the two-phase commitment procedure, if necessary, when called with a one-phase commit optimisation. The CRM may request the TM to change one-phase commit into two-phase commit using a return code on *xa_commit*( ). This is necessary if the CRM expects the TM to log CRM recovery information. A TM can optionally support changing a one-phase commit into a two-phase commit at the request of a CRM.

3.5.3 Transaction Commitment and Rollback in Subordinates

The CRM propagating a subordinate branch determines whether the subordinate AP can issue *tx_commit*( ) or *tx_rollback*( ). The rules are controlled by flag settings in the CRM's switch structure. These flag settings are ignored for other RMs in the thread of control.

3.6 Synchronous, Non-blocking and Asynchronous Modes

3.6.1 Synchronous

The *xa_**( ) functions typically operate synchronously: control does not return to the caller until the operation is complete. Some functions, notably *xa_start*( ) (see Section 3.3 on page 17), may block the calling thread.

Two other calling modes help the TM schedule its work when dealing with several RMs, as described in the following sections.

3.6.2 Non-blocking

Certain *xa_**( ) calls direct the RM to operate synchronously with the caller but without blocking it. If the RM cannot complete the call without blocking, it reports this immediately.

3.6.3 Asynchronous

Most *xa_*( )* functions have a form by which the caller requests asynchrony. Asynchronous calls should return immediately. The caller can subsequently call *xa_complete*( ) to test the asynchronous operation for completion.

A TM might give an RM an asynchronous request (particularly a request to prepare to commit a transaction branch) so that the TM could do other work in the meantime. Within the same thread of control, a TM cannot use asynchrony to give additional work to the same RM for the same branch; the only *xa_**( ) call a TM can give to the RM for the same branch is *xa_complete*( ) to test that operation's completion. However, for *xa_forget*( ), and for the branch-completion functions, *xa_commit*( ), *xa_prepare*( ) and *xa_rollback*( ), the TM may issue multiple commands to the same RM from within the same thread of control. Each of these commands must be for a different branch.

3.7 Failure Recovery

A TM must ensure orderly completion of all transaction branches. A TM calls *xa_recover*( ) during failure recovery to get a list of all branches that an RM has prepared or heuristically completed. If a TM does not have a record of a branch returned by *xa_recover*( ), it normally orders the RM to roll the branch back by issuing *xa_rollback*( ).

When a TM is undergoing recovery processing, it may find one or more transaction branches in the prepared state for which it is not the commitment coordinator. In this case, the TM requests the superior-side CRM to find out the transaction branch's disposition from the immediate superior. The TM does this by issuing an *xa_ready*( ) call to the CRM. The CRM in the superior reports this to the TM by issuing *ax_ready*( ).

During recovery situations, a TM may be informed that a subordinate has, in fact, committed a transaction branch when it receives an *ax_done*( ) call from a CRM. This event fixes a previous problem due to a communications failure during the second phase of the two-phase commit procedure. If this TM, in turn, has a superior, it calls the superior-side CRM using *xa_done*( ) to request that the CRM pass this information on to the superior. The CRM in the superior reports this to the TM by issuing *ax_done*( ).

A special process can be set up at each node to handle *ax_ready*( ) and *ax_done*( ) calls. The TM in this process passes control to the CRM by issuing *xa_wait_recovery*( ). The CRM may never respond to this call, but it is free to issue the *ax_ready*( ) and *ax_done*( ) calls at any time.

Unilateral RM Action

An RM can mark a transaction branch as rollback-only any time except after a successful prepare. A TM detects this when the RM returns a rollback-only return code. An RM can also unilaterally roll back and forget a branch any time except after a successful prepare. A TM detects this when a subsequent call indicates that the RM does not recognise the XID. The former technique gives the TM more information.

If a thread of control terminates, an RM must dissociate and roll back any associated transaction branch. If an RM experiences machine failure or termination, it must also unilaterally roll back all branches that have not successfully prepared.

3.8 Shutdown Processing

A TM may need to shut down a thread of control prior to the completion of the two-phase commit protocol for transaction branches for which it is responsible and for which it is a subordinate. The TM instructs the CRM to wait for commands from superiors by issuing the *xa_wait*( ) call. The CRM calls the TM with *ax_commit*( ) or *ax_rollback*( ) as appropriate.

Chapter 4

The <xa.h> Header

This chapter specifies structure definitions, flags and error codes to which conforming products must adhere. It also declares the functions by which RMs call a TM. (Entry points to an RM are contained in the RM's switch; see Section 4.4 on page 29.) This is the minimum content of an include file called **<xa.h>**. Fully standardising this information lets RMs be written independently of the TMs that use them. It also lets users interchange TMs and RMs without recompiling.

Appendix A contains an **<xa.h>** header file with **#define** statements suitable for ISO C (see the ISO C standard) and Common Usage C implementations. This chapter contains excerpts from the ISO C code in **<xa.h>**. The synopses in Chapter 5 also use ISO C.

4.1 Naming Conventions

The XA interface uses certain naming conventions to name its functions, flags and return codes. All names that appear in **<xa.h>** are part of the XA name space. This section describes the XA naming conventions.

- The negative (error) codes returned by the *xa_* functions all begin with XAER_. Their non-negative return codes all begin with XA_.
- The names of all TM functions that RMs call begin with *ax_* (for example, *ax_reg*( )). Their negative (error) return codes all begin with TMER_. Their non-negative return codes all begin with TM_.
- Names of flags passed to XA functions, and of flags in the RM switch, begin with TM.

4.2 Transaction Identification

The **<xa.h>** header defines a public structure called an XID to identify a transaction branch. RMs and TMs both use the XID structure. This lets an RM work with several TMs without recompilation.

The XID structure is specified in the C code below in **struct xid_t**. The XID contains a format identifier, two length fields and a data field. The data field comprises at most two contiguous components: a global transaction identifier (*gtrid*) and a branch qualifier (*bqual*).

The *gtrid_length* element specifies the number of bytes (1-64) that constitute *gtrid*, starting at the first byte of the data element (that is, at *data*[0]). The *bqual_length* element specifies the number of bytes (1-64) that constitute *bqual*, starting at the first byte after *gtrid* (that is, at *data*[*gtrid_length*]). Neither component in *data* is null-terminated. The TM need not initialise any unused bytes in *data*.

Although **<xa.h>** constrains the length and byte-alignment of the data element within an XID, it does not specify the data's contents. The only requirement is that both *gtrid* and *bqual*, taken together, must be globally unique. The recommended way of achieving global uniqueness is to use the naming rules specified for OSI CCR atomic action identifiers (see the OSI CCR standard). If OSI CCR naming is used, then the XID's *formatID* element should be set to 0; if some other format is used, then the *formatID* element should be greater than 0. A value of −1 in *formatID* means that the XID is null.

The RM must be able to map the XID to the recoverable work it did for the corresponding branch. RMs may perform bitwise comparisons on the data components of an XID for the lengths specified in the XID structure. Most XA functions pass a pointer to the XID. These pointers are valid only for the duration of the call. If the RM needs to refer to the XID after it returns from the call, it must make a local copy before returning.

```
/*
 * Transaction branch identification: XID and NULLXID:
 */
#define XIDDATASIZE     128     /* size in bytes */
#define MAXGTRIDSIZE     64     /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE     64     /* maximum size in bytes of bqual */
struct xid_t {
    long formatID;              /* format identifier */
    long gtrid_length;          /* value 1-64 */
    long bqual_length;          /* value 1-64 */
    char data[XIDDATASIZE];
    };
typedef struct xid_t XID;
/*
 *  A value of -1 in formatID means that the XID is null.
 */
```

4.3 XA Options

A structure of type XACTL (see below) is defined to pass optional information from TMs to RMs and CRMs, and from CRMs to TMs. A flag value tells whether a structure element contains a valid value. Structure elements are defined to contain optional values. Currently, a timeout value is the only structure element defined. In future versions of this specification, other elements and corresponding flags may also be defined.

```
/*
 * XA Options
 */
typedef long TRANSACTION_TIMEOUT;    /* type of transaction timeouts */
/*
 * Structure for optional XA information
 */
struct xactl_t {
   long flags;                       /* valid element flags */
   TRANSACTION_TIMEOUT timeout;      /* timeout value */
};
typedef struct xactl_t XACTL;
#define XAOPTS_NOFLAGS 0x00000000L       /* no optional values */
#define XAOPTS_TIMEOUT 0x00000001L       /* timeout value present */
};
```

4.4 Resource Manager Switch

The TM administrator can add or remove an RM from the DTP system by simply controlling the set of RMs linked to executable modules. Each RM must provide a *switch* that gives the TM access to the RM's *xa_* functions. This lets the administrator change the set of RMs linked with an executable module without having to recompile the application. A different set of RMs and their switches may be linked into each separate application-executable module in the DTP system. Several instances of an RM can share the RM's switch structure.

An RM's switch uses a structure called **xa_switch_t**. The switch contains the RM's name, non-null pointers to the RM's entry points, and a flag and a version word. The flags tell whether the RM uses dynamic registration (see Section 3.3.2 on page 19), whether the RM operates asynchronously (see Section 3.6 on page 24) and whether the RM supports the migration of associations (see Section 3.3 on page 17). Section 4.6 on page 33 defines constants used as these flags. The RM cannot change these declarations during the operation of the DTP system.

```
/*
 * XA Switch Data Structure
 */
#define RMNAMESZ 32      /* length of resource manager name */
                         /* including the null terminator */
#define MAXINFOSIZE 256  /* maximum size in bytes of xa_info strings */
                         /* including the null terminator */
struct xa_switch_t {
   char name[RMNAMESZ];  /* name of resource manager */
   long flags;           /* options specific to the resource manager */
   long version;         /* must be 1 */
   int (*xa_open_entry)(char *, int, long);
                         /* xa_open function pointer */
   int (*xa_close_entry)(char *, int, long);
                         /* xa_close function pointer */
   int (*xa_start_entry)(XID *, int, long);
                         /* xa_start function pointer */
   int (*xa_end_entry)(XID *, int, long);
                         /* xa_end function pointer */
   int (*xa_rollback_entry)(XID *, int, long);
                         /* xa_rollback function pointer */
   int (*xa_prepare_entry)(XID *, int, long);
                         /* xa_prepare function pointer */
   int (*xa_commit_entry)(XID *, int, long);
                         /* xa_commit function pointer */
   int (*xa_recover_entry)(XID *, long, int, long);
                         /* xa_recover function pointer */
   int (*xa_forget_entry)(XID *, int, long);
                         /* xa_forget function pointer */
   int (*xa_complete_entry)(int *, int *, int, long);
                         /* xa_complete function pointer */
   int (*xa_ready_entry)(XID *, int, long);
                         /* xa_ready function pointer */
   int (*xa_done_entry)(XID *, int, long);
                         /* xa_done function pointer */
   int (*xa_wait_recovery_entry)(int, long);
                         /* xa_wait_recovery function pointer */
   int (*xa_wait_entry)(int, long);
                         /* xa_wait function pointer */
   int (*xa_start_2_entry)(XID *, int, XACTL *, long);
                         /* xa_start_2 function pointer */
   struct ax_switch_t **xa_tmswitch;
                         /* Location of TM switch pointer */
};
```

4.5 Transaction Manager Switch

The TM provides a switch structure for RMs to access the TM's *ax_*()* functions. If the flag TMSWITCHOK is set in the RM's switch, the TM places a pointer to the TM's switch structure in the location pointed to by the **xa_tmswitch** pointer field in the RM's switch structure.

A TM's switch uses a structure called **ax_switch_t**. The switch contains pointers to the TM's entry points, a flag word and a version word. The pointers are undefined for functions not supported by the TM. The flags tell whether the TM supports threads and the subordinate set of *ax_*()* functions including:

- *ax_add_branch*( )
- *ax_commit*( )
- *ax_done*( )
- *ax_end*( )
- *ax_forget_branch*( )
- *ax_get_branch_info*( )
- *ax_prepare*( )
- *ax_ready*( )
- *ax_recover*( )
- *ax_rollback*( )
- *ax_set_branch_info*( )
- *ax_start*( ).

The TM cannot change these flag settings during the operation of the DTP system.

```
/*
 * AX Switch Data Structure
 */
struct ax_switch_t {
   long flags;      /* transaction manager options */
   long version;    /* must be 0 */
   int (*ax_reg_entry)(int, XID *, long);
                         /* ax_reg function pointer */
   int (*ax_unreg_entry)(int, long);
                         /* ax_unreg function pointer */
   int (*ax_start_entry)(int, XID *, long);
                         /* ax_start function pointer */
   int (*ax_end_entry)(int, XID *, long);
                         /* ax_end function pointer */
   int (*ax_rollback_entry)(int, XID *, long);
                         /* ax_rollback function pointer */
   int (*ax_prepare_entry)(int, XID *, long);
                         /* ax_prepare function pointer */
   int (*ax_commit_entry)(int, XID *, long);
                         /* ax_commit function pointer */
   int (*ax_recover_entry)(int, XID *, long, long);
                         /* ax_recover function pointer */
   int (*ax_add_branch_entry)(int, XID *, long);
                         /* ax_add_branch function pointer */
   int (*ax_forget_branch_entry)(int, XID *, long);
                         /* ax_forget_branch function pointer */
   int (*ax_set_branch_info_entry)(int, XID *, char *, long, long);
                         /* ax_set_branch_info function pointer */
   int (*ax_get_branch_info_entry)(int, XID *, char *, long *, long);
                         /* ax_get_branch_info function pointer */
   int (*ax_ready_entry)(int, XID *, long);
                         /* ax_ready function pointer */
   int (*ax_done_entry)(int, XID *, long);
                         /* ax_done function pointer */
   int (*ax_reg_2_entry)(int, XID *, XACTL *, long);
                         /* ax_reg_2_entry function pointer */
   int (*ax_start_2_entry)(int, XID *, XACTL *, long);
                         /* ax_start_2_entry function pointer */
};
```

4.6 Flag Definitions

The XA interface uses the following flag definitions. For a TM to work with different RMs without change or recompilation, each RM uses these flags, defined in the **<xa.h>** header.

The **<xa.h>** header defines a constant, TMNOFLAGS, for use in situations where no other flags are specified. An RM that does not use any flags to specify special features in its switch (see Section 4.4 on page 29) should specify TMNOFLAGS. In addition, TMs and RMs should use the same TMNOFLAGS constant as the flag argument in any *xa_* or *ax_* call in which they do not use explicit options.

Flag definitions for the XA interface are as follows:

```
/*
 * Flag definitions for the RM switch
 */
#define TMNOFLAGS         0x00000000L  /* no resource manager features
                                       selected */
#define TMREGISTER        0x00000001L  /* resource manager dynamically
                                       registers */
#define TMNOMIGRATE       0x00000002L  /* resource manager does not support
                                       association migration */
#define TMUSEASYNC        0x00000004L  /* resource manager supports
                                       asynchronous operations */
#define TMUSECHAIN        0x00000008L  /* resource manager
                                       supports transaction chaining */
#define TMUSEOPTS         0x00000010L  /* resource manager supports
                                       xa_start_2() */
#define TMUSE2PHASE       0x00000020L  /* The RM might force upgrading
                                       one-phase commit to two-phase commit*/
#define TMSWITCHOK        0x00000040L  /* resource manager has provided
                                       location for address of
                                       transaction manager switch */
#define TMNOROLLALLOWED   0x00000080L  /* tx_rollback() is not permitted
                                       in subordinates */
#define TMNOCOMALLOWED    0x00000100L  /* tx_commit() is not permitted
                                       in subordinates */
#define TMUSETHREADS      0x00000200L  /* resource manager can use threads
                                       as thread of control */
/*
 * Flag definitions for the TM switch
 */
#define TMSUPPORTSTHREADS 0x00000001L  /* The TM is prepared to use
                                       threads as thread of control */
#define TMSUBORDINATE     0x00000002L  /* The subordinate set of ax_()
                                       functions can be called */
/*
 * Flag definitions for xa_ and ax_ functions
 */
/* use TMNOFLAGS, defined above, when not specifying other flags */
#define TMASYNC           0x80000000L  /* perform function asynchronously */
#define TMONEPHASE        0x40000000L  /* caller is using one-phase
                                       commit optimisation */
#define TMFAIL            0x20000000L  /* dissociates caller and marks
                                       transaction branch rollback-only */
```

```
#define TMNOWAIT         0x10000000L  /* return if blocking condition
                                         exists */
#define TMRESUME         0x08000000L  /* caller is resuming association
                                         with suspended transaction branch */
#define TMSUCCESS        0x04000000L  /* dissociate caller from
                                         transaction branch */
#define TMSUSPEND        0x02000000L  /* caller is suspending, not ending,
                                         association */
#define TMSTARTRSCAN     0x01000000L  /* start a recovery scan */
#define TMENDRSCAN       0x00800000L  /* end a recovery scan */
#define TMMULTIPLE       0x00400000L  /* wait for any asynchronous
                                         operation */
#define TMJOIN           0x00200000L  /* caller is joining existing
                                         transaction branch */
#define TMMIGRATE        0x00100000L  /* caller intends to perform
                                         migration */
#define TMRECOVER        0x00080000L  /* call is in recovery mode */
#define TMCHAINED        0x00040000L  /* call is in transaction
                                         chaining mode */
#define TMDEFERRED       0x00020000L  /* start is pending acceptance by
                                         the application program */
```

4.7 Maximum Values

The following values are the maximum sizes permitted on the referenced *ax_**() calls. A TM must support at least these maximums. A TM may optionally support higher values.

```
/*
 * Maximum values for ax_* functions
 */
#define TMMAXBLOBLEN 1024  /* maximum blob_len for
                              ax_set_branch_info() */
#define TMMAXBLOBTOT 8192  /* maximum total blob data created
                              using ax_set_branch_info() for all
                              branches created at this node for a
                              given transaction */
```

4.8 Return Codes

As with flag definitions, all TMs and RMs must ensure interchangeability by using these return codes, defined in the **<xa.h>** header.

```
/*
 * ax_() return codes (transaction manager reports to resource manager)
 */
#define TM_RBBASE       100          /* the inclusive lower bound
                                         of the rollback codes */
#define TM_RBROLLBACK   TM_RBBASE    /* the rollback was caused by
                                        an unspecified reason */
```

```
#define TM_RBCOMMFAIL   TM_RBBASE+1    /* the rollback was caused
                                       by a communication failure */
#define TM_RBDEADLOCK   TM_RBBASE+2    /* a deadlock was detected */
#define TM_RBINTEGRITY  TM_RBBASE+3    /* a condition that violates
                                       the integrity of the
                                       resources was detected */
#define TM_RBOTHER      TM_RBBASE+4    /* the resource manager rolled
                                       back the transaction branch
                                       for a reason not on this list */
#define TM_RBPROTO      TM_RBBASE+5    /* a protocol error occurred
                                       in the resource manager */
#define TM_RBTIMEOUT    TM_RBBASE+6    /* a transaction branch took
                                       too long */
#define TM_RBTRANSIENT  TM_RBBASE+7    /* may retry the transaction
                                       branch */
#define TM_RBEND        TM_RBTRANSIENT /* the inclusive upper bound
                                       of the rollback codes */

#define TM_DEFERRED         11  /* the commit decision has not been
                                made */
#define TM_RETRY_COMMFAIL   10  /* ax_commit could not be completed
                                due to communication failure */
#define TM_NOMIGRATE        9   /* resumption must occur where
                                suspension occurred */
#define TM_HEURHAZ          8   /* the transaction branch may have
                                been heuristically completed */
#define TM_HEURCOM          7   /* the transaction branch has been
                                heuristically committed */
#define TM_HEURRB           6   /* the transaction branch has been
                                heuristically rolled back */
#define TM_HEURMIX          5   /* the transaction branch has been
                                heuristically committed and rolled
                                back */
#define TM_RDONLY           3   /* the transaction branch was read-only
                                and has been committed */
#define TM_JOIN             2   /* caller is joining existing
                                transaction branch */
#define TM_RESUME           1   /* caller is resuming association
                                with suspended transaction branch */
#define TM_OK               0   /* normal execution */

#define TMER_TMERR          -1  /* an error occurred in the
                                transaction manager */
#define TMER_INVAL          -2  /* invalid arguments were given */
#define TMER_PROTO          -3  /* function invoked in an improper
                                context */
#define TMER_NOTA           -4  /* the XID is not valid */
#define TMER_DUPID          -8  /* the XID already exists */
```

```
/*
 * xa_() return codes (resource manager reports to transaction manager)
 */
#define XA_RBBASE      100           /* the inclusive lower bound of
                                        the rollback codes */
#define XA_RBROLLBACK  XA_RBBASE     /* the rollback was caused by an
                                        unspecified reason */
#define XA_RBCOMMFAIL  XA_RBBASE+1   /* the rollback was caused by a
                                        communication failure */
#define XA_RBDEADLOCK  XA_RBBASE+2   /* a deadlock was detected */
#define XA_RBINTEGRITY XA_RBBASE+3   /* a condition that violates the
                                        integrity of the resources
                                        was detected */
#define XA_RBOTHER     XA_RBBASE+4   /* the resource manager rolled
                                        back the transaction branch for
                                        a reason not on this list */
#define XA_RBPROTO     XA_RBBASE+5   /* a protocol error occurred in
                                        the resource manager */
#define XA_RBTIMEOUT   XA_RBBASE+6   /* a transaction branch took
                                        too long */
#define XA_RBTRANSIENT XA_RBBASE+7   /* may retry the transaction
                                        branch */
#define XA_RBEND       XA_RBTRANSIENT /* the inclusive upper bound of
                                        the rollback codes */

#define XA_TWOPHASE         13  /* Use two-phase commit */
#define XA_PROMOTED         12  /* AP promoted to initiator */
#define XA_DEFERRED         11  /* the commit decision has not
                                   been made */
#define XA_RETRY_COMMFAIL   10  /* xa_commit could not be completed
                                   due to communication failure */
#define XA_NOMIGRATE        9   /* resumption must occur where
                                   suspension occurred */
#define XA_HEURHAZ          8   /* the transaction branch may have
                                   been heuristically completed */
#define XA_HEURCOM          7   /* the transaction branch has been
                                   heuristically committed */
#define XA_HEURRB           6   /* the transaction branch has been
                                   heuristically rolled back */
#define XA_HEURMIX          5   /* the transaction branch has been
                                   heuristically committed and rolled
                                   back */
#define XA_RETRY            4   /* function returned with no effect
                                   and may be re-issued */
#define XA_RDONLY           3   /* the transaction branch was read-only
                                   and has been committed */
#define XA_OK               0   /* normal execution */
```

```
#define XAER_ASYNC     -2   /* asynchronous operation already
                               outstanding */
#define XAER_RMERR     -3   /* a resource manager error occurred in the
                               transaction branch */
#define XAER_NOTA      -4   /* the XID is not valid */
#define XAER_INVAL     -5   /* invalid arguments were given */
#define XAER_PROTO     -6   /* function invoked in an improper context */
#define XAER_RMFAIL    -7   /* resource manager unavailable */
#define XAER_DUPID     -8   /* the XID already exists */
#define XAER_OUTSIDE   -9   /* resource manager doing work outside
                               global transaction */
```

Chapter 5

Reference Manual Pages

This chapter describes the interfaces to the XA service set. Reference manual pages appear, in alphabetical order, for each service in the XA interface. The *ax_* functions are provided by a TM for use by RMs. The *xa_* functions are provided by each RM for use by the TM.

The symbolic constants and error names are described in the **<xa.h>** header (see Chapter 4). The state tables referred to in the reference manual pages appear in Chapter 6.

NAME

ax_add_branch — generate a new branch for an existing global transaction

SYNOPSIS

```
#include <xa.h>

int ax_add_branch(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager to create a transaction branch.

The *ax_add_branch*( ) function allows a communication resource manager to request the transaction manager to generate a new branch for the current global transaction. The resource manager calls the transaction manager with this function when it needs to propagate an existing global transaction to a different thread of control that is loosely coupled to the current thread of control. The resource manager calls this function multiple times within the same thread of control to create multiple, separate, transaction branches of the same global transaction.

The function's first argument, *rmid,* is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument returned to the communication resource manager is a pointer to the XID that identifies the transaction branch. The communication resource manager uses this XID when propagating this global transaction. The transaction manager guarantees the XID to be unique for different transaction branches by generating a new branch qualifier when *ax_add_branch*( ) is called.

The function's last argument, *flags,* is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_add_branch*( ) has the following return values:

[TM_OK]
Normal execution.

[TMER_TMERR]
The transaction manager encountered an error in generating a new XID.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *ax_get_branch_info*( ), *ax_set_branch_info*( ).

NAME

ax_commit — propagate transaction branch commitment to a transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_commit(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can propagate the two-phase commit protocol for transaction branches. The *ax_commit*( ) function notifies the transaction manager that a commit indication has been received from a superior. The transaction manager, in turn, issues an *xa_commit*( ) to all local resource managers registered for the transaction branch and all branches subordinate to XID that were created with *ax_add_branch*( ). Any thread of control may invoke *ax_commit*( ).

The function's first argument, *rmid,* is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument is a pointer to the XID that identifies the transaction branch being committed.

If the communication resource manager is committing a transaction branch while transaction chaining is in effect, the communication resource manager sets the TMCHAINED flag. *xid*[0] is the transaction branch to commit, and *xid*[1] is the chained transaction branch to assign. If the transaction manager returns any [TMER_*] negative return value, the new transaction branch is not assigned.

The communication resource manager begins local work on the new transaction later by issuing *ax_start*( ) to the transaction manager in the application thread of control. The transaction manager then issues *xa_start*( ) to statically registering resource managers. Dynamically registering resource managers need to register using *ax_reg*( ) in the application thread of control when application work in the resource manager begins.

The following are valid settings of *flags*:

TMONEPHASE
: The communication resource manager must set this flag if it is using the one-phase commit optimisation for the specified transaction branch.

TMCHAINED
: This flag indicates that processing is in chained transaction mode, and in addition to committing the existing transaction branch, a new XID is provided for the next transaction. *xid*[0] is the XID to commit. *xid*[1] is the XID to assign for the next transaction.

TMRECOVER
: This flag indicates that this operation is being performed to recover a subordinate branch. The transaction manager sets this same flag when it calls *xa_commit*( ) for subordinate branches.

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_commit*( ) has the following return values:

[TM_HEURHAZ]
Due to some failure, it is not known whether all subordinates in the transaction branch performed the same operation (commit or rollback). One or more may be in danger of making an inconsistent heuristic decision.

[TM_HEURCOM]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed.

[TM_HEURRB]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back.

[TM_HEURMIX]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back.

[TM_OK]
Normal execution.

[TM_RB*]
The transaction manager did not commit the work on behalf of the transaction branch. Upon return, the transaction manager has rolled back the branch's work and released all held resources. The following values may be returned only if TMONEPHASE is set in *flags*:

[TM_RBROLLBACK]
The transaction manager marked the transaction branch rollback-only for an unspecified reason.

[TM_RBCOMMFAIL]
A communication failure occurred within a resource manager.

[TM_RBDEADLOCK]
A resource manager detected a deadlock.

[TM_RBINTEGRITY]
The transaction manager or a resource manager detected a violation of the integrity of its resources.

[TM_RBOTHER]
The transaction manager or a resource manager marked the transaction branch rollback-only for a reason not on this list.

[TM_RBPROTO]
A protocol error occurred within a resource manager.

[TM_RBTIMEOUT]
The work represented by this transaction branch took too long.

[TM_RBTRANSIENT]
A resource manager detected a transient error.

[TMER_TMERR]
The transaction manager encountered an error while trying to commit the transaction branch.

[TMER_NOTA]
The specified XID is not known by the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_prepare( ), *xa_commit*( ), *xa_rollback*( ), *ax_add_branch*( ), *ax_prepare*( ), *ax_rollback*( ).

NAME

ax_done — report that recovery associated with a transaction branch is complete

SYNOPSIS

```
#include <xa.h>

int ax_done(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager to inform a transaction manager that recovery is complete with a transaction branch after a disruption, and to report the outcome of the recovery.

The *ax_done*( ) function allows the transaction manager to mark the branch as completed. The communication resource manager calls the transaction manager when requested by a subordinate during recovery processing after a communication failure.

If heuristic decisions are detected during recovery processing initiated either by the subordinate or by a communication resource manager using the presumed-nothing protocol, the function *ax_done*( ) may be called after the transaction manager has rolled back the transaction and forgotten the transaction branch. The TM is assumed to know the outcome of the transaction, so this function only reports discrepancies that occurred in the transaction branch.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The second argument, *xid*, is a pointer to the XID that identifies the transaction branch. The XID must be the same as one generated by the transaction manager when the communication resource manager issued an *ax_add_branch*( ).

The function's last argument, *flags*, must be set to the following value:

TMRECOVER
: This flag indicates that this operation is being performed because of recovery at a subordinate branch.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_done*( ) has the following return values:

[TM_HEURHAZ]
: Due to some failure, it is not known whether all subordinates in the transaction branch performed the same operation (commit or rollback). One or more may be in danger of making an inconsistent heuristic decision.

[TM_HEURCOM]
: Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed. This may or may not be consistent with the outcome at the coordinator.

[TM_HEURRB]
: Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back. This may or may not be consistent with the outcome at the coordinator.

[TM_HEURMIX]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back.

[TM_OK]
Normal execution.

[TMER_TMERR]
The transaction manager encountered an error.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_done( ), *xa_open*( ), *xa_ready*( ), *ax_add_branch*( ), *ax_commit*( ), *ax_ready*( ), *ax_rollback*( ).

NAME

ax_end — notify a transaction manager to end work performed on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int ax_end(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can propagate transaction branches. The *ax_end*( ) function notifies a transaction manager that the thread of control has finished, or needs to suspend work on, a transaction branch. This occurs when the application completes a portion of its work, either in its entirety or partially (for example, before blocking on some event in order to let other threads of control work on the branch). When *ax_end*( ) successfully returns, the calling thread of control is dissociated from the branch.

The communication resource manager must be superior to the transaction manager to completely end an association either successfully or unsuccessfully. A communication resource manager that is subordinate to a transaction manager suspends an association before blocking on a communication event.

The transaction manager notifies the resource manager that the transaction branch is being dissociated from the thread of control (by calling *xa_end*( )).

The function's first argument, *rmid*, is the integer that the the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The second argument, *xid*, is a pointer to the XID that identifies the transaction branch. If the communication resource manager is acting as a subordinate, *xid* must point to the same XID that was either passed on the *xa_start*( ) call or returned from the *ax_reg*( ) call. If the communication resource manager is acting as a superior, *xid* must point to the same XID passed to the transaction manager on the original *ax_start*( ) call that established the thread's association. Otherwise, the transaction manager returns an error ([TMER_NOTA]).

The function's last argument, *flags*, must be set to one of the following values:

TMSUSPEND

Suspend a transaction branch on behalf of the calling thread of control. For a resource manager that allows multiple threads of control, but only one at a time working on a specific global transaction, the *xa_end*( ) call generated by this command gives the resource manager a chance to allow another thread of control to work on the global transaction at this point. If this flag is not accompanied by the TMMIGRATE flag, then the communication resource manager issuing *ax_end*( ) must resume the association in the current thread. TMSUSPEND cannot be used in conjunction with either TMSUCCESS or TMFAIL.

TMMIGRATE

If this flag is used with TMSUSPEND, the communication resource manager intends (but is not required) to resume the association in a thread different from the calling one. If this flag is used with TMSUCCESS, the communication resource manager intends (but is not required) either to perform an *ax_start*( ) with TMJOIN, or *ax_prepare*( ), *ax_commit*( ) or *ax_rollback*( ) in a different thread from the calling one. Setting TMMIGRATE in *flags*, while another thread's association for **xid* is currently suspended with TMMIGRATE, makes *ax_end*( ) fail, returning [TMER_PROTO]. If this flag is not used, a communication resource manager is required to resume the association or perform transaction completion in the current thread.

TMSUCCESS

The portion of work has succeeded. This flag cannot be used in conjunction with TMSUSPEND or TMFAIL.

TMFAIL

The portion of work has failed. The transaction manager marks the transaction branch as rollback-only and passes the TMFAIL flag to the resource managers on the *xa_end*( ) call. In effect, the global transaction is marked for rollback-only at this point. This flag cannot be used in conjunction with TMMIGRATE, TMSUSPEND or TMSUCCESS.

TMSUCCESS and TMFAIL mark the completion of the association of a transaction branch. These *flags* are used only by communication resource managers that are superior to the transaction manager.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_end*( ) has the following return values:

[TM_NOMIGRATE]

A resource manager was unable to prepare the transaction context for migration. However, the transaction manager successfully suspended the association if TMSUSPEND was set in *flags*. In this case, the calling communication resource manager can resume the association only in the current thread. This code may be returned when both TMSUSPEND and TMMIGRATE are set in *flags*.

If the transaction manager (due to some resource manager restrictions) is unable to perform *xa_prepare*( ), *xa_commit*( ) or *xa_rollback*( ) for this transaction in any other thread, it returns TM_NOMIGRATE when TMSUCCESS is set in *flags*. The transaction manager successfully ended the current association.

[TM_RDONLY]

No resource managers statically or dynamically registered for this transaction branch which was begun with *ax_start*( ) with TMNOFLAGS set in flags and ended with *ax_end*( ) with TMSUCCESS set in flags. Or, this transaction branch was begun with *ax_start*( ) with TMDEFERRED set in flags, ended with *ax_end*( ) with TMSUCCESS or TMFAIL set in flags, and the application program did not issue *tx_begin*( ). The transaction manager has removed all record of the transaction branch as if *ax_start*( ) were never issued by the communication resource manager.

[TM_OK]

Normal execution.

[TM_RB*]

The transaction manager has dissociated the transaction branch from the thread of control and has marked rollback-only the work done on behalf of **xid*. The following values may be returned regardless of the setting of *flags*:

[TM_RBROLLBACK]

The transaction manager marked the transaction branch rollback-only for an unspecified reason.

[TM_RBCOMMFAIL]

A communication failure occurred within a resource manager.

[TM_RBDEADLOCK]
A resource manager detected a deadlock.

[TM_RBINTEGRITY]
The transaction manager or a resource manager detected a violation of the integrity of its resources.

[TM_RBOTHER]
The transaction manager or a resource manager marked the transaction branch rollback-only for a reason not on this list.

[TM_RBPROTO]
A protocol error occurred within a resource manager.

[TM_RBTIMEOUT]
The work represented by this transaction branch took too long.

[TM_RBTRANSIENT]
A resource manager detected a transient error.

[TMER_TMERR]
An error occurred in dissociating the transaction branch from the thread of control.

[TMER_NOTA]
The specified XID is not known to the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *xa_end*( ), *ax_start*( ).

NAME

ax_forget_branch — remove a branch from an existing global transaction

SYNOPSIS

```
#include <xa.h>

int ax_forget_branch(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager to remove an unused transaction branch.

The *ax_forget_branch*( ) function allows a communication resource manager to request the transaction manager to remove a branch from the current global transaction. The communication resource manager calls the transaction manager with this function when it propagates the existing global transaction to a different thread of control that runs loosely coupled with the current thread of control, but the transaction branch (generated by the transaction manager when the communication resource manager called *ax_add_branch*( )) was not used.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The second argument, *xid*, is a pointer to the XID that identifies the transaction branch. The XID must be the same as one generated by the transaction manager when the communication resource manager issued an *ax_add_branch*( ).

The function's last argument, *flags*, is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_forget_branch*( ) has the following return values:

[TM_OK]
Normal execution.

[TMER_TMERR]
The transaction manager encountered an error deleting the XID.

[TMER_NOTA]
The specified XID is not known to the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *ax_add_branch*( ).

NAME

ax_get_branch_info — access information for a transaction branch

SYNOPSIS

```
#include <xa.h>

int ax_get_branch_info(int rmid, XID *xid, char *blob,
                       long *blob_len, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can create transaction branches. The *ax_get_branch_info*( ) function allows the communication resource manager to access information for a specific transaction branch that was saved by a transaction manager. The communication resource manager requests information to be saved using the *ax_set_branch_info*( ) call. The communication resource manager may call *ax_get_branch_info*( ) multiple times for the same *xid*.

The first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

xid must point to an XID passed to the transaction manager on an *ax_set_branch_info*( ) call. Otherwise, the transaction manager returns an error ([TMER_NOTA]).

The *blob* argument is a pointer to an area to receive the previously saved information. It is the communication resource manager's responsibility to ensure that the area is big enough to hold the information.

The *blob_len* argument is a pointer to an area in which the transaction manager returns the size of *blob*.

The function's last argument, *flags*, is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_get_branch_info*( ) has the following return values:

[TM_OK]
Normal execution.

[TMER_TMERR]
An error occurred in the transaction manager.

[TMER_NOTA]
The specified XID is not known by the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in the improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *ax_set_branch_info*( ).

NAME

ax_prepare — propagate transaction branch prepare to commit to the transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_prepare(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can propagate transaction branches. The *ax_prepare*( ) function notifies a transaction manager that a prepare to commit indication has been received from a superior. The transaction manager, in turn, issues an *xa_prepare*( ) to all local resource managers registered for the transaction branch and all branches subordinate to XID that were created with *ax_add_branch*( ). Any thread of control may invoke *ax_prepare*( ).

The first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument is a pointer to the XID that identifies the transaction branch being prepared for commitment.

The function's last argument, *flags*, is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_prepare*( ) has the following return values:

[TM_RDONLY]
The transaction branch was read-only. The branch does not need to be included in the second phase of the commit operation. On return, the resource manager has released all held resources.

[TM_OK]
Normal execution.

[TM_RB*]
The transaction manager did not prepare to commit the work done on behalf of the transaction branch. Upon return, the transaction manager has rolled back the branch's work and has released all held resources. The following values may be returned:

[TM_RBROLLBACK]
The transaction manager rolled back the transaction branch for an unspecified reason.

[TM_RBCOMMFAIL]
A communication failure occurred within a resource manager.

[TM_RBDEADLOCK]
A resource manager detected a deadlock.

[TM_RBINTEGRITY]
The transaction manager or a resource manager detected a violation of the integrity of its resources.

[TM_RBOTHER]
The transaction manager or a resource manager rolled back the transaction branch for a reason not on this list.

[TM_RBPROTO]
A protocol error occurred within a resource manager.

[TM_RBTIMEOUT]
The work represented by this transaction branch took too long.

[TM_RBTRANSIENT]
A resource manager detected a transient error.

[TMER_TMERR]
The transaction manager encountered an error while trying to prepare to commit the transaction branch.

[TMER_NOTA]
The specified XID is not known by the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *xa_prepare*( ), *xa_commit*( ), *xa_rollback*( ), *ax_commit*( ), *ax_rollback*( ).

NAME

ax_ready — find out if a subordinate should commit or rollback

SYNOPSIS

```
#include <xa.h>

int ax_ready(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager to get the status of a transaction branch at the request of a subordinate that is attempting recovery from the disruption of the two-phase commit process.

The *ax_ready*( ) function allows the communication resource manager to request a transaction manager to report the status of a global transaction. The communication resource manager calls the transaction manager with this function when it receives a query from a subordinate. The transaction manager responds with either an order to commit or rollback the subordinate or an indication that the outcome of the prepare phase of the two-phase commit process is not yet known.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The second argument, *xid*, is a pointer to the XID that identifies the transaction branch. The XID must be the same as one generated by the transaction manager when the communication resource manager issued an *ax_add_branch*( ).

The function's last argument, *flags*, must be set to the following value:

TMRECOVER
: This flag indicates that this operation is being performed to recover a subordinate branch.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_ready*( ) has the following return values:

[TM_RBROLLBACK]
: The transaction manager marked the transaction branch *rollback-only*. The transaction manager is instructing the CRM to roll back the transaction branch at the subordinate.

[TM_DEFERRED]
: A decision has not yet been made. The transaction manager completes the two-phase commit process later. The subordinate has no need for immediate action.

[TM_OK]
: Normal execution. The transaction manager has marked the transaction branch *decided*. The transaction manager is instructing the CRM to commit the transaction branch at the subordinate.

[TMER_TMERR]
: The transaction manager encountered an error.

[TMER_INVAL]
: Invalid arguments were specified.

[TMER_PROTO]

The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *xa_ready*( ), *ax_add_branch*( ), *ax_commit*( ), *ax_rollback*( ).

NAME

ax_recover — obtain a list of transaction branches from a transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_recover(int rmid, XID *xids, long count, long flags)
```

DESCRIPTION

A communication resource manager calls *ax_recover*( ) to obtain a list of transaction branches known by a transaction manager for which the communication resource manager has previously issued *ax_set_branch_info*( ). The caller points **xids* to an array into which the transaction manager places the XIDs, and sets count to the maximum number of XIDs that fit into that array.

The transaction manager responds to *ax_recover*( ) as though it maintained an ordered list of transaction branches for which it has saved information on behalf of the communication resource manager. Each such branch appears only once in this list, and removal of a branch from the list does not change the relative order of the remaining branches in the list.

Depending on the setting of flags, certain calls to *ax_recover*( ) denote the start and/or end of a *recovery scan*. During a recovery scan, the transaction manager defines a *cursor* marking the current position in the list of transaction branches. The start of a recovery scan moves the cursor to the start of the list. Each call advances the cursor past the set of XIDs returned.

Two consecutive complete recovery scans return the same list of transaction branches unless an intervening event allows the transaction manager to forget some transaction branches.

The communication resource manager may call this function from any thread of control, but all calls in a given recovery scan must be made from the same thread.

Upon success, *ax_recover*( ) places zero or more XIDs in the space pointed to by **xids*. The function returns the number of XIDs it has placed there. If this value is less than count, there are no more XIDs to recover and the current scan ends. (That is, the communication resource manager need not call *ax_recover*( ) again with TMENDRSCAN set in flags.) Multiple invocations of *ax_recover*( ) are used to retrieve all the transaction branches for which the transaction manager has information saved on behalf of the communication resource manager.

The function’s first argument, *rmid*, is the integer the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The following are valid settings of *flags*:

TMSTARTRSCAN

This flag indicates that *ax_recover*( ) should start a recovery scan for the thread of control and position the cursor to the start of the list. XIDs are returned from that point. If a recovery scan is already open, the effect is as if the recovery scan were ended and then restarted.

TMENDRSCAN

This flag indicates that *ax_recover*( ) should end the recovery scan after returning the XIDs. If this flag is used in conjunction with TMSTARTRSCAN, the single *ax_recover*( ) call starts and then ends a scan.

TMNOFLAGS

This flag must be used when no other flags are set in flags. A recovery scan must already be started. XIDs are returned starting at the current cursor position.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_recover*( ) has the following return values:

[≥ 0]

As a return value, *ax_recover*( ) normally returns the total number of XIDs it returned in **xids*.

[TMER_TMERR]

An error occurred in determining the XIDs to return.

[TMER_INVAL]

xids is null and count is greater than 0; *count* is negative; an invalid flags was specified; or the thread of control does not have a recovery scan open and did not specify TMSTARTRSCAN in flags.

[TMER_PROTO]

The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

ax_set_branch_info( ), *ax_get_branch_info*( ).

WARNINGS

If *xids* points to a buffer that cannot hold all of the XIDs requested, then *ax_recover*( ) may overwrite the caller's data space.

NAME

ax_reg, ax_reg_2 — dynamically register a resource manager with a transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_reg(int rmid, XID *xid, long flags)

int ax_reg_2(int rmid, XID *xid, XACTL *ctl, long flags)
```

DESCRIPTION

A resource manager calls *ax_reg*( ) to inform a transaction manager that it is about to perform work on behalf of an application thread of control. The transaction manager, in turn, replies to the resource manager with an indication of whether or not that work should be performed on behalf of a transaction branch. If the transaction manager determines that the calling thread of control is involved in a branch then, upon successful return, *xid* points to a valid XID. If the resource manager's work is outside any global transaction, then *xid* points to NULLXID. The caller is responsible for allocating the space to which *xid* points.

The resource manager must call this function from the same thread of control from which the application accesses the resource manager. A resource manager taking advantage of this facility must have TMREGISTER set in the *flags* element of its **xa_switch_t** structure (see Chapter 4). Moreover, *ax_reg*( ) returns failure [TMER_TMERR] when issued by a resource manager that has not set TMREGISTER.

When the resource manager calls *ax_reg*( ) for a new thread of control association (that is, when [TM_RESUME] is not returned; see below), the transaction manager may generate a unique branch qualifier within the returned XID.

If the transaction manager elects to reuse within **xid* a branch qualifier previously given to the resource manager, it informs the resource manager of this by returning [TM_JOIN]. If the resource manager receives [TM_JOIN] and does not recognise **xid*, it must return a failure indication to the application.

If the resource manager is resuming work on a suspended transaction branch, it informs the resource manager of this by returning [TM_RESUME]. When [TM_RESUME] is returned, *xid* points to the same XID that was passed to the *xa_end*( ) call that suspended the association. If the resource manager receives [TM_RESUME] and does not recognise **xid*, it must return a failure indication to the application.

If the transaction manager generated a new branch qualifier within the returned XID, this thread is loosely-coupled in relation to the other threads in this same global transaction. That is, the resource manager may treat this thread's work as a separate transaction with respect to its isolation policies. If the transaction manager reuses a branch qualifier within the returned XID, this thread is tightly-coupled to the other threads in the same transaction branch. The resource manager must guarantee that tightly-coupled threads are treated as a single entity with respect to its isolation policies and that no resource deadlock can occur within the transaction branch among these tightly-coupled threads.

The implications of dynamically registering are as follows: when a thread of control begins working on behalf of a transaction branch, the transaction manager calls *xa_start*( ) for all resource managers known to the thread except those having TMREGISTER set in their **xa_switch_t** structure. Thus, those resource managers with this flag set must explicitly join a branch with *ax_reg*( ). Secondly, when a thread of control is finished working on behalf of a branch, the transaction manager calls *xa_end*( ) for all resource managers known to the thread that either do not have TMREGISTER set in their **xa_switch_t** structure or have dynamically registered with *ax_reg*( ).

The function's first argument, *rmid,* is the integer that the resource manager received when the transaction manager called *xa_open*( ). It identifies the resource manager in the thread of control.

The function's last argument, *flags,* is reserved for future use and must be set to TMNOFLAGS.

In order to promote migration to future extensions, the resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags.*

On input to *ax_reg_2*( ), the resource manager sets flag bits in *ctl→flags* to inform the transaction manager of the options in which it is interested. The following settings are valid:

XAOPTS_NOFLAGS

No flags or values are set. The resource manager is not interested in finding out any extra information from the transaction manager. This is equivalent to calling *ax_reg*( ).

XAOPTS_TIMEOUT

Setting this flag indicates that the resource manager wants the transaction manager to set in *ctl→timeout* the number of seconds before which the transaction can be timed out and rolled back.

On output, the transaction manager populates the items that the resource manager has requested. If the transaction manager does not have a value for a setting that the resource manager requested, it turns that setting off on output. In this way, the resource manager can determine exactly which values the transaction manager returned.

RETURN VALUE

The function *ax_reg*( ) has the following return values:

[TM_JOIN]

The resource manager is joining the work of an existing transaction branch. The resource manager should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the branch. If the resource manager does not recognise **xid,* it must return a failure indication to the application.

[TM_RESUME]

The resource manager should resume work on a previously-suspended transaction branch. The resource manager should make available at least the transaction context that is specific to the resource manager, present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

If the resource manager does not recognise **xid,* then it must return a failure indication to the application. If the resource manager allows an association to be resumed in a different thread from the one that suspended the work, and the transaction manager expressed its intention to migrate the association (by means of the TMMIGRATE flag on *xa_end*( )), then the current thread may be different from the one that suspended the work. Otherwise, the current thread must be the same, or the resource manager must return a failure indication to the application.

If **xid* contains a reused branch qualifier, and the transaction manager has multiple outstanding suspended thread associations for **xid,* the following rules apply:

- The transaction manager can have only one of them outstanding at any time with TMMIGRATE set in *flags.*

- Moreover, the transaction manager cannot resume this association in a thread that currently has a non-migratable suspended association.

These rules prevent ambiguity as to which context is restored.

[TM_OK]
Normal execution.

[TMER_TMERR]
The transaction manager encountered an error in registering the resource manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

ax_unreg( ), *xa_end*( ), *xa_open*( ), *xa_start*( ).

WARNINGS

If *xid* does not point to a buffer that is at least as large as the size of an XID, then *ax_reg*( ) may overwrite the caller's data space. In addition, the buffer must be properly aligned (on a long word boundary) in the event that structure assignments are performed.

NAME

ax_rollback — propagate transaction branch rollback to a transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_rollback(int rmid, XID *xid, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can propagate the two-phase commit protocol for transaction branches. The *ax_rollback*( ) function notifies a transaction manager that a rollback indication has been received from a superior. The transaction manager, in turn, issues an *xa_rollback*( ) to all local resource managers registered for the transaction branch and all branches subordinate to the XID pointed to by *xid* that were created with *ax_add_branch*( ). Any thread of control may invoke *ax_rollback*( ).

The function's first argument, *rmid,* is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument is a pointer to the XID that identifies the transaction branch being rolled back.

If the communication resource manager is rolling back a transaction branch while transaction chaining is in effect, the communication resource manager sets the TMCHAINED flag. *xid*[0] is the transaction branch to roll back, and *xid*[1] is the chained transaction branch to assign. If the transaction manager returns any [TMER_*] negative return value, the new transaction branch is not assigned.

The communication resource manager begins local work on the new transaction later by issuing *ax_start*( ) to the transaction manager in the application thread of control. The transaction manager then issues *xa_start*( ) to statically registering resource managers. Dynamically registering resource managers need to register using *ax_reg*( ) in the application thread of control when application work in the resource manager begins.

The function's last argument, *flags*, must be set to one of the following values:

TMCHAINED

This flag indicates that processing is in chained transaction mode, and in addition to rolling back the existing transaction branch, a new XID is provided for the next transaction. *xid*[0] is the XID to roll back. *xid*[1] is the XID to assign for the next transaction.

TMRECOVER

This flag indicates that this operation is being performed to recover a subordinate branch. The transaction manager sets this same flag when it calls *xa_rollback*( ) for subordinate branches.

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags.*

RETURN VALUE

The function *ax_rollback*( ) has the following return values:

[TM_HEURHAZ]

Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed.

[TM_HEURCOM]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed.

[TM_HEURRB]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back.

[TM_HEURMIX]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back.

[TM_OK]
Normal execution.

[TM_RB*]
The transaction manager has rolled back the transaction branch's work releasing all held resources. These values are typically returned when the branch was already marked rollback-only. The following values may be returned:

[TM_RBROLLBACK]
The transaction manager or a resource manager rolled back the transaction branch for an unspecified reason.

[TM_RBCOMMFAIL]
A communication failure occurred within a resource manager.

[TM_RBDEADLOCK]
A resource manager detected a deadlock.

[TM_RBINTEGRITY]
The transaction manager or a resource manager detected a violation of the integrity of its resources.

[TM_RBOTHER]
The transaction manager or a resource manager rolled back the transaction branch for a reason not on this list.

[TM_RBPROTO]
A protocol error occurred within a resource manager.

[TM_RBTIMEOUT]
The work represented by this transaction branch took too long.

[TM_RBTRANSIENT]
A resource manager detected a transient error.

[TMER_TMERR]
The transaction manager encountered an error while trying to roll back the transaction branch.

[TMER_NOTA]
The specified XID is not known to the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *xa_prepare*( ), *xa_commit*( ), *xa_rollback*( ), *ax_prepare*( ), *ax_commit*( ), *ax_add_branch*( ).

NAME

ax_set_branch_info — save information for a transaction branch

SYNOPSIS

```
#include <xa.h>

int ax_set_branch_info(int rmid, XID *xid, char *blob, long blob_len,
                       long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can create transaction branches.

The *ax_set_branch_info*( ) function allows the communication resource manager to request a transaction manager to save information for a specific transaction branch. A communication resource manager that is subordinate to the transaction manager can save information for branches it creates from *ax_add_branch*( ) calls, and a communication resource manager that is superior to the transaction manager can save information for the transaction branch that is currently active in the thread of control. The communication resource manager can retrieve this information later (for example, during recovery) by calling *ax_get_branch_info*( ). The communication resource manager may call this function multiple times for the same XID. Each subsequent call requests the transaction manager to replace previously saved information for the transaction branch.

The transaction manager records this information stably at the completion of the prepare-to-commit process for the superior transaction branch. This service is not intended to provide a general-purpose logging facility for resource managers. It simply provides a mechanism to save information for communication resource managers that would ordinarily not require any other logging service.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument is a pointer to the XID that identifies the transaction branch for which this information is to be saved. *xid* must point to the same XID received when the communication resource manager called *ax_reg*( ), *ax_add_branch*( ), or when the transaction manager called the communication resource manager function, *xa_start*( ), or the same XID passed to the transaction manager when the communication resource manager called *ax_start*( ).

The *blob* argument points to a character string of information to be saved.

The *blob_len* argument contains the size of *blob*. The function returns [TMER_INVAL] if *blob_len* is greater than TMMAXBLOBLEN. The function returns [TMER_TMERR] if this *blob_len* forces TMMAXBLOBTOT to be exceeded.

The function's last argument, *flags*, is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_set_branch_info*( ) has the following return values:

[TM_OK]

Normal execution.

[TMER_TMERR]
An error occurred in the transaction manager.

[TMER_NOTA]
The specified XID is not known by the transaction manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in the improper context. See Chapter 6 for details.

SEE ALSO

xa_start( ), *ax_start*( ), *ax_add_branch*( ), *ax_reg*( ), *ax_get_branch_info*( ).

NAME

ax_start, ax_start_2 — notify a transaction manager to propagate a transaction branch association or to resume the branch association in this thread of control

SYNOPSIS

```
#include <xa.h>

int ax_start(int rmid, XID *xid, long flags)

int ax_start_2(int rmid, XID *xid, XACTL *ctl, long flags)
```

DESCRIPTION

This function is called by a communication resource manager that can propagate transaction branches.

When *ax_start*( ) is called with TMNOFLAGS, the communication resource manager is propagating a new transaction branch to this thread of control. The transaction manager accepts the XID passed by the communication resource manager, but the transaction manager may map the branch qualifier portion of the XID to one of its own. The communication resource manager finds out about the mapping when its *xa_start*( ) function is subsequently called or when it calls *ax_reg*( ).

When *ax_start*( ) is called with the TMRESUME flag, the communication resource manager is resuming the association of a transaction branch that was previously suspended by an *ax_end*( ) call with the TMSUSPEND flag set. The association is being resumed in the same thread of control where the suspension occurred if TMMIGRATE was set on *ax_end*( ), and migration was not permitted by the transaction manager (that is, TM_NOMIGRATE was returned). Otherwise, the resumption may be in a different thread of control.

When *ax_start*( ) is called with the TMJOIN flag, the communication resource manager is beginning an association with a transaction branch that was previously begun in this or another thread of control under the supervision of the transaction manager. TMJOIN allows threads of control to share the same XID serially or in parallel. XID-sharing can be among an application and one or more subordinate applications it calls or among a set of subordinate applications independent of the calling application.

When *ax_start*( ) is called with the TMDEFERRED flag, the communication resource manager is deferring the propagation of a new transaction branch in this thread of control pending the agreement of the AP. The AP agrees to participate in the transaction by issuing *tx_begin*( ) later. When this happens, the transaction manager proceeds as if *ax_start*( ) were called with TMNOFLAGS set in flags.

The communication resource manager calls *ax_start*( ) to instruct the transaction manager to notify resource managers that an application may do work on behalf of a transaction branch. The transaction manager does so by calling *xa_start*( ) for resource managers that do not have TMREGISTER set in the *flags* element of their **xa_switch_t** structure. The transaction manager passes on the *flags* settings it received on the *ax_start*( ) call when it issues *xa_start*( ) calls.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The *xid* argument is a pointer to the XID that identifies the transaction branch whose association is being started or resumed in the thread of control. The XID must not have been previously associated with a thread by the transaction manager if a new branch is being propagated (the *flags* argument set to TMNOFLAGS). The XID must have been previously associated with a thread by the transaction manager if the setting of *flags* is either TMRESUME or TMJOIN.

The following are valid settings of flags:

TMJOIN

This flag indicates that a transaction branch that has previously been associated with this or another thread of control under the supervision of the transaction manager is being propagated to this thread. This thread is joining a set of tightly-coupled threads sharing a common XID. This flag cannot be used in conjunction with TMRESUME.

TMRESUME

This flag indicates that an operation which suspended the previously associated transaction branch is complete and the association is being resumed. This flag cannot be used in conjunction with TMJOIN.

TMDEFERRED

This flag indicates that the propagation of a new transaction branch to this thread of control is pending acceptance by the application program.

TMNOFLAGS

This flag value is used when other settings of flags do not apply.

In order to promote migration to future extensions, the communication resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

On input to *ax_start_2*( ), the communication resource manager sets flag bits in *ctl*→*flags* to inform the transaction manager of the options that are being passed. The following settings are valid:

XAOPTS_NOFLAGS

No flags or values are set. The communication resource manager has passed no information to control the transaction manager. This is equivalent to calling *ax_start*( ).

XAOPTS_TIMEOUT

Setting this flag indicates that the communication resource manager has placed in *ctl*→*timeout* the number of seconds before which the transaction manager can timeout and rollback the transaction branch.

RETURN VALUE

The function *ax_start*( ) has the following return values:

[TM_OK]

Normal execution.

[TM_RB*]

The transaction manager has not associated the transaction branch with the thread of control and has marked **xid* rollback-only. The following values may be returned regardless of the setting of *flags*:

[TM_RBROLLBACK]

The transaction manager marked the transaction branch rollback-only for an unspecified reason.

[TM_RBCOMMFAIL]

A communication failure occurred within a resource manager.

[TM_RBDEADLOCK]

A resource manager detected a deadlock.

[TM_RBINTEGRITY]
The transaction manager or a resource manager detected a violation of the integrity of its resources.

[TM_RBOTHER]
The transaction manager or a resource manager marked the transaction branch rollback-only for a reason not on this list.

[TM_RBPROTO]
A protocol error occurred within a resource manager.

[TM_RBTIMEOUT]
The work represented by this transaction branch took too long.

[TM_RBTRANSIENT]
A resource manager detected a transient error.

[TMER_TMERR]
The transaction manager encountered an error while trying to associate **xid* with the thread of control.

[TMER_NOTA]
Either TMJOIN or TMRESUME was set in *flags*, and the specified XID has not been previously associated by the transaction manager with this or another thread of control.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[TMER_DUPID]
Neither TMJOIN nor TMRESUME was set in *flags*, but the transaction manager has previously associated **xid* with this or another thread of control. The transaction manager did perform the association of this thread with the transaction branch.

SEE ALSO

xa_open( ), *xa_start*( ), *xa_end*( ), *ax_end*( ), *ax_reg*( ).

NAME

ax_unreg — dynamically unregister a resource manager with a transaction manager

SYNOPSIS

```
#include <xa.h>

int ax_unreg(int rmid, long flags)
```

DESCRIPTION

A resource manager calls *ax_unreg*( ) to inform a transaction manager that it has completed work, outside any global transaction, that it began after receiving the NULLXID from *ax_reg*( ). In addition, the resource manager is informing the transaction manager that the accessing thread of control is free to participate (from the resource manager's perspective) in a global transaction. So long as any resource manager in a thread of control is registered with a transaction manager and is performing work outside any global transaction, that application thread cannot participate in a global transaction.

A resource manager must call this function from the same thread of control that originally called *ax_reg*( ). A resource manager taking advantage of this facility must have TMREGISTER set in the *flags* element of its **xa_switch_t** structure (see Chapter 4). Moreover, *ax_unreg*( ) returns failure [TMER_TMERR] when issued by a resource manager that has not set TMREGISTER.

The function's first argument, *rmid*, is the integer that the resource manager received when the transaction manager called *xa_open*( ). It identifies the resource manager in the thread of control.

The function's last argument, *flags*, is reserved for future use and must be set to TMNOFLAGS.

In order to promote migration to future extensions, the resource manager should not set any unused flags, and the transaction manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *ax_unreg*( ) has the following return values:

[TM_OK]
Normal execution.

[TMER_TMERR]
The transaction manager encountered an error in unregistering the resource manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

ax_reg( ).

NAME

xa_close — close a resource manager

SYNOPSIS

```
#include <xa.h>

int xa_close(char *xa_info, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_close*( ) to close a currently open resource manager in the calling thread of control. Once closed, the resource manager cannot participate in global transactions on behalf of the calling thread until it is re-opened.

The argument *xa_info* points to a null-terminated character string that may contain instance-specific information for the resource manager. The maximum length of this string is 256 bytes (including the null terminator). The argument *xa_info* may point to an empty string if the resource manager does not require instance-specific information to be available. The argument *xa_info* must not be a null pointer.

The transaction manager must call this function from the same thread of control that accesses the resource manager. In addition, attempts to close a resource manager that is already closed have no effect and return success, [XA_OK].

It is an error [XAER_PROTO] for the transaction manager to call *xa_close*( ) within a thread of control that is associated with a transaction branch (that is, the transaction manager must call *xa_end*( ) before calling *xa_close*( )). In addition, if the transaction manager calls *xa_close*( ) while an asynchronous operation is pending at a resource manager, an error [XAER_PROTO] is returned.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC

This flag indicates that *xa_close*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_close*( ) has the following return values:

[XA_OK]

Normal execution.

[XAER_ASYNC]

TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred when closing the resource.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_complete( ), *xa_end*( ), *xa_open*( ).

WARNINGS

From the resource manager's perspective, the pointer *xa_info* is valid only for the duration of the call to *xa_close*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xa_info* points. Resource managers are encouraged to use private copies of **xa_info* after the function completes.

NAME

xa_commit — commit work done on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_commit(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_commit*( ) to commit the work associated with **xid*. Any changes made to resources held during the transaction branch are made permanent. The transaction manager may call this function from any thread of control. All associations for **xid* must have been ended by using *xa_end*( ) with TMSUCCESS set in *flags*.

If a resource manager already completed the work associated with **xid* heuristically, this function merely reports how the resource manager completed the transaction branch. The resource manager cannot forget about a heuristically completed branch until the transaction manager calls *xa_forget*( ).

The transaction manager must issue a separate *xa_commit*( ) for each transaction branch that accessed the resource manager.

If the transaction manager is committing a transaction branch while transaction chaining is in effect and the resource manager supports chained transactions, the transaction manager sets the TMCHAINED flag. *xid*[0] is the transaction branch to commit, and *xid*[1] is the chained transaction branch to assign. If the resource manager returns any [XAER_*] negative return value, or [XA_RETRY] or [XA_RETRY_COMMFAIL], the new transaction branch is not assigned.

The transaction manager begins local work on the new transaction later by issuing *xa_start*( ) in the application thread of control for statically registering resource managers. Dynamically registering resource managers need to register using *ax_reg*( ) in the application thread of control when application work in the resource manager begins.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags* (note that at most one of TMNOWAIT and TMASYNC may be set):

TMNOWAIT

When this flag is set and a blocking condition exists, *xa_commit*( ) returns [XA_RETRY] and does not commit the transaction branch (that is, the call has no effect). The function *xa_commit*( ) must be called at a later time to commit the branch. TMNOWAIT is ignored if TMONEPHASE is set.

TMASYNC

This flag indicates that *xa_commit*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMONEPHASE

The transaction manager must set this flag if it is using the one-phase commit optimisation for the specified transaction branch.

TMCHAINED

This flag indicates that processing is in chained transaction mode, and in addition to committing the existing transaction branch, a new XID is provided for the next transaction. *xid*[0] is the XID to commit. *xid*[1] is the XID to assign for the next transaction.

TMRECOVER

This flag indicates that this operation is being performed to recover a subordinate branch.

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_commit*( ) has the following return values:

[XA_TWOPHASE]

The communication resource manager requires the use of two-phase commit. The transaction manager must restart the commit procedure using two-phase commit by calling *xa_prepare*( ).

[XA_RETRY_COMMFAIL]

The resource manager is not able to commit the transaction branch at this time. This value is returned when a subordinate cannot be contacted due to communications failure. If previously prepared, all resources held on behalf of **xid* remain in a prepared state until commitment is possible. The transaction manager should re-issue *xa_commit*( ) at a later time. The subordinate may query the transaction manager using *ax_ready*( ) when communications are re-established in order to finish the commit process.

When [XA_RETRY_COMMFAIL] is returned to the TM, it returns [TX_HAZARD] to the application.

[XA_HEURHAZ]

Due to some failure, it is not known whether all subordinates in the transaction branch performed the same operation (commit or rollback). One or more may be in danger of making an inconsistent heuristic decision.

[XA_HEURCOM]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed.

[XA_HEURRB]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back.

[XA_HEURMIX]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back.

[XA_RETRY]

The resource manager is not able to commit the transaction branch at this time. This value may be returned when a blocking condition exists and TMNOWAIT was set. Note, however, that this value may also be returned even when TMNOWAIT is not set (for example, if the necessary stable storage is currently unavailable). This value cannot be returned if TMONEPHASE is set in *flags*. All resources held on behalf of **xid* remain in a prepared state until commitment is possible. The transaction manager should re-issue *xa_commit*( ) at a later time.

[XA_OK]

Normal execution.

[XA_RB*]

The resource manager did not commit the work done on behalf of the transaction branch. Upon return, the resource manager has rolled back the branch's work and has released all held resources. These values may be returned only if TMONEPHASE is set in *flags*:

[XA_RBROLLBACK]

The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]

A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]

The resource manager detected a deadlock.

[XA_RBINTEGRITY]

The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]

The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]

A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]

The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]

The resource manager detected a transient error.

[XAER_ASYNC]

TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]

An error occurred in committing the work performed on behalf of the transaction branch and the branch's work has been rolled back. Note that returning this error signals a catastrophic event to the transaction manager since other resource managers may successfully commit their work on behalf of this branch. This error should be returned only when a resource manager concludes that it can never commit the branch and that it cannot hold the branch's resources in a prepared state. Otherwise, [XA_RETRY] should be returned.

[XAER_NOTA]

The specified XID is not known to the resource manager.

[XAER_INVAL]

Invalid arguments were specified.

[XAER_PROTO]

The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]

An error occurred that makes the resource manager unavailable.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

ax_ready( ), *xa_complete*( ), *xa_forget*( ), *xa_open*( ), *xa_prepare*( ), *xa_rollback*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_commit*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_complete — wait for an asynchronous operation to complete

SYNOPSIS

```
#include <xa.h>

int xa_complete(int *handle, int *retval, int rmid,
        long flags)
```

DESCRIPTION

A transaction manager calls *xa_complete*( ) to wait for the completion of an asynchronous operation. By default, this function waits for the operation pointed to by *handle* to complete. The argument **handle* must have previously been returned by a function that had TMASYNC set. In addition, the transaction manager must call *xa_complete*( ) from the same thread of control that received **handle*.

Upon successful return, [XA_OK], *retval* points to the return value of the asynchronous operation and **handle* is no longer valid. If *xa_complete*( ) returns any other value, **handle*, **retval*, and any outstanding asynchronous operation are not affected. The caller is responsible for allocating the space to which *handle* and *retval* point.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMMULTIPLE

When this flag is set, *xa_complete*( ) tests for the completion of any outstanding asynchronous operation. Upon success, the resource manager places the handle of the completed asynchronous operation in the area pointed to by **handle*.

TMNOWAIT

When this flag is set, *xa_complete*( ) tests for the completion of an operation without blocking. That is, if the operation denoted by **handle* (or any operation, if TMMULTIPLE is also set) has not completed, *xa_complete*( ) returns [XA_RETRY] and does not wait for the operation to complete.

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_complete*( ) has the following return values:

[XA_RETRY]

TMNOWAIT was set in *flags* and no asynchronous operation has completed.

[XA_OK]

Normal execution.

[XAER_INVAL]

Invalid arguments were specified.

[XAER_PROTO]

The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_close( ), *xa_commit*( ), *xa_end*( ), *xa_forget*( ), *xa_open*( ), *xa_prepare*( ), *xa_rollback*( ), *xa_start*( ).

NAME

xa_done — report that a transaction branch has been committed

SYNOPSIS

```
#include <xa.h>

int xa_done(XID *xid, int rmid, long flags)
```

DESCRIPTION

This function is called by a transaction manager to inform a communication resource manager that a transaction branch has performed the commit phase of the two-phase commit process after a disruption.

The *xa_done*( ) function orders the communication resource manager to inform the superior transaction manager that the branch was committed after a disruption during recovery processing.

The first argument, *xid*, is a pointer to the XID that identifies the transaction branch. The XID must be the same as one generated by the transaction manager when the communication resource manager issued an *ax_start*( ).

The function's second argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

Following are the valid settings of *flags*:

TMRECOVER

This flag indicates that this operation is being performed because of recovery at a subordinate branch.

Note: TMNOFLAGS is not valid.

TMASYNC

This flag indicates that *xa_done*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, then this function fails, returning XAER_ASYNC.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the communication resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_done*( ) has the following return values:

[XA_OK]

Normal execution.

[XAER_ASYNC]

TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]

The resource manager encountered an error.

[XAER_INVAL]

Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_open( ), *xa_ready*( ), *ax_done*( ), *ax_ready*( ), *ax_rollback*( ), *ax_start*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_done*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_end — end work performed on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_end(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_end*( ) when a thread of control finishes, or needs to suspend work on, a transaction branch. This occurs when the application completes a portion of its work, either partially or in its entirety (for example, before blocking on some event in order to let other threads of control work on the branch). When *xa_end*( ) successfully returns, the calling thread of control is dissociated from the branch but the branch still exists. The transaction manager must call this function from the same thread of control that accesses the resource manager.

The transaction manager always calls *xa_end*( ) for those resource managers that do not have TMREGISTER set in the *flags* element of their **xa_switch_t** structure. Unlike *xa_start*( ), *xa_end*( ) is also issued to those resource managers that have previously registered with *ax_reg*( ). After the transaction manager calls *xa_end*( ), it should no longer consider the calling thread associated with that resource manager (although it must consider the resource manager part of the transaction branch when it prepares the branch.) Thus, a resource manager that dynamically registers must re-register after an *xa_end*( ) that suspends its association (that is, after an *xa_end*( ) with TMSUSPEND set in *flags*) but before the application thread of control continues to access the resource manager.

The first argument, *xid*, is a pointer to an XID. The argument *xid* must point to the same XID that was either passed to the *xa_start*( ) call or returned from the *ax_reg*( ) call that established the thread's association; otherwise, an error [XAER_NOTA] is returned.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags* (note that one and only one of TMSUSPEND, TMSUCCESS or TMFAIL must be set):

TMSUSPEND

Suspend a transaction branch on behalf of the calling thread of control. For a resource manager that allows multiple threads of control, but only one at a time working on a specific branch, it might choose to allow another thread of control to work on the branch at this point. If this flag is not accompanied by the TMMIGRATE flag, the transaction manager must resume or end the suspended association in the current thread. TMSUSPEND cannot be used in conjunction with either TMSUCCESS or TMFAIL.

TMMIGRATE

The transaction manager intends (but is not required) to resume the association in a thread different from the calling one. This flag may be used only in conjunction with TMSUSPEND and only if a resource manager does not have TMNOMIGRATE set in the *flags* element of its **xa_switch_t** structure. Setting TMMIGRATE in flags, while another thread's association for **xid* is currently suspended with TMMIGRATE, makes *xa_end*( ) fail, returning [XAER_PROTO]. This is because the transaction manager can have at any given time at most one suspended thread association migrating for a particular transaction branch. If this flag is not used, the transaction manager is required to resume the association in the current thread.

TMSUCCESS
The portion of work has succeeded. This flag cannot be used in conjunction with either TMSUSPEND or TMFAIL.

TMFAIL
The portion of work has failed. A resource manager might choose to mark a transaction branch as rollback-only at this point. In fact, the transaction manager does so for the global transaction. If a resource manager chooses to do so also, then *xa_end*( ) returns one of the [XA_RB*] values. TMFAIL cannot be used in conjunction with either TMSUSPEND or TMSUCCESS.

TMASYNC
This flag indicates that *xa_end*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_end*( ) has the following return values:

[XA_NOMIGRATE]
The resource manager was unable to prepare the transaction context for migration. However, the resource manager has suspended the association. The transaction manager can resume the association only in the current thread. This code may be returned when both TMSUSPEND and TMMIGRATE are set in *flags*. A resource manager that sets TMNOMIGRATE in the *flags* element of its **xa_switch_t** structure need not return [XA_NOMIGRATE].

If a resource manager is unable to perform *xa_prepare*( ), *xa_commit*( ) and *xa_rollback*( ) for this transaction in any other process, it returns [XA_NOMIGRATE] if *flags* is set to TMSUCCESS regardless of the setting of *flags* in its **xa_switch_t** structure.

[XA_RDONLY]
The communication resource manager does not need to participate in the two-phase commit procedure for **xid*. The communication resource manager usually needs to participate only in the two-phase commit procedure for the transaction branches it created by means of *ax_add_branch*( ). By returning this value, the communication resource manager is permitting an optimisation in the two-phase commit procedure by the transaction manager. It may even result in the transaction manager being able to use the one-phase commit optimisation.

[XA_OK]
Normal execution.

[XA_RB*]
The resource manager has dissociated the transaction branch from the thread of control and has marked rollback-only the work performed on behalf of **xid*. The following values may be returned regardless of the setting of *flags*:

[XA_RBROLLBACK]
The resource manager marked the transaction branch rollback-only for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager marked the transaction branch rollback-only for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in dissociating the transaction branch from the thread of control.

[XAER_NOTA]
The specified XID is not known to the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

ax_reg( ), *xa_complete*( ), *xa_open*( ), *xa_start*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_end*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_forget — forget about a heuristically completed transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_forget(XID *xid, int rmid, long flags)
```

DESCRIPTION

A resource manager that heuristically completes work done on behalf of a transaction branch must keep track of that branch along with the decision (that is, heuristically committed, rolled back or mixed) until told otherwise. The transaction manager calls *xa_forget*( ) to permit the resource manager to erase its knowledge of **xid*. Upon successful return [XA_OK], **xid* is no longer valid (from the resource manager's point of view). The transaction manager may call this function from any thread of control.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_forget*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, then this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_forget*( ) has the following return values:

[XA_OK]
: Normal execution.

[XAER_ASYNC]
: TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
: An error occurred in the resource manager and the resource manager has not forgotten the transaction branch.

[XAER_NOTA]
: The specified XID is not known to the resource manager as a heuristically completed XID.

[XAER_INVAL]
: Invalid arguments were specified.

[XAER_PROTO]
: The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
: An error occurred that makes the resource manager unavailable.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit( ), *xa_complete*( ), *xa_open*( ), *xa_recover*( ), *xa_rollback*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_forget*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_open — open a resource manager

SYNOPSIS

```
#include <xa.h>

int xa_open(char *xa_info, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_open*( ) to initialise a resource manager for the calling thread of control and prepare it for use in a distributed transaction processing environment. It applies to resource managers that support the notion of *open* and must be called before any other resource manager (*xa_**( )) calls are made.

The argument *xa_info* points to a null-terminated character string that may contain instance-specific information for the resource manager. The maximum length of this string is 256 bytes (including the null terminator). The argument *xa_info* may point to an empty string if the resource manager does not require instance-specific information to be available. The argument *xa_info* must not be a null pointer.

The argument *rmid*, an integer assigned by the transaction manager, uniquely identifies the called resource manager instance within the thread of control. The transaction manager passes the *rmid* on subsequent calls to XA functions to identify the resource manager. This identifier remains constant until the transaction manager in this thread closes the resource manager.

If the resource manager supports multiple instances, the transaction manager can call *xa_open*( ) more than once for the same resource manager. The transaction manager generates a new *rmid* value for each call, and must use different values for **xa_info* on each call, typically to identify the respective resource domain.

The transaction manager must call this function from the same thread of control that accesses the resource manager. In addition, attempts to open a resource manager instance that is already open have no effect and return success [XA_OK].

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_open*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, then this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_open*( ) has the following return values:

[XA_OK]
: Normal execution.

[XAER_ASYNC]
: TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred when opening the resource.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_close( ), *xa_complete*( ).

WARNINGS

From the resource manager's perspective, the pointer *xa_info* is valid only for the duration of the call to *xa_open*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xa_info* points. Resource managers are encouraged to use private copies of **xa_info* after the function completes.

NAME

xa_prepare — prepare to commit work done on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_prepare(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_prepare*( ) to request a resource manager to prepare for commitment any work performed on behalf of **xid*. The resource manager places any resources it held or modified in such a state that it can make the results permanent when it receives a commit request (that is, when the transaction manager calls *xa_commit*( )). If the transaction branch has already been prepared with *xa_prepare*( ), subsequent calls to *xa_prepare*( ) return [XAER_PROTO]. The transaction manager may call this function from any thread of control. All associations for **xid* must have been ended by using *xa_end*( ) with TMSUCCESS set in *flags*.

Once this function successfully returns, the resource manager must guarantee that the transaction branch may be either committed or rolled back regardless of failures. A resource manager cannot erase its knowledge of a branch until the transaction manager calls either *xa_commit*( ) or *xa_rollback*( ) to complete the branch.

As an optimisation, the resource manager may indicate either that the work performed on behalf of a transaction branch was read-only or that the resource manager was not accessed on behalf of a branch (that is, *xa_prepare*( ) may return [XA_RDONLY]). In either case, the resource manager may release all resources and forget about the branch.

The transaction manager must issue a separate *xa_prepare*( ) for each transaction branch that accessed the resource manager on behalf of the global transaction.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC

This flag indicates that *xa_prepare*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_prepare*( ) has the following return values:

[XA_RDONLY]

The transaction branch was read-only. The branch does not need to be included in the second phase of the commit operation. On return, the resource manager has released all held resources.

[XA_OK]

Normal execution.

[XA_RB*]
The resource manager did not prepare to commit the work done on behalf of the transaction branch. Upon return, the resource manager has rolled back the branch's work and has released all held resources. The following values may be returned:

[XA_RBROLLBACK]
The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
The resource manager encountered an error in preparing to commit the transaction branch's work. The specified XID may or may not have been prepared.

[XAER_NOTA]
The specified XID is not known to the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable. The specified XID may or may not have been prepared.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit( ), *xa_complete*( ), *xa_open*( ), *xa_rollback*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_prepare*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_ready — report that a transaction branch has been prepared

SYNOPSIS

```
#include <xa.h>

int xa_ready(XID *xid, int rmid, long flags)
```

DESCRIPTION

This function is called by a transaction manager to inform a communication resource manager that a transaction branch has been marked *ready*, but the two-phase commit process was disrupted by a failure.

The *xa_ready*( ) function orders the communication resource manager to request the superior transaction manager to report the status of a global transaction. The transaction manager calls this function during recovery processing. The communication resource manager responds with either an order to commit or rollback the subordinate or an indication that the outcome of the prepare phase of the two-phase commit process is not yet known.

The first argument, *xid*, is a pointer to the XID that identifies the transaction branch. The XID must be the same as one given to the transaction manager when the communication resource manager issued an *ax_start*( ).

The function's second argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

Following are the valid settings of *flags*:

TMRECOVER

This flag indicates that this operation is being performed to recover a subordinate branch.

Note: TMNOFLAGS is not valid.

TMASYNC

This flag indicates that *xa_ready*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, then this function fails, returning [XAER_ASYNC].

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the communication resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_ready*( ) has the following return values:

[XA_RBROLLBACK]

The superior transaction manager marked the transaction branch *rollback-only*. Perform a rollback.

[XA_DEFERRED]

A decision has not yet been made. The superior transaction manager completes the two-phase commit process later. This is the normal return value for the client/server style of communication.

[XA_OK]
Normal execution, which indicates that the superior transaction manager has marked the transaction branch *decided* and that the commit phase should be performed.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
The resource manager encountered an error.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_open( ), *ax_add_branch*( ), *ax_commit*( ), *ax_ready*( ), *ax_rollback*( ), *ax_start*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_ready*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_recover — obtain a list of prepared transaction branches from a resource manager

SYNOPSIS

```
#include <xa.h>

int xa_recover(XID *xids, long count, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_recover*( ) during recovery to obtain a list of transaction branches that are currently in a prepared or heuristically completed state. The caller points **xids* to an array into which the resource manager places XIDs for these transactions, and sets *count* to the maximum number of XIDs that fit into that array.

So that all XIDs may be returned irrespective of the size of the array *xids*, one or more *xa_recover*( ) calls may be used within a single recovery scan. The *flags* parameter to *xa_recover*( ) defines when a recovery scan should start or end, or start and end. The start of a recovery scan moves a cursor to the start of a list of prepared and heuristically completed transactions. Throughout the recovery scan the cursor marks the current position in that list. Each call advances the cursor past the set of XIDs it returns.

Two consecutive complete recovery scans return the same list of transaction branches unless the transaction manager calls *xa_commit*( ), *xa_forget*( ), *xa_prepare*( ) or *xa_rollback*( ) for that resource manager, or unless that resource manager heuristically completes some branches, between the two recovery scans.

The transaction manager may call this function from any thread of control, but all calls in a given recovery scan must be made by the same thread.

Upon success, *xa_recover*( ) places zero or more XIDs in the space pointed to by **xids*. The function returns the number of XIDs it has placed there. If this value is less than *count*, there are no more XIDs to recover and the current scan ends. (That is, the transaction manager need not call *xa_recover*( ) again with TMENDRSCAN set in *flags*.) Multiple invocations of *xa_recover*( ) retrieve all the prepared and heuristically completed transaction branches.

It is the transaction manager's responsibility to ignore XIDs that do not belong to it.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMSTARTRSCAN

> This flag indicates that *xa_recover*( ) should start a recovery scan for the thread of control and position the cursor to the start of the list. XIDs are returned from that point. If a recovery scan is already open, the effect is as if the recovery scan were ended and then restarted.

TMENDRSCAN

> This flag indicates that *xa_recover*( ) should end the recovery scan after returning the XIDs. If this flag is used in conjunction with TMSTARTRSCAN, the single *xa_recover*( ) call starts and then ends a scan.

TMNOFLAGS

> This flag must be used when no other flags are set in *flags*. A recovery scan must already be started. XIDs are returned starting at the current cursor position.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_recover*( ) has the following return values:

[≥ 0]
As a return value, *xa_recover*( ) normally returns the total number of XIDs it returned in **xids*.

[XAER_RMERR]
An error occurred in determining the XIDs to return.

[XAER_INVAL]
The pointer *xids* is NULL and *count* is greater than 0, *count* is negative, an invalid *flags* was specified, or the thread of control does not have a recovery scan open and did not specify TMSTARTRSCAN in *flags*.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

SEE ALSO

xa_commit( ), *xa_forget*( ), *xa_open*( ), *xa_rollback*( ).

WARNINGS

If *xids* points to a buffer that cannot hold all of the XIDs requested, then *xa_recover*( ) may overwrite the caller's data space.

NAME

xa_rollback — roll back work done on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_rollback(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_rollback*( ) to roll back the work performed at a resource manager on behalf of the transaction branch. A branch must be capable of being rolled back until it has successfully committed. Any resources held by the resource manager for the branch are released and those modified are restored to their values at the start of the branch. The transaction manager may call this function from any thread of control.

The resource manager can forget a rolled back transaction branch either after it has notified all associated threads of control of the branch's failure (by returning [XAER_NOTA] or [XA_RB*] on a call to *xa_end*( )), or after the transaction manager calls it using *xa_start*( ) with TMRESUME set in *flags*. The transaction manager must ensure that no new threads of control are allowed to access a resource manager with a rolled back (or marked rollback-only) XID.

In addition, *xa_rollback*( ) must guarantee that forward progress can be made in releasing resources and restoring them to their initial values. That is, because this function may be used by the transaction manager to resolve deadlocks (defined in a manner dependent on the transaction manager), *xa_rollback*( ) must not itself be susceptible to indefinite blocking.

If the resource manager already completed the work associated with **xid* heuristically, this function merely reports how the resource manager completed the transaction branch. A resource manager cannot forget about a heuristically completed branch until the transaction manager calls *xa_forget*( ).

The transaction manager must issue a separate *xa_rollback*( ) for each transaction branch that accessed the resource manager on behalf of the global transaction.

If the transaction manager is rolling back a transaction branch while transaction chaining is in effect and the resource manager supports chained transactions, the transaction manager sets the TMCHAINED flag. *xid*[0] is the transaction branch to roll back, and *xid*[1] is the chained transaction branch to assign. If the resource manager returns any [XAER_*] negative return value, or [XA_RETRY_COMMFAIL], the new transaction branch is not assigned.

The transaction manager begins local work on the new transaction later by issuing *xa_start*( ) in the application thread of control for statically registering resource managers. Dynamically registering resource managers need to register using *ax_reg*( ) in the application thread of control when application work in the resource manager begins.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC

This flag indicates that *xa_rollback*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMCHAINED

This flag indicates that processing is in chained transaction mode, and in addition to rolling back the existing transaction branch, a new XID is provided for the next transaction. *xid*[0] is the XID to roll back. *xid*[1] is the XID to assign for the next transaction.

TMRECOVER

This flag indicates that this operation is being performed to recover a subordinate branch.

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_rollback*( ) has the following return values:

[XA_RETRY_COMMFAIL]

The resource manager is not able to roll back the transaction branch at this time. This value is returned when a subordinate cannot be contacted due to communication failure. If previously prepared, all resources held on behalf of **xid* remain in a prepared state until roll back is possible. The transaction manager should reissue *xa_rollback*( ) at a later time. The subordinate may query the transaction manager using *ax_ready*( ) when communications are re-established in order to find out about the roll back condition.

[XA_HEURHAZ]

Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed. A resource manager may return this value only if it has successfully prepared **xid*.

[XA_HEURCOM]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed. A resource manager may return this value only if it has successfully prepared **xid*.

[XA_HEURRB]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back. A resource manager may return this value only if it has successfully prepared **xid*.

[XA_HEURMIX]

Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back. A resource manager may return this value only if it has successfully prepared **xid*.

[XA_OK]

Normal execution.

[XA_RB*]

The resource manager has rolled back the transaction branch's work and has released all held resources. These values are typically returned when the branch was already marked rollback-only. The following values may be returned:

[XA_RBROLLBACK]

The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]

A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in rolling back the transaction branch. The resource manager is free to forget about the branch when returning this error as long as all accessing threads of control have been notified of the branch's state.

[XAER_NOTA]
The specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit( ), *xa_complete*( ), *xa_forget*( ), *xa_open*( ), *xa_prepare*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_rollback*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_start, xa_start_2 — start work on behalf of a transaction branch

SYNOPSIS

```
#include <xa.h>

int xa_start  (XID *xid, int rmid, long flags)

int xa_start_2(XID *xid, int rmid, XACTL *ctl, long flags)
```

DESCRIPTION

A transaction manager calls *xa_start*( ) to inform a resource manager that an application may do work on behalf of a transaction branch. Since many threads of control can participate in a branch and each one may be invoked more than once, *xa_start*( ) must recognise whether or not the XID exists. If another thread is accessing the calling thread's resource manager for the same branch, *xa_start*( ) may block and wait for the active thread to release control of the branch (by means of *xa_end*( )). The transaction manager must call this function from the same thread of control that accesses the resource manager. If the resource manager is doing work outside any global transaction on behalf of the application, *xa_start*( ) returns [XAER_OUTSIDE].

The transaction manager calls *xa_start*( ) only for those resource managers that do not have TMREGISTER set in the *flags* element of their **xa_switch_t** structure. Resource managers with TMREGISTER set must use *ax_reg*( ) to join a transaction branch (see *ax_reg*( ) for details).

The first argument, *xid*, is a pointer to the XID that a resource manager must associate with the calling thread of control. The transaction manager guarantees the XID to be unique for different transaction branches. The transaction manager may generate a new branch qualifier within the XID when it calls *xa_start*( ) for a new thread of control association (that is, when TMRESUME is not set in *flags*; see TMRESUME below). If the transaction manager elects to reuse a branch qualifier previously given to the resource manager for the XID, the transaction manager must inform the resource manager that it is doing so (by setting TMJOIN in *flags*; see TMJOIN below).

If the transaction manager generates a new branch qualifier in the XID, then this thread is loosely-coupled to the other threads in the same branch. That is, the resource manager may treat this thread's work as a separate global transaction with respect to its isolation policies. If the transaction manager reuses a branch qualifier in the XID, then this thread is tightly-coupled to the other threads that share the branch. An RM must guarantee that tightly-coupled threads are treated as a single entity with respect to its isolation policies and that no deadlock occurs within the branch among these tightly-coupled threads.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMJOIN

This flag indicates that the thread of control is joining the work of an existing transaction branch. The resource manager should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the branch.

If a resource manager does not recognise **xid*, the function fails, returning [XAER_NOTA]. Note that this flag cannot be used in conjunction with TMRESUME.

TMRESUME

This flag indicates that a thread of control is resuming work on the specified transaction branch. The resource manager should make available at least the transaction context that is specific to the resource manager, present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

If a resource manager does not recognise **xid*, the function fails, returning [XAER_NOTA]. If the resource manager allows an association to be resumed in a different thread from the one that suspended the work, and the transaction manager expressed its intention to migrate the association (by means of the TMMIGRATE flag on *xa_end*( )), the current thread may be different from the one that suspended the work. Otherwise, the current thread must be the same, or the resource manager returns [XAER_PROTO]. When TMRESUME is set, the transaction manager uses the same XID it used in the *xa_end*( ) call that suspended the association.

If **xid* contains a reused branch qualifier, and the transaction manager has multiple outstanding suspended thread associations for **xid*, the following rules apply:

- The transaction manager can have only one of them outstanding at any time with TMMIGRATE set in *flags*.
- Moreover, the transaction manager cannot resume this association in a thread that currently has a non-migratable suspended association.

These rules prevent ambiguity as to which context is restored.

TMNOWAIT

When this flag is set and a blocking condition exists, *xa_start*( ) returns [XA_RETRY] and the resource manager does not associate the calling thread of control with **xid* (that is, the call has no effect). Note that this flag cannot be used in conjunction with TMASYNC.

TMASYNC

This flag indicates that *xa_start*( ) shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*( ) to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

TMNOFLAGS

This flag must be used when no other flags are set in *flags*.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the resource manager should not validate the unused settings of *flags*.

On input to *xa_start_2*( ), the transaction manager sets flag bits in *ctl→flags* to inform the resource manager of the options that are being passed. The following settings are valid:

XAOPTS_NOFLAGS

No flags or values are set. The transaction manager has passed no information in the control structure to the resource manager. This is equivalent to calling *xa_start*( ).

XAOPTS_TIMEOUT

Setting this flag indicates that the TM has placed in *ctl→timeout* the number of seconds before which the resource manager can timeout and rollback the transaction.

On output, the resource manager may change the flags word. The significance of changing any particular flag is defined under the description of that option.

RETURN VALUE

The function *xa_start*( ) has the following return values:

[XA_RETRY]

TMNOWAIT was set in *flags* and a blocking condition exists.

[XA_OK]
Normal execution.

[XA_RB*]
The resource manager has not associated the transaction branch with the thread of control and has marked **xid* rollback-only. The following values may be returned regardless of the setting of *flags*:

[XA_RBROLLBACK]
The resource manager marked the transaction branch rollback-only for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager marked the transaction branch rollback-only for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in associating the transaction branch with the thread of control.

[XAER_NOTA]
Either TMRESUME or TMJOIN was set in *flags*, and the specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

[XAER_DUPID]
If neither TMRESUME nor TMJOIN was set in *flags* (indicating the initial use of **xid*) and the XID already exists within the resource manager, then the resource manager must return [XAER_DUPID]. The resource manager failed to associate the transaction branch with the thread of control.

[XAER_OUTSIDE]

The resource manager is doing work outside any global transaction on behalf of the application.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

ax_reg( ), *xa_complete*( ), *xa_end*( ), *xa_open*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_start*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

Similarly, the data pointed to by *ctl* is good only for the duration of the call to *xa_start_2*( ).

NAME

xa_wait — give control to a communication resource manager so that it can report status from superiors

SYNOPSIS

```
#include <xa.h>

int xa_wait(int rmid, long flags)
```

DESCRIPTION

This function is called by a transaction manager to inform a communication resource manager that a subordinate AP issued *tx_commit*( ) or *tx_rollback*( ). That transaction-completion sequence executed by the communication resource manager is *ax_prepare*( ) followed by *ax_commit*( ) for successful transactions, *ax_rollback*( ) for transactions rolled back by the superior or the subordinate, or *ax_prepare*( ) followed by *ax_rollback*( ) for transactions that fail in the prepare phase. A call to *ax_start*( ) may be issued after the transaction-completion sequence to start a chained transaction.

This function is also called by a transaction manager (with *flags* set to TMNOFLAGS) to inform a communication resource manager that the AP issued *tx_close*( ) and there are still some two-phase commit messages from superiors to receive before closing all resource managers. The communication resource manager waits for the receipt of a message from a superior. Receipt of a two-phase commit protocol message from a superior causes the communication resource manager to issue *ax_prepare*( ), *ax_commit*( ) or *ax_rollback*( ).

The *xa_wait*( ) function gives control of the thread of control to a communication resource manager so that the communication resource manager can receive a message from a superior that directs the two-phase commit protocol. The transaction manager issues this call in a thread of control to a communication resource manager so that the communication resource manager can receive messages from superiors. After the communication resource manager finishes processing two-phase commit protocol from a superior, the communication resource manager responds to the *xa_wait*( ) call by returning [XA_OK]. If the transaction manager determines that there are more messages to receive, the transaction manager reissues the *xa_wait*( ) call to the communication resource manager.

Because the application program has indicated that it will not process any more messages from superiors by issuing *tx_commit*( ), *tx_rollback*( ) or *tx_close*( ), receipt of any messages that are not part of the two-phase commit protocol must be rejected. If a message that is not part of the two-phase commit protocol is received for a transaction in progress, the communication resource manager must roll the transaction back by issuing *ax_rollback*( ). It must also inform the superior about the rollback. If a message is an attempt to start a new transaction, the communication resource manager must reject it and inform the superior.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The function's last argument, *flags*, may be set to one of the following values:

TMSUCCESS

The portion of work has succeeded. This flag cannot be used in conjunction with TMFAIL.

TMFAIL

The portion of work has failed. The communication resource manager marks the transaction branch as rollback-only and informs the superior, This flag cannot be used in conjunction with TMSUCCESS.

TMNOFLAGS
This flag value is used when other settings of *flags* do not apply.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the communication resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_wait*( ) has the following return values:

[XA_PROMOTED]
The two-phase commit process has been completed, and the application program has been promoted to transaction initiator.

[XA_OK]
Normal execution. This value indicates that a two-phase commit protocol message has been received and processed. The communication resource manager is returning control to the transaction manager to see if any more messages are expected.

[XAER_RMERR]
The resource manager encountered an error.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *ax_commit*( ), *ax_prepare*( ), *ax_rollback*( ).

NAME

xa_wait_recovery — give control to a communication resource manager so that it can report status from subordinates

SYNOPSIS

```
#include <xa.h>

int xa_wait_recovery(int rmid, long flags)
```

DESCRIPTION

This function is called by a transaction manager to inform a communication resource manager that the transaction manager is available to process *ax_ready*( ) and *ax_done*( ) requests from the communication resource manager.

The *xa_wait_recovery*( ) function gives control of the thread of control to a communication resource manager so that the communication resource manager can inquire about the status of global transactions and report completion of subordinate branches. The transaction manager calls this function in a thread of control once. The communication resource manager responds with [XA_OK] during system shutdown.

The function's first argument, *rmid*, is the integer that the communication resource manager received when the transaction manager called *xa_open*( ). It identifies the communication resource manager in the thread of control.

The function's last argument, *flags*, is reserved for future use and is ordinarily set to TMNOFLAGS.

In order to promote migration to future extensions, the transaction manager should not set any unused flags, and the communication resource manager should not validate the unused settings of *flags*.

RETURN VALUE

The function *xa_wait_recovery*( ) has the following return values:

[XA_OK]
: Normal execution, which indicates that the communication resource manager is shutting down.

[XAER_RMERR]
: The resource manager encountered an error.

[XAER_INVAL]
: Invalid arguments were specified.

[XAER_PROTO]
: The function was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_open( ), *ax_add_branch*( ), *ax_ready*( ), *ax_done*( ).

Chapter 6

State Tables

This chapter contains state tables that show legal calling sequences for the XA functions. TMs must sequence their use of the XA functions so that the calling thread of control makes legal transitions through each applicable table in this chapter. That is, any TM must, on behalf of each AP thread of control:

- open and close each RM as shown in Table 6-1 on page 104
- associate itself with, and dissociate itself from, transaction branches as shown in Table 6-2 on page 105 or Table 6-3 on page 106, whichever applies
- advance any transaction branch toward completion through legal transitions as shown in Table 6-4 on page 108
- make legal transitions through Table 6-5 on page 110, whenever the TM calls XA functions in asynchronous mode.

For some examples, see Appendix B on page 125.

CRMs initiate state transitions as shown in Table 6-2 on page 105 and Table 6-4 on page 108. CRMs must sequence their use of the XA functions so that the calling thread of control makes legal transitions through each applicable table in this chapter.

Table 6-5 on page 110 is the only table that addresses the asynchronous mode of XA functions. The other tables also describe functions that can be called asynchronously. In this case, the tables view the XA call that initiates an operation, and the *xa_complete*( ) call that shows that the operation is complete, as a single event.

Interpretation of the Tables

A single call may make transitions in more than one of the state tables. Services that are not pertinent to a given state table are omitted from that table.

All the tables describe the state of a thread of control with respect to a particular RM. That is, the tables indicate the validity of a sequence of XA calls only if the calls all pertain to the same RM. The thread of control could be dealing with other RMs at the same time, which might be in entirely different states. Each state table indicates the valid initial state or states for such a sequence; it is not always the leftmost state (the state with the zero subscript).

Table 6-2 on page 105, Table 6-3 on page 106 and Table 6-4 on page 108 describe the sequence of calls with respect to the progress of a particular XID. Other XIDs within the same RM thread of control may be in different states as they progress from initial creation through completion, except that a thread can have only one active association at a time. Thus, while one XID may be actively associated in a thread of control, the same thread of control may make branch completion calls for other XIDs.

An entry under a particular state in the table asserts that an XA function can be called in that state, and shows the resulting state. A blank entry asserts that it is an error to call the function in that state. The function should return the protocol error [XAER_PROTO] or [TMER_PROTO], unless another error code that gives more specific information also applies.

Note: For the purposes of the state tables in this chapter, the functions *ax_reg_2*( ), *ax_start_2*( ) and *xa_start_2*( ) are equivalent to *ax_reg*( ), *ax_start*( ) and *xa_start*( ) respectively.

Notation

Sometimes a function makes a state transition only when the caller gives it certain input, or only when the function returns certain output (such as a return code). Specific state table entries describe these cases. The tables describe input to the function in parentheses, even though that may not be the exact syntax used; for example, *xa_end*(TMFAIL) describes a call to *xa_end*( ) in which the caller sets the TMFAIL bit in the *flags* argument. The tables denote output from the function, including return status, using an arrow (→) followed by the specific output.

For example, the legend:

xa_end → [XA_RB]

describes the case where a call to *xa_end*( ) returns one of the [XA_RB*] codes.

A general state table entry (one that does not show flags or output values) describes all remaining cases of calls to that function. These general entries assume the function returns success. (The *xa_* functions return the [XA_OK] code; the *ax_* functions return [TM_OK].) Calls that return failure do not make state transitions, except where described by specific state table entries.

The notation *xa_** refers to all applicable *xa_* functions.

6.1 Resource Manager Initialisation

For each thread of control, each RM is either open or closed. The *initial state* is closed (R_0). The *xa_open*( ) and *xa_close*( ) functions move an RM between these states. Redundant uses of these functions are valid, as Table 6-1 shows:

Table 6-1 State Table for Resource Manager Initialisation

XA Routines	Resource Manager States	
	Not Initialised R_0	**Initialised** R_1
xa_open( )	R_1	R_1
xa_close	R_0	R_0

A transition to R_1 enables the use of Table 6-2 on page 105 to Table 6-4 on page 108 inclusive. The state R_0 appears in these tables to illustrate that closing an RM precludes its use in global transactions. At this point, Table 6-1 governs legal sequences.

In Table 6-2 on page 105 to Table 6-4 on page 108 a return of [XAER_RMFAIL] on any function causes a state transition in that thread to state R_0.

6.2 Association of Threads of Control with Transactions

Table 6-2 shows the state of an association between a thread of control and a transaction branch. (See Table 6-3 on page 106 for RMs that dynamically register with a TM.) Valid initial states of association for a thread of control are T_0 and T_2. (The *Association Suspended* state, T_2, includes the case where a thread has suspended an association migratably. After returning to T_0, any thread can re-enter this table in column T_2 to resume or end that other association.)

Table 6-2 makes the following assumptions: the calling thread remains in state R_1; the RM does not have TMREGISTER set in its switch; and the caller passes the same *rmid* and XID as arguments to each applicable function listed below.

Table 6-2 shows the effect of *xa_start*( ) and *xa_end*( ) on the thread of control's association with a single transaction branch. These functions may also change the state of the branch itself. Therefore, Table 6-4 on page 108 further constrains their use. If a thread suspends its association, it can perform work on behalf of other transaction branches before resuming the suspended association.

Table 6-2 State Table for Transaction Branch Association

XA Routines	**Transaction Branch Association States**		
	Not Associated	**Associated**	**Association Suspended**
	T_0	T_1	T_2
Resource Manager Calls			
ax_start( )	T_1		
ax_start(TMRESUME)			T_1
ax_start(TMRESUME) → [TM_RB]			T_0
ax_start(TMJOIN)	T_1		
ax_start(TMJOIN) → [TM_RB]	T_0		
ax_end(TMSUSPEND)		T_2	
ax_end(TMSUSPEND) → [TM_RB]		T_0	
ax_end(TMSUCCESS)		T_0	T_0
ax_end(TMFAIL)		T_0	T_0
*ax_**( ) → [TMER_TMERR]	R_0	R_0	R_0
Transaction Manager Calls			
xa_start( )	T_1		
xa_start(TMRESUME)			T_1
xa_start(TMRESUME) → [XA_RB]			T_0
xa_start(TMJOIN)	T_1		
xa_start(TMJOIN) → [XA_RB]	T_0		
xa_end(TMSUSPEND)		T_2	
xa_end(TMSUSPEND) → [XA_RB]		T_0	
xa_end(TMSUCCESS)		T_0	T_0
xa_end(TMFAIL)		T_0	T_0
xa_open( )	T_0	T_1	T_2
xa_recover( )	T_0	T_1	T_2
xa_close( )	R_0		R_0
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0

6.2.1 Dynamic Registration of Threads

Table 6-3 shows the state of an association between a thread of control and a transaction branch. This table is for RMs that dynamically register with a TM. Valid initial states in Table 6-3 are D_0 and D_2. (The Association Suspended state, D_2, includes the case where a thread has suspended an association migratably. After returning to D_0, any thread can re-enter this table in column D_2 to resume or end that other association.)

The top half of the table shows the legal sequence of calls for an RM thread of control. The bottom half of the table shows the legal sequence of calls for a TM thread of control. The thread of control calling these functions must comply with the applicable half of the table.

Table 6-3 makes the following assumptions:

- The TM calling thread remains in state R_1.
- The RM has TMREGISTER set in its switch.
- The caller passes the same *rmid* and XID as arguments to each applicable function listed below.
- The same RM and XID are used for the dynamic registration functions.

Table 6-3 defines the behaviour of a single transaction branch in a thread. If a thread suspends its association, it can perform work on behalf of other branches before resuming the suspended association.

Table 6-3 State Table for Transaction Branch Association (Dynamic Registration)

XA Routines	**Transaction Branch Association States**			
	Not Registered D_0	**Registered with Valid XID** D_1	**Registration Suspended** D_2	**Registered with NULLXID** D_3
Resource Manager Calls				
ax_reg → valid XID	D_1			
ax_reg → NULLXID	D_3			
ax_reg → [TM_RESUME]			D_1	
ax_unreg				D_0
Transaction Manager Calls				
xa_end(TMSUSPEND)		D_2		
xa_end(TMSUSPEND) → [XA_RB]		D_0		
xa_end(TMSUCCESS)		D_0	D_0	
xa_end(TMFAIL)		D_0	D_0	
xa_open( )	D_0	D_1	D_2	D_3
xa_recover( )	D_0	D_1	D_2	D_3
xa_close( )	R_0		R_0	
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0	R_0

6.3 Transaction States

Table 6-4 on page 108 shows the commitment protocol for a transaction branch. Any state listed in Table 6-4 on page 108 except state S_1 is a valid initial state for a thread of control. The table applies to sequential calls by a thread of control that:

- remains in state R_1
- passes the same *XID* in each *xa_* call that requires an *XID*.

Table 6-4 on page 108 shows the effect of *xa_start*( ) and *xa_end*( ) on the state of a transaction branch. These functions may also change the thread's association with the branch. Therefore, Table 6-2 on page 105 and Table 6-3 on page 106 further constrain their use.

Table 6-4 on page 108 does not apply to uses of *xa_end*(TMSUSPEND), *xa_start*(TMRESUME) or *ax_reg* in which [TM_RESUME] is returned; the uses of these are constrained by Table 6-2 on page 105 and Table 6-3 on page 106 and the following rules:

- *xa_end*(TMSUSPEND | TMMIGRATE) may be used only if no other thread association for this branch was suspended with the TMMIGRATE flag. (This rule ensures that there exists at most one migratable suspended association for a branch.)
- *xa_start*(TMRESUME) may be used, and *ax_reg* may return [TM_RESUME], only on a branch that has at least one suspended association. That suspended association must either have been suspended non-migratably by the acting thread or suspended migratably by any thread. If both conditions are true, the association which was suspended non-migratably by the acting thread is the one resumed.
- *xa_end*(TMSUCCESS) may be used only on a branch that is associated with the current thread or that has at least one suspended association. If the branch is associated with the current thread, it is that association which is ended. Otherwise, a suspended association ends, as though an implicit *xa_start*(TMRESUME) were performed (see above) before the *xa_end*(TMSUCCESS).
- In Table 6-4 on page 108, a branch is considered Idle only if all associations for the branch have been successfully ended by using *xa_end*( ) with TMSUCCESS set in flags. However, a special case exists for rollback. *xa_rollback*( ) may be used for an active branch from any thread except the one currently associated with it, and for a suspended branch from any thread including the one which suspended.

Table 6-4 State Table for Transaction Branches

	XA Routines	**Transaction Branch States**					
		Non-existent Transaction S_0	**Active** S_1	**Idle** S_2	**Prepared** S_3	**Rollback Only** S_4	**Heuristically Completed** S_5
	Resource Manager Calls						
	ax_start( )	S_1		S_1			
†	*ax_start*( ) → [TM_RB]			S_4			
	ax_reg( )	S_1		S_1			
	ax_end( )		S_2				
†	*ax_end*( ) → [TM_RB]		S_4				
	ax_prepare( )			S_3			
†	*ax_prepare*( ) → [TM_RDONLY] or [TM_RB]			S_0			
	ax_prepare( ) → [TMER_RMERR]			S_2			
	ax_commit( ) → [TM_OK] or [TMER_TMERR]			S_0	S_0		
†	*ax_commit*( ) → [TM_RB]			S_0			
†	*ax_commit*( ) → [TM_HEUR]			S_5	S_5		S_5
†	*ax_rollback*( ) → [TM_OK] or [TM_RB] or [TMER_TMERR]			S_0	S_0	S_0	
†	*ax_rollback*( ) → [TM_HEUR]				S_5	S_5	S_5
	ax_add_branch( )		S_1				
	ax_forget_branch( )		S_1	S_2			
	ax_recover( )	S_0	S_1	S_2	S_3	S_4	S_5
	ax_done( )	S_0			S_0		S_5
	ax_ready( )	S_0			S_3	S_4	S_5
	ax_set_branch_info( )		S_1	S_2			
	ax_get_branch_info( )		S_1	S_2	S_3	S_4	S_5
	Transaction Manager Calls						
	xa_start( )	S_1		S_1			
‡	*xa_start*( ) → [XA_RB]			S_4			
	xa_wait( )		S_2	S_2			
	xa_wait_recovery( )	S_0		S_2	S_3	S_4	S_5
	xa_ready( )				S_3		
	xa_done( )	S_0					
	xa_end( )		S_2				
‡	*xa_end*( ) → [XA_RB]		S_4				
	xa_prepare( )			S_3			

XA Routines	Transaction Branch States					
	Non-existent Transaction S_0	**Active** S_1	**Idle** S_2	**Prepared** S_3	**Rollback Only** S_4	**Heuristically Completed** S_5
‡ *xa_prepare*( ) → [XA_RDONLY] or [XA_RB]			S_0			
xa_prepare( ) → [XAER_RMERR]			S_2			
xa_commit( ) → [XA_OK] or [XAER_RMERR]			S_0	S_0		
‡ *xa_commit*( ) → [XA_RB]			S_0			
xa_commit( ) → [XA_RETRY]				S_3		
‡ *xa_commit*( ) → [XA_HEUR]			S_5	S_5		S_5
‡ *xa_rollback*( ) → [XA_OK] or [XA_RB] or [XAER_RMERR]			S_0	S_0	S_0	
‡ *xa_rollback*( ) → [XA_HEUR]				S_5	S_5	S_5
xa_forget( )						S_0
xa_forget( ) → [XAER_RMERR]						S_5
xa_open( )	S_0	S_1	S_2	S_3	S_4	S_5
xa_recover( )	S_0	S_1	S_2	S_3	S_4	S_5
xa_close( )	R_0		R_0	R_0	R_0	R_0
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0	R_0	R_0	R_0

Notes:

† [TM_HEUR] denotes any of [TM_HEURCOM], [TM_HEURRB], [TM_HEURMIX] or [TM_HEURHAZ]. [TM_RB] denotes any return value with a prefix [TM_RB.

‡ [XA_HEUR] denotes any of [XA_HEURCOM], [XA_HEURRB], [XA_HEURMIX] or [XA_HEURHAZ]. [XA_RB] denotes any return value with a prefix [XA_RB.

6.4 Asynchronous Operations

Table 6-5 describes asynchronous operations. The preceding tables do not take into account asynchronous operations.

Table 6-5 illustrates that once a TM thread makes an asynchronous request to an RM on behalf of an XID, the only valid request the TM thread can give the same RM for that XID is the corresponding *xa_complete*( ). For those functions that do not take an XID, the thread must wait for the operation's completion before issuing another request to that same RM. The only functions by which the TM can give additional work to the same RM are *xa_commit*( ), *xa_forget*( ), *xa_prepare*( ) and *xa_rollback*( ).

The valid **initial state** for a thread of control is A_0.

The asynchronous calls (with TMASYNC) achieve a state transition in Table 6-5 when the function returns a valid handle (a positive value). The table entries describing *xa_complete*( ) assume that the caller passes this same, valid handle to *xa_complete*( ).

Table 6-5 State Table for Asynchronous Operations

XA Routines	**Asynchronous Operation States**	
	Initial Call A_0	**Operation Pending** A_1
Any synchronous *xa_** call	A_0	
xa_rollback(TMASYNC)	A_1	
xa_close(TMASYNC)	A_1	
xa_commit(TMASYNC)	A_1	
xa_end(TMASYNC)	A_1	
xa_forget(TMASYNC)	A_1	
xa_open(TMASYNC)	A_1	
xa_prepare(TMASYNC)	A_1	
xa_start(TMASYNC)	A_1	
xa_done(TMASYNC)	A_1	
xa_ready(TMASYNC)	A_1	
xa_complete( )		A_0
xa_complete(TMNOWAIT)		A_0
xa_complete(TMNOWAIT) → [XA_RETRY]		A_1

Chapter 7

Implementation Requirements

This chapter summarises the implications for implementors of this specification. It also identifies features of this specification that implementors of RMs or TMs can regard as optional.

These requirements are designed to facilitate portability — specifically, the ability to move a software component to a different DTP system without modifying the source code. It is anticipated that DTP products will be delivered as object modules and that the administrator will control the mix and operation of components at a particular site by:

- re-linking object modules
- supplying text strings to the software components (or executing a vendor-supplied procedure that generates suitable text strings).

7.1 Application Program Requirements

Any AP in a DTP system must use a TM and delegate to it responsibility to control and coordinate each global transaction. X/Open will specify a TX interface separately.

The AP is not involved in either the commitment protocol or the recovery process. An AP thread can have only one global transaction active at a time.

The AP may ask for work to be done by calling one or more RMs. It uses the RM's native interface exactly as it would if no TM existed, except that it calls the TM to define global transactions (see Section 7.2.1 on page 114).

7.2 Resource Manager Requirements

The X/Open DTP model affects only RMs operating in the DTP environment. The model puts these constraints on the architecture or implementation of the RM.

Interfaces

RMs must provide all the *xa_* functions specified in Chapter 3 for use by TMs, even if a particular function requires no real action by that RM. RMs must also provide a native application program interface (see Section 7.2.1 on page 114). As this is the second version of the XA interface, RMs must also set to 1 the version element of their switches.

An RM in an executable is linked to at most one TM.

Ability to Recognise XIDs

RMs must accept XIDs from TMs. They must associate with the XID all the work they do for an AP. For example, if an RM identifies a thread of control using a process identifier, the RM must map the process identifier to the XID.

An important attribute of the XID is global uniqueness, based on the exact order of the bits in the data portion of the XID for the lengths specified. Therefore, RMs must not alter in any way the bits in the data portion of the XID. For example, if an RM remotely communicates an XID, it must ensure that the data bits of the XID are not altered by the communication process. That is, the data part of the XID should be treated as an OCTET STRING (opaque data) using terminology defined in the ASN.1 standard and the BER standard.

Transaction Identifiers

If the RM generates XIDs for transaction branches, it must use the *gtrid* portion generated by the TM. The RM generates the *bqual* portion of the XID following the same rules for TMs as described in Section 7.3 on page 115.

Calling Protocol

A single instance of an RM must allow multiple threads to logically gain access to it on behalf of the same transaction branch, although it has the option of actually implementing single-threaded access (for example, blocking a thread at the call to *xa_start*( ) or channeling all accesses through a single back-end process). An RM can use whatever it wishes from the AP's environment (for example, process ID, task ID or user ID) to identify the calling thread.

An RM must ensure that each calling thread makes *xa_* calls in a legal sequence, and must return [XAER_PROTO] or other suitable error if the caller violates the state tables (see Chapter 6).

Commitment Protocol

The RM must support the commitment protocol specified in Section 2.3 on page 10. This has the following implications:

- An RM must provide an *xa_prepare*( ) function, and must be able to report whether it can guarantee its ability to commit the transaction branch. If it reports that it can, it must reserve all the resources needed to commit the branch. It must hold those resources until the TM directs it to either commit or roll back the branch. CRMs need not stably record their decision.
- An RM must support the one-phase commit optimisation (see Section 2.3.2 on page 10). That is, it must allow *xa_commit*( ) (specifying TMONEPHASE) even if it has not yet received *xa_prepare*( ) for the transaction branch in question.

Support for Recovery

An RM must track the status of all transaction branches in which it is involved. After responding affirmatively to the TM's *xa_prepare*( ) call, the RM cannot erase its knowledge of the branch, or of the work it has done in support of that branch, until it receives and successfully performs the TM's order to commit or roll back the branch. CRMs need not stably record their decision. If an RM heuristically completes a branch, it cannot forget the branch until explicitly permitted to by the TM. On command, an RM must return a list of all its branches that it has prepared to commit or has heuristically completed.

When an RM recovers from its own failure, it recovers prepared and heuristically completed transaction branches. It forgets all other branches. CRMs rely on information saved by the TM to perform recovery after system failure.

Public Information

An RM product must publish the following information:

- the name of its **xa_switch_t** structure

  This switch gives RM entry points and other information; Section 4.4 on page 29 specifies its structure. Multiple RMs may share the same **xa_switch_t** structure. Each pointer must point to an actual function, even pointers that are theoretically unused on that RM. These functions must return in an orderly way, in case they are mistakenly called. (The RM does not need to publish the names of the individual functions.) RMs that are not CRMs need not supply the *xa_*( )* functions added at the end of the **xa_switch_t** structure that only apply to CRMs. In that way, **xa_switch_t** structures with version 0 are still upwardly compatible.

- the text of the string, within the RM switch, that specifies the name of the RM
- the forms of the information strings that its *xa_open*( ) and *xa_close*( ) functions accept, and the way that different *xa_open*( ) strings specify different resource domains for different RM instances.
- the names of libraries or object files, in the correct sequence, that the administrator must use when linking APs with the RM
- any semantics in the native interface that affect global transaction processing; for example, specifying isolation level, transaction completion, or effects that differ from non-DTP operation
- the operating system entity that is considered to be a thread of control, and any restrictions on use of multiple-threading facilities of the operating system.

X/Open will specify how an RM provider can guarantee that its names do not conflict with the name of any other RM product or version produced by that or another organisation.

Implementor Options

Each RM has the option of implementing these features:

- **Open/close informational strings**
  Any RM may accept a null string in place of the informational string argument on calls to its *xa_open*( ) and *xa_close*( ) functions. (If it does so, it must publish this fact.)
- **Protocol optimisations**
  The **read-only** optimisation discussed in Section 2.3.2 on page 10 is optional.

- **Association migration**
  An RM can require a TM to resume a suspended association in a thread of control other than the one where the suspension occurred.

- **Branch identification**
  An RM can use the *bqual* component of the XID structure to let different branches of the same global transaction prepare to commit at different times, and to avoid deadlock (see Section 4.2 on page 28).

- **Dynamic registration**
  This feature, as described in Section 3.3.2 on page 19, is optional.

- **Asynchrony**
  Support for the asynchronous mode discussed in Section 3.6 on page 24 is optional. If the TMUSEASYNC flag is set in the RM's switch, then the RM must support the asynchronous mode. It may still complete some requests synchronously, either when the TM makes the original asynchronous call or when the TM calls *xa_complete*( ). If the TMUSEASYNC flag is not set in the RM's switch and the TM makes an asynchronous call, the RM must return [XAER_ASYNC].

- **Heuristics**
  As described in the *xa_commit*( ) and *xa_rollback*( ) reference manual pages, an RM can report that it has heuristically completed the transaction branch. This feature is optional.

- **Receiving Timeouts**
  Support for the *xa_start_2*( ) function is optional.

7.2.1 The Application Program (Native) Interface

RMs must provide a well-defined native interface that APs can use to request work. To maximise portability, all AP-RM interfaces should adhere to any applicable X / Open publication. For example, a relational DBMS RM should use the SQL defined in the **X/Open SQL** specification.

In the DTP environment, RMs must rely on the TM to manage global transactions. Some RMs, such as some Indexed Sequential Access Method (ISAM) file managers, have no concept of transactions. So this is a new requirement, but it does not change the native interface. In other RMs, such as SQL RDBMSs, the native interface defines transactions. An AP must not use these services in a DTP context, since TMs have no knowledge of an RM's transaction. For example, the application program interface for an SQL RM in the DTP environment must not let use of these statements affect the global transaction:

```
EXEC SQL COMMIT WORK
EXEC SQL ROLLBACK WORK
```

In addition, any service in the native AP-RM interface that affects its own commitment, or any non-standard service that has an effect on transactions, must not be used within a global transaction.

7.3 Transaction Manager Requirements

Service Interfaces

TMs must use the *xa_* functions the RM provides (see Chapter 3 on page 13) to coordinate the work of all the local RMs that the AP uses. TMs must call *xa_open*( ) and *xa_close*( ) on any local RM associated with the TM. Each TM must ensure that each of its *xa_**( ) calls addresses the correct RM.

TMs must be written to handle consistently any information or status that an RM can legally return. A TM must assume that it may be linked with RMs that use any RM options that this specification allows, including multiple RMs in a single AP object program, dynamic registration of RMs, RMs that make heuristic decisions, and RMs that use the read-only protocol optimisation.

Transaction Identifiers

A TM must generate XIDs conforming to the structure described in Section 4.2 on page 28. They must be globally unique and must adequately describe the transaction branch. To guarantee global uniqueness, the TM should use an ISO OBJECT IDENTIFIER (see the ASN.1 standard and the BER standard) that the TM knows is unique within the **gtrid** component of the XID. The TM should also use an ISO OBJECT IDENTIFIER within the **bqual** field, but the same OBJECT IDENTIFIER can be used for all XIDs that a given TM generates. The *bqual* fields identify the TM that generated them, and identify the transaction branch with which they are associated.

Failure to use the ISO OBJECT IDENTIFIER format for XIDs could cause interoperability problems when multiple TMs are either involved in the same global transaction or affect shared resources.

Public Information

A TM product must publish the following information:

- linking directions for producing an application object program — these directions must describe how to link RM and TM libraries, and how to incorporate the switch from each RM
- instructions for generating or locating appropriate informational strings that the TM uses when it calls *xa_open*( ) or *xa_close*( ) for each RM
- the operating system entity that is considered to be a thread of control, and any restrictions on use of multiple-threading facilities of the operating system.

Implementor Options

The one-phase commit protocol optimisation (see Section 2.3.2 on page 10) and the asynchronous and non-blocking modes for calling RMs (see Section 3.6 on page 24) are optional.

Appendix A

Complete Text of <xa.h>

This appendix specifies the complete text of an **<xa.h>** file in both ISO C (see the ISO C standard) and Common Usage C.

```
/*
 * Start of xa.h header
 *
 * Define a symbol to prevent multiple inclusions of this header file
 */
#ifndef XA_H
#define XA_H
/*
 * Transaction branch identification: XID and NULLXID:
 */
#define XIDDATASIZE 128   /* size in bytes */
#define MAXGTRIDSIZE 64   /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE 64   /* maximum size in bytes of bqual */
struct xid_t {
  long formatID;          /* format identifier */
  long gtrid_length;      /* value not to exceed 64 */
  long bqual_length;      /* value not to exceed 64 */
  char data[XIDDATASIZE];
  };
typedef struct xid_t XID;
/*
 * A value of -1 in formatID means the XID is null.
 */
/*
 * Declarations of functions by which RMs call TMs:
 */
#ifdef __STDC__
extern int ax_reg(int, XID *, long);
extern int ax_unreg(int, long);
#else /* ifndef __STDC__ */
extern int ax_reg();
extern int ax_unreg();
#endif /* ifndef __STDC__ */
/*
 * XA Options
 */
typedef long TRANSACTION_TIMEOUT;    /* type of transaction timeouts */
/*
 * Structure for optional XA information
 */
struct xactl_t {
   long flags;                       /* valid element flags */
   TRANSACTION_TIMEOUT timeout;      /* timeout value */
};
typedef struct xactl_t XACTL;
#define XAOPTS_NOFLAGS 0x00000000L   /* no optional values */
```

```
#define XAOPTS_TIMEOUT 0x00000001L   /* timeout value present */
};
/*
 * XA Switch Data Structure
 */
#define RMNAMESZ 32       /* length of resource manager name */
struct xa_switch_t {
  char name[RMNAMESZ];    /* name of resource manager */
  long flags;             /* resource manager specific options */
  long version;           /* must be 1 */
#ifdef __STDC__
  int (*xa_open_entry)(char *, int, long);
                                /* xa_open function pointer */
  int (*xa_close_entry)(char *, int, long);
                                /* xa_close function pointer*/
  int (*xa_start_entry)(XID *, int, long);
                                /* xa_start function pointer */
  int (*xa_end_entry)(XID *, int, long);
                                /* xa_end function pointer */
  int (*xa_rollback_entry)(XID *, int, long);
                                /* xa_rollback function pointer */
  int (*xa_prepare_entry)(XID *, int, long);
                                /* xa_prepare function pointer */
  int (*xa_commit_entry)(XID *, int, long);
                                /* xa_commit function pointer */
  int (*xa_recover_entry)(XID *, long, int, long);
                                /* xa_recover function pointer*/
  int (*xa_forget_entry)(XID *, int, long);
                                /* xa_forget function pointer */
  int (*xa_complete_entry)(int *, int *, int, long);
                                /* xa_complete function pointer */
  int (*xa_ready_entry)(XID *, int, long);
                                /* xa_ready function pointer */
  int (*xa_done_entry)(XID *, int, long);
                                /* xa_done function pointer */
  int (*xa_wait_recovery_entry)(int, long);
                                /* xa_wait_recovery function pointer */
  int (*xa_wait_entry)(int, long);
                                /* xa_wait function pointer */
  int (*xa_start_2_entry)(XID *, int, XACTL *, long);
                                /* xa_start_2 function pointer */
#else /* ifndef __STDC__ */
  int (*xa_open_entry)();       /* xa_open function pointer */
  int (*xa_close_entry)();      /* xa_close function pointer */
  int (*xa_start_entry)();      /* xa_start function pointer */
  int (*xa_end_entry)();        /* xa_end function pointer */
  int (*xa_rollback_entry)();   /* xa_rollback function pointer */
  int (*xa_prepare_entry)();    /* xa_prepare function pointer */
  int (*xa_commit_entry)();     /* xa_commit function pointer */
  int (*xa_recover_entry)();    /* xa_recover function pointer */
  int (*xa_forget_entry)();     /* xa_forget function pointer */
  int (*xa_complete_entry)();   /* xa_complete function pointer */
  int (*xa_ready_entry)();      /* xa_ready function pointer */
  int (*xa_done_entry)();       /* xa_done function pointer */
  int (*xa_wait_recovery_entry)();
                                /* xa_wait_recovery function pointer */
  int (*xa_wait_entry)();       /* xa_wait function pointer */
  int (*xa_start_2_entry)();    /* xa_start_2 function pointer */
#endif /* ifndef __STDC__ */
  struct ax_switch_t **xa_tmswitch;
                                /* Location of TM switch pointer */
  };
/*
 * AX Switch Data Structure
 */
struct ax_switch_t {
  long flags;                 /* transaction manager options */
  long version;               /* must be 0 */
#ifdef _STDC_
  int (*ax_reg_entry)(int, XID *, long);
                              /* ax_reg function pointer */
  int (*ax_unreg_entry)(int, long);
                              /* ax_unreg function pointer */
  int (*ax_start_entry)(int, XID *, long);
                              /* ax_start function pointer */
  int (*ax_end_entry)(int, XID *, long);
                              /* ax_end function pointer */
  int (*ax_rollback_entry)(int, XID *, long);
                              /* ax_rollback function pointer */
  int (*ax_prepare_entry)(int, XID *, long);
                              /* ax_prepare function pointer */
  int (*ax_commit_entry)(int, XID *, long);
                              /* ax_commit function pointer */
  int (*ax_recover_entry)(int, XID *, long, long);
                              /* ax_recover function pointer */
  int (*ax_add_branch_entry)(int, XID *, long);
                              /* ax_add_branch function pointer */
  int (*ax_forget_branch_entry)(int, XID *, long);
                              /* ax_forget_branch function pointer */
  int (*ax_set_branch_info_entry)(int, XID *, char *, long, long);
                              /* ax_set_branch_info function pointer */
  int (*ax_get_branch_info_entry)(int, XID *, char *, long *, long);
                              /* ax_get_branch_info function pointer */
  int (*ax_ready_entry)(int, XID *, long);
                              /* ax_ready function pointer */
  int (*ax_done_entry)(int, XID *, long);
                              /* ax_done function pointer */
  int (*ax_reg_2_entry)(int, XID *, XACTL *, long);
                              /* ax_reg_2_entry function pointer */
  int (*ax_start_2_entry)(int, XID *, XACTL *, long);
                              /* ax_start_2_entry function pointer */
#else /* #ifndef _STDC_ */
  int (*ax_reg_entry)();      /* ax_reg function pointer */
  int (*ax_unreg_entry)();    /* ax_unreg function pointer */
  int (*ax_start_entry)();    /* ax_start function pointer */
  int (*ax_end_entry)();      /* ax_end function pointer */
  int (*ax_rollback_entry)(); /* ax_rollback function pointer */
  int (*ax_prepare_entry)();  /* ax_prepare function pointer */
  int (*ax_commit_entry)();   /* ax_commit function pointer */
  int (*ax_recover_entry)();  /* ax_recover function pointer */
  int (*ax_add_branch_entry)();
                              /* ax_add_branch function pointer */
  int (*ax_forget_branch_entry)();
                              /* ax_forget_branch function pointer */
  int (*ax_set_branch_info_entry)();
                              /* ax_set_branch_info function pointer */
  int (*ax_get_branch_info_entry)();
                              /* ax_get_branch_info function pointer */
  int (*ax_ready_entry)();    /* ax_ready function pointer */
  int (*ax_done_entry)();     /* ax_done function pointer */
  int (*ax_reg_2_entry)();    /* ax_reg_2_entry function pointer */
  int (*ax_start_2_entry)();  /* ax_start_2_entry function pointer */
#endif /* #ifndef _STDC_ */
};
/*
 * Flag definitions for the RM switch
 */
#define TMNOFLAGS       0x00000000L   /* no resource manager features
                                      selected */
#define TMREGISTER      0x00000001L   /* resource manager dynamically
                                      registers */
#define TMNOMIGRATE     0x00000002L   /* resource manager does not support
                                      association migration */
#define TMUSEASYNC      0x00000004L   /* resource manager supports
                                      asynchronous operations */
#define TMUSECHAIN      0x00000008L   /* resource manager
                                      supports transaction chaining */
#define TMUSEOPTS       0x00000010L   /* resource manager supports
                                      xa_start_2() */
#define TMUSE2PHASE     0x00000020L   /* resource manager might force
                                      upgrading 1-phase commit to
                                      2-phase commit */
#define TMSWITCHOK      0x00000040L   /* resource manager has provided
                                      location for address of
                                      transaction manager switch */
#define TMNOROLLALLOWED 0x00000080L   /* tx_rollback() is not permitted
                                      in subordinates */
#define TMNOCOMALLOWED  0x00000100L   /* tx_commit() is not permitted
                                      in subordinates */
#define TMUSETHREADS    0x00000200L   /* resource manager can use threads
                                      as thread of control */
```

```
/*
 * Flag definitions for the TM switch
 */
#define TMSUPPORTSTHREADS 0x00000001L /* transaction manager is prepared to
                                      use threads as thread of control */
#define TMSUBORDINATE     0x00000002L /* The subordinate set of ax_()
                                      functions can be called */
/*
 * Flag definitions for xa_ and ax_ functions
 */
/* use TMNOFLAGS, defined above, when not specifying other flags */
#define TMASYNC 0x80000000L     /* perform function asynchronously */
#define TMONEPHASE 0x40000000L  /* caller is using one-phase commit
                                optimisation */
#define TMFAIL 0x20000000L      /* dissociates caller and marks
                                transaction branch rollback-only */
#define TMNOWAIT 0x10000000L    /* return if blocking condition
                                exists */
#define TMRESUME 0x08000000L    /* caller is resuming association
                                with suspended transaction branch */
#define TMSUCCESS 0x04000000L   /* dissociate caller from transaction
                                branch*/
#define TMSUSPEND 0x02000000L   /* caller is suspending, not ending,
                                association */
#define TMSTARTRSCAN 0x01000000L
                                /* start a recovery scan */
#define TMENDRSCAN 0x00800000L  /* end a recovery scan */
#define TMMULTIPLE 0x00400000L  /* wait for any asynchronous
                                operation */
#define TMJOIN 0x00200000L      /* caller is joining existing
                                transaction branch */
#define TMMIGRATE 0x00100000L   /* caller intends to perform
                                migration */
#define TMRECOVER 0x00080000L   /* call in recovery mode */
#define TMCHAINED 0x00040000L   /* call is in transaction chaining
                                mode */
#define TMDEFERRED 0x00020000L  /* start is pending acceptance by
                                the application program */
/*
 * Maximum values for ax_* functions
 */
#define TMMAXBLOBLEN 1024       /* maximum blob_len for
                                ax_set_branch_info() */
#define TMMAXBLOBTOT 8192       /* maximum total blob data created using
                                ax_set_branch_info() for all branches
                                created at this node for a given
                                transaction */
```

```
/*
 * ax_() return codes (transaction manager reports to resource manager)
 */
#define TM_RBBASE 100                 /* the inclusive lower bound of
                                      the rollback codes */
#define TM_RBROLLBACK TM_RBBASE       /* the rollback was caused by an
                                      unspecified reason */
#define TM_RBCOMMFAIL TM_RBBASE+1
                                      /* the rollback was caused by a
                                      communication failure */
#define TM_RBDEADLOCK TM_RBBASE+2     /* a deadlock was detected */
#define TM_RBINTEGRITY TM_RBBASE+3    /* a condition that violates the
                                      integrity of the resources was
                                      detected */
#define TM_RBOTHER TM_RBBASE+4        /* the resource manager rolled
                                      back the transaction branch for
                                      a reason not on this list */
#define TM_RBPROTO TM_RBBASE+5        /* a protocol error occurred in
                                      in the resource manager */
#define TM_RBTIMEOUT TM_RBBASE+6      /* a transaction branch took
                                      too long */
#define TM_RBTRANSIENT TM_RBBASE+7    /* may retry the transaction
                                      branch */
#define TM_RBEND TM_RBTRANSIENT /* the inclusive upper bound of the
                                rollback codes */
#define TM_DEFERRED 11          /* the commit decision has not
                                been made */
#define TM_RETRY_COMMFAIL 10    /* ax_commit could not be completed
                                due to communication failure */
#define TM_NOMIGRATE 9          /* resumption must occur where
                                suspension occurred */
#define TM_HEURHAZ 8            /* the transaction branch may have
                                been heuristically completed */
#define TM_HEURCOM 7            /* the transaction branch has been
                                heuristically committed */
#define TM_HEURRB 6             /* the transaction branch has been
                                heuristically rolled back */
#define TM_HEURMIX 5            /* the transaction branch has been
                                heuristically committed and rolled
                                back */
#define TM_RDONLY 3             /* the transaction branch was read-only
                                and has been committed */
#define TM_JOIN 2               /* caller is joining existing
                                transaction branch */
#define TM_RESUME 1             /* caller is resuming association
                                with suspended transaction branch */
#define TM_OK 0                 /* normal execution */
#define TMER_TMERR -1           /* an error occurred in the
                                transaction manager */
#define TMER_INVAL -2           /* invalid arguments were given */
#define TMER_PROTO -3           /* function invoked in an improper
                                context */
```

```
#define TMER_NOTA -4            /* the XID is not valid */
#define TMER_DUPID -8           /* the XID already exists */
/*
 * xa_() return codes (resource manager reports to transaction manager)
 */
#define XA_RBBASE 100                 /* the inclusive lower bound of
                                      the rollback codes */
#define XA_RBROLLBACK XA_RBBASE       /* the rollback was caused by
                                      an unspecified reason */
#define XA_RBCOMMFAIL XA_RBBASE+1     /* the rollback was caused by a
                                      communication failure */
#define XA_RBDEADLOCK XA_RBBASE+2     /* a deadlock was detected */
#define XA_RBINTEGRITY XA_RBBASE+3    /* a condition that violates
                                      the integrity of the resources
                                      was detected */
#define XA_RBOTHER XA_RBBASE+4        /* the resource manager rolled
                                      back the transaction branch for
                                      a reason not on this list */
#define XA_RBPROTO XA_RBBASE+5        /* a protocol error occurred
                                      in the resource manager */
#define XA_RBTIMEOUT XA_RBBASE+6      /* a transaction branch took
                                      too long */
#define XA_RBTRANSIENT XA_RBBASE+7    /* may retry the transaction
                                      branch */
#define XA_RBEND XA_RBTRANSIENT       /* the inclusive upper bound
                                      of the rollback codes */
#define XA_TWOPHASE 13          /* Use two-phase commit */
#define XA_PROMOTED 12          /* AP promoted to initiator */
#define XA_DEFERRED 11          /* the commit decision has not been
                                made */
#define XA_RETRY_COMMFAIL 10    /* xa_commit could not be completed
                                due to communication failure */
#define XA_NOMIGRATE 9          /* resumption must occur where
                                suspension occurred */
#define XA_HEURHAZ 8            /* the transaction branch may have
                                been heuristically completed */
#define XA_HEURCOM 7            /* the transaction branch has been
                                heuristically committed */
#define XA_HEURRB 6             /* the transaction branch has been
                                heuristically rolled back */
#define XA_HEURMIX 5            /* the transaction branch has been
                                heuristically committed and rolled
                                back */
#define XA_RETRY 4              /* function returned with no effect
                                and may be re-issued */
#define XA_RDONLY 3             /* the transaction branch was read-only
                                and has been committed */
```

```
#define XA_OK 0                 /* normal execution */
#define XAER_ASYNC -2           /* asynchronous operation already
                                outstanding */
#define XAER_RMERR -3           /* a resource manager error occurred in
                                the transaction branch */
#define XAER_NOTA -4            /* the XID is not valid */
#define XAER_INVAL -5           /* invalid arguments were given */
#define XAER_PROTO -6           /* function invoked in an improper
                                context */
#define XAER_RMFAIL -7          /* resource manager unavailable */
#define XAER_DUPID -8           /* the XID already exists */
#define XAER_OUTSIDE -9         /* resource manager doing work outside
                                global transaction */

#endif /* ifndef XA_H */
/*
 * End of xa.h header
 */
```

Appendix B

Scenarios

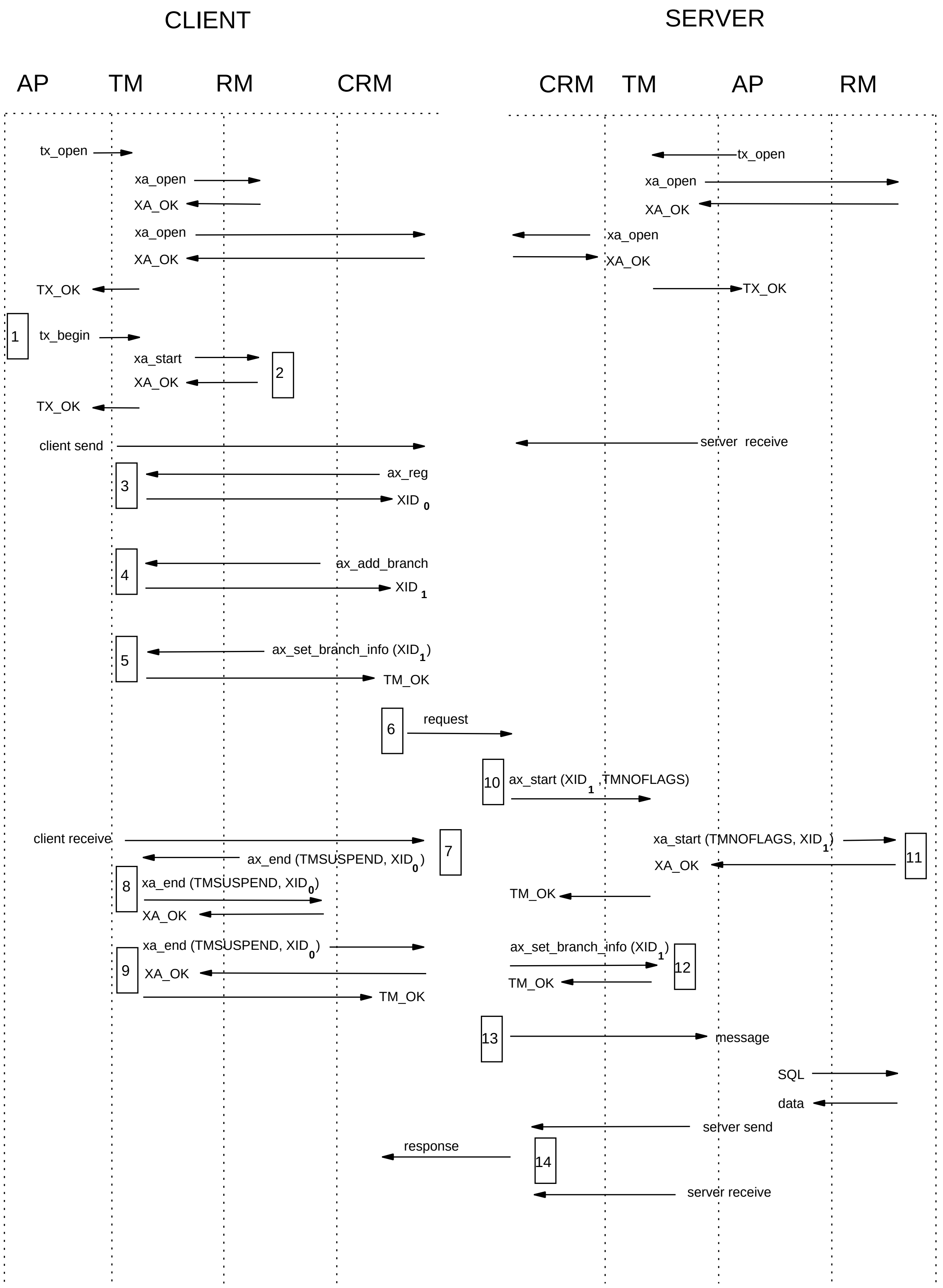
This appendix contains examples of the usage of XA+.

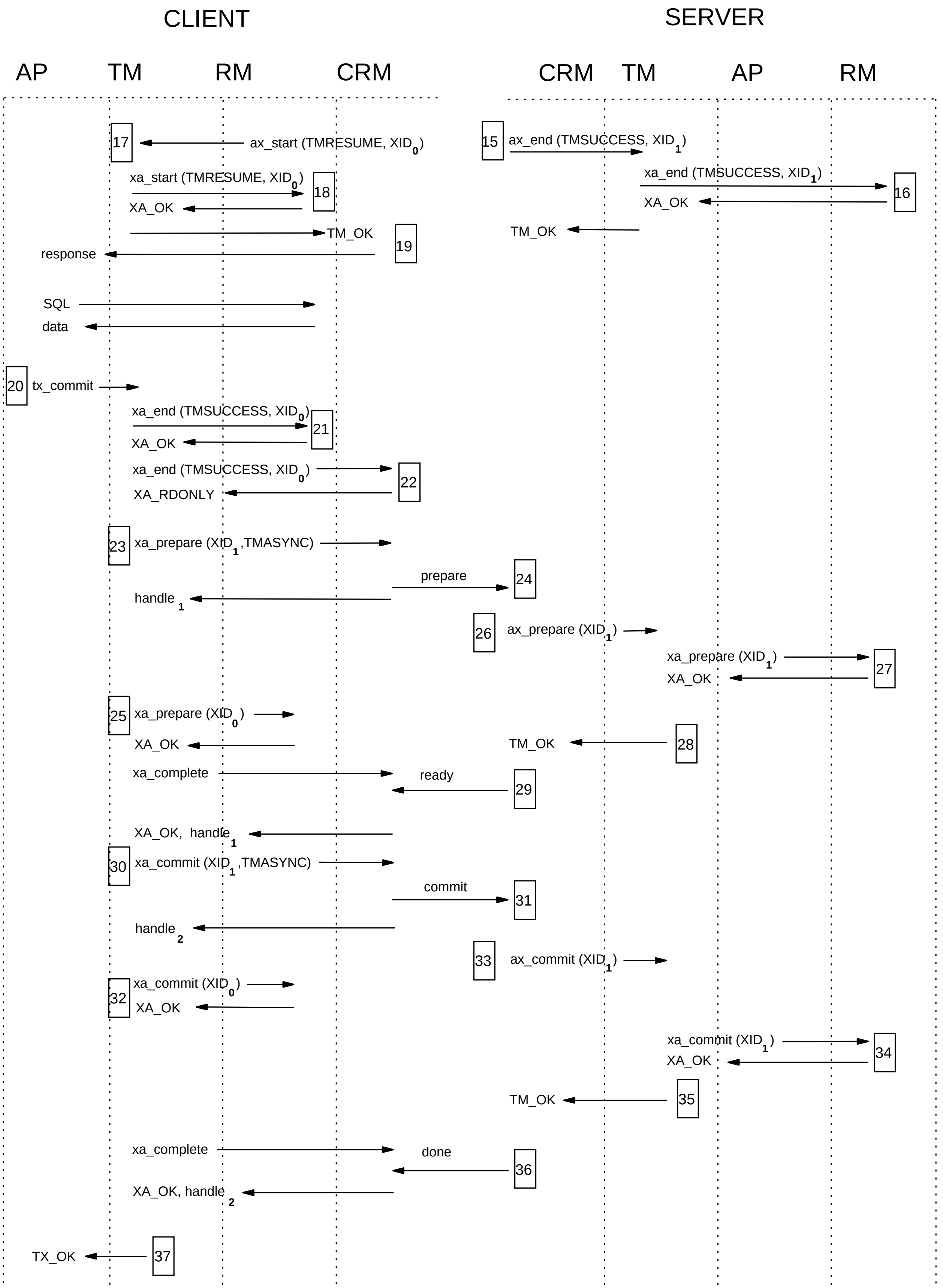
B.1 Single-step Client/Server Interaction

This scenario shows a client beginning a global transaction, sending a message to a remote server, receiving the response, and committing the global transaction.

The following notes describe the flows in the scenario opposite:

1. The client AP begins a transaction.
2. The RM receives *xa_start*( ) because it is statically registered. The CRM does not because it is a dynamically registering RM.
3. The CRM registers with the TM in response to the call from the AP.
4. The CRM requests the TM to add a transaction branch and add a unique branch qualifier to the XID the TM returns. The CRM uses this XID for the message to be sent to the server.
5. The CRM requests the TM to save necessary information to keep track of the branch. This might include the name of the remote location and any other information that the CRM needs, especially during recovery.
6. The request is sent.
7. The CRM in the client asks the TM to suspend the RMs in the thread of control because of the blocking receive call from the AP.
8. The TM issues *xa_end*(TMSUSPEND) to the local RM.
9. The TM issues *xa_end*(TMSUSPEND) to the CRM because it was registered.
10. The CRM informs the TM in the server that a transaction branch is about to begin.
11. The TM passes the transaction branch's XID to the statically registering local RM. The TM does not issue *xa_start*( ) to the CRM because it has selected dynamic registration.
12. The CRM in the server requests the TM to save necessary information to keep track of the superior. This might include routing information and any other information that the CRM needs, especially during recovery.
13. The message is delivered to the service.
14. The service sends a reply, and the CRM transmits the reply to the client.


CLIENT
SERVER
AP
TM
RM
CRM
CRM
TM
AP
RM
tx_open
xa_open
XA_OK
xa_open
XA_OK
TX_OK
1
tx_begin
xa_start
XA_OK
2
TX_OK
client send
3
ax_reg
XID 0
4
ax_add_branch
XID 1
5
ax_set_branch_info (XID1)
TM_OK
6
request
10
ax_start (XID 1 ,TMNOFLAGS)
client receive
7
ax_end (TMSUSPEND, XID0)
8
xa_end (TMSUSPEND, XID0)
XA_OK
9
xa_end (TMSUSPEND, XID0)
XA_OK
TM_OK
tx_open
xa_open
XA_OK
xa_open
XA_OK
TX_OK
server receive
xa_start (TMNOFLAGS, XID1)
XA_OK
11
TM_OK
ax_set_branch_info (XID1)
TM_OK
12
13
message
SQL
data
server send
response
14
server receive


CLIENT
SERVER
AP
TM
RM
CRM
CRM
TM
AP
RM
17
ax_start (TMRESUME, XID$_0$)
15
ax_end (TMSUCCESS, XID$_1$)
xa_start (TMRESUME, XID$_0$)
18
XA_OK
xa_end (TMSUCCESS, XID$_1$)
16
XA_OK
TM_OK
19
TM_OK
response
SQL
data
20
tx_commit
xa_end (TMSUCCESS, XID$_0$)
21
XA_OK
xa_end (TMSUCCESS, XID$_0$)
22
XA_RDONLY
23
xa_prepare (XID$_1$,TMASYNC)
prepare
24
handle $_1$
26
ax_prepare (XID$_1$)
xa_prepare (XID$_1$)
27
XA_OK
25
xa_prepare (XID$_0$)
XA_OK
TM_OK
28
xa_complete
ready
29
XA_OK, handle$_1$
30
xa_commit (XID$_1$,TMASYNC)
commit
31
handle $_2$
33
ax_commit (XID$_1$)
32
xa_commit (XID$_0$)
XA_OK
xa_commit (XID$_1$)
34
XA_OK
TM_OK
35
xa_complete
done
36
XA_OK, handle$_2$
TX_OK
37

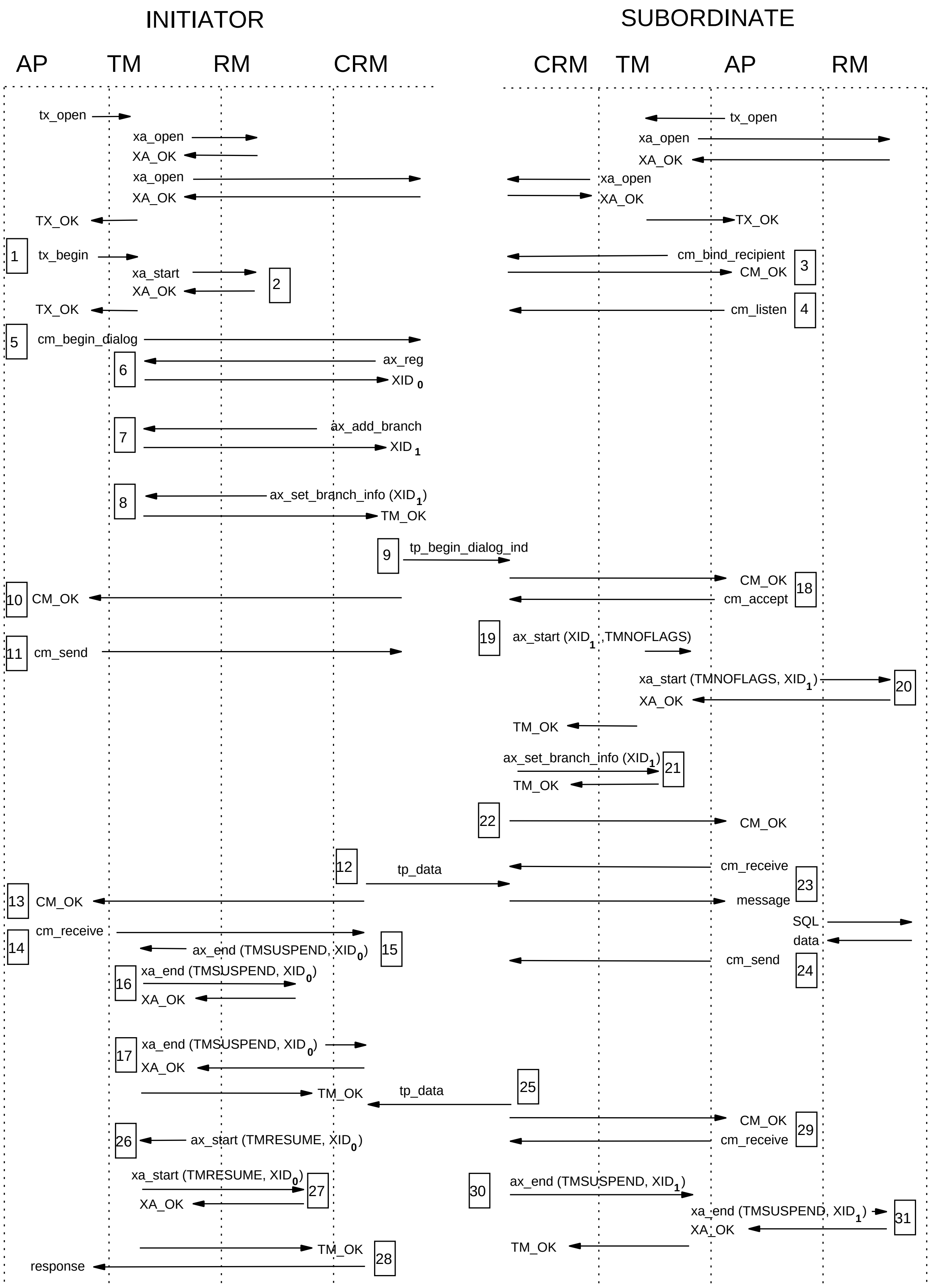
15. The service blocks on a receive. The CRM terminates the association of the thread of control with the transaction branch.
16. The TM terminates the association with the local RM.
17. The CRM in the client calls the TM to resume the association with the thread of control.
18. The TM resumes the association with the local RM. The TM does not resume the association with the CRM because it has selected dynamic registration.
19. The response is returned to the client AP.
20. The client AP calls the TM to commit the transaction.
21. The TM ends the association with the local RM.
22. The TM ends the suspended association with the CRM. The CRM responds read-only so that it will not be included further for the root XID.
23. The TM in the client asks the CRM to prepare the transaction branch in the server.
24. The CRM sends the prepare message to the subordinate (server).
25. The TM prepares the local RM.
26. The CRM propagates the prepare to the remote TM.
27. The TM prepares the transaction branch in the local RM in the server. The CRM is not prepared because it never registered for the transaction branch.
28. The TM in the server logs the ready decision.
29. The ready response is sent to the superior.
30. The coordinator TM logs the ready decision and asks the CRM to commit the transaction branch in the server.
31. The commit message is sent by the CRM to the subordinate (server).
32. The TM commits the local RM.
33. The CRM propagates the commit decision to the TM.
34. The TM commits the transaction branch in the local RM in the server.
35. The TM logs the commit decision in the server. This is a forget transaction branch decision.
36. The commit response is sent to the superior.
37. The coordinator TM logs the commit decision. This is a forget transaction decision. The TM responds to the AP informing it that *tx_commit*( ) was successful.

B.2 Peer-to-Peer Transaction Mandatory Propagation

This scenario shows peer-to-peer communications between a transaction initiator and a subordinate using a transaction mandatory dialogue. The initiator begins the global transaction.

The following notes describe the flows in this scenario:

1. The superior AP begins a transaction.


INITIATOR
SUBORDINATE
AP
TM
RM
CRM
CRM
TM
AP
RM
tx_open
xa_open
XA_OK
xa_open
XA_OK
TX_OK
tx_open
xa_open
XA_OK
xa_open
XA_OK
TX_OK
1
tx_begin
xa_start
XA_OK
2
TX_OK
cm_bind_recipient
CM_OK
3
cm_listen
4
5
cm_begin_dialog
ax_reg
6
XID$_0$
7
ax_add_branch
XID$_1$
8
ax_set_branch_info (XID$_1$)
TM_OK
9
tp_begin_dialog_ind
CM_OK
cm_accept
18
10
CM_OK
19
ax_start (XID$_1$,TMNOFLAGS)
11
cm_send
xa_start (TMNOFLAGS, XID$_1$)
XA_OK
20
TM_OK
ax_set_branch_info (XID$_1$)
TM_OK
21
22
CM_OK
12
tp_data
cm_receive
23
13
CM_OK
message
SQL
data
14
cm_receive
ax_end (TMSUSPEND, XID$_0$)
15
cm_send
24
16
xa_end (TMSUSPEND, XID$_0$)
XA_OK
17
xa_end (TMSUSPEND, XID$_0$)
XA_OK
TM_OK
tp_data
25
CM_OK
cm_receive
29
26
ax_start (TMRESUME, XID$_0$)
xa_start (TMRESUME, XID$_0$)
XA_OK
27
30
ax_end (TMSUSPEND, XID$_1$)
xa_end (TMSUSPEND, XID$_1$)
XA_OK
31
TM_OK
28
response
TM_OK

2. The RM receives *xa_start*( ) because it is statically registered. The CRM does not because it is a dynamically registering RM.
3. The subordinate AP registers with the CRM.
4. The subordinate AP listens for a connection with a superior.
5. The superior AP starts a dialogue with a subordinate.
6. The CRM registers with the TM in response to the call from the superior AP.
7. The CRM requests the TM to add a transaction branch and add a unique branch qualifier to the XID the TM returns. The CRM uses this XID for the message to be sent to the subordinate.
8. The CRM requests the TM to save necessary information to keep track of the branch. This might include the name of the remote location and any other information that the CRM needs, especially during recovery.
9. The begin dialog indication is (logically) sent.
10. The CRM returns success to the superior AP.
11. The superior AP calls the CRM to send a message to the subordinate.
12. The CRM sends the message.
13. The CRM returns success to the superior AP.
14. The superior AP issues a receive that blocks.
15. The CRM in the superior asks the TM to suspend the RMs in the thread of control because of the blocking receive call from the AP.
16. The TM issues *xa_end*(TMSUSPEND) to the local RM.
17. The TM issues *xa_end*(TMSUSPEND) to the CRM because it was registered.
18. The CRM in the subordinate responds to the *cm_listen*( ) issued by the AP, and the subordinate AP accepts the dialogue.
19. The CRM informs the TM in the subordinate that a transaction branch is about to begin.
20. The TM passes the transaction branch's XID to the statically registering local RM. The TM does not issue *xa_start*( ) to the CRM because it has selected dynamic registration.
21. The CRM in the subordinate requests the TM to save necessary information to keep track of the superior. This might include routing information and any other information that the CRM needs, especially during recovery.
22. The CRM returns success to the subordinate AP's *cm_accept*( ) call.
23. The subordinate AP receives the message from the superior.
24. The subordinate AP sends a message to the superior.
25. The response is sent by the CRM to the superior.
26. The CRM in the superior calls the TM to resume the association with the thread of control.
27. The TM resumes the association with the local RM. The TM does not resume the association with the CRM because it has selected dynamic registration.
28. The response is returned to the superior AP.

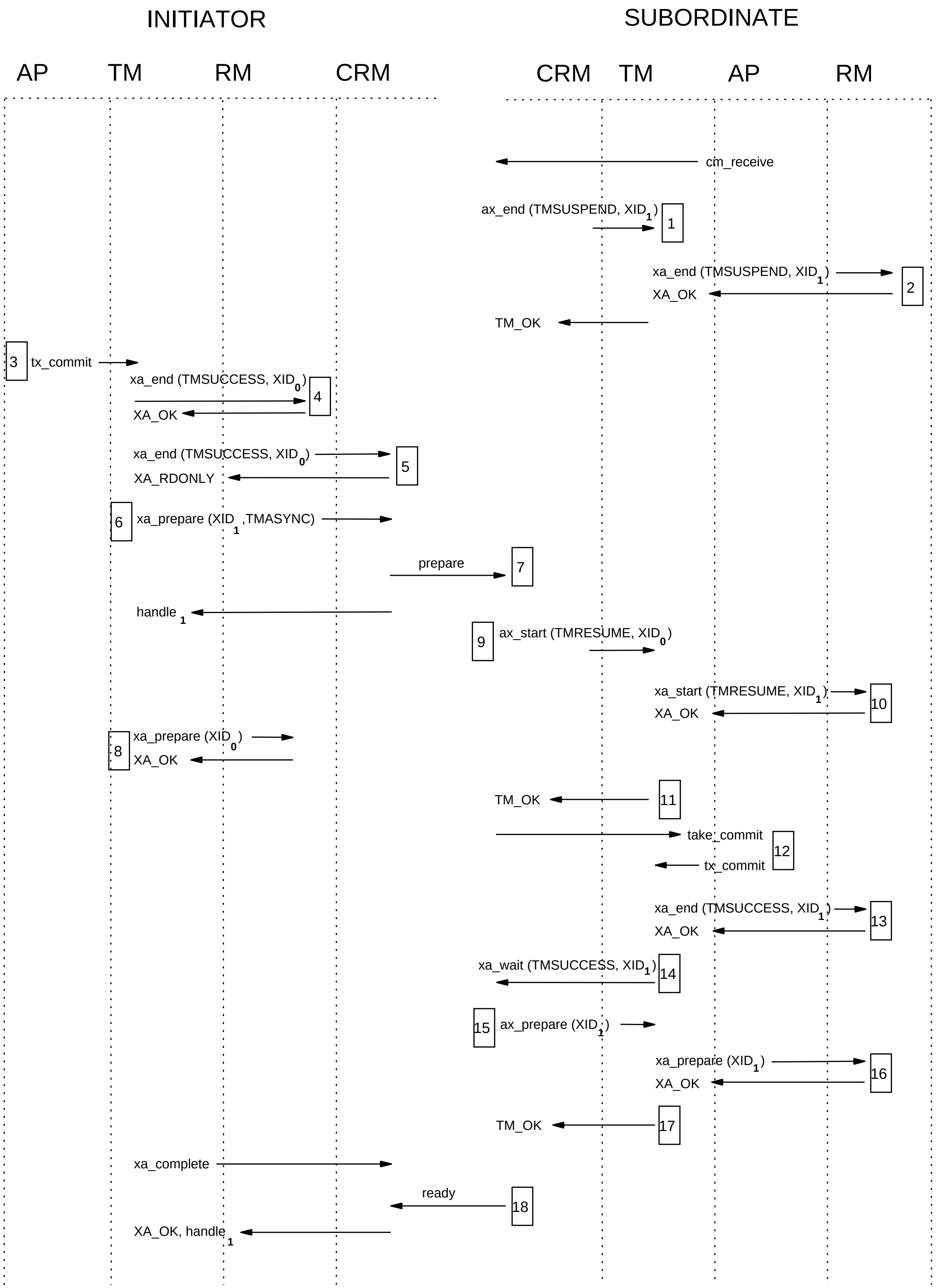
29. The subordinate blocks on a receive.
30. The CRM in the subordinate calls the TM to suspend the transaction's association in the thread of control because of a blocking receive done by the subordinate AP.
31. The TM in the subordinate suspends the association with the local RM in the thread of control. The TM does not call the CRM because it uses dynamic registration and it has not registered.

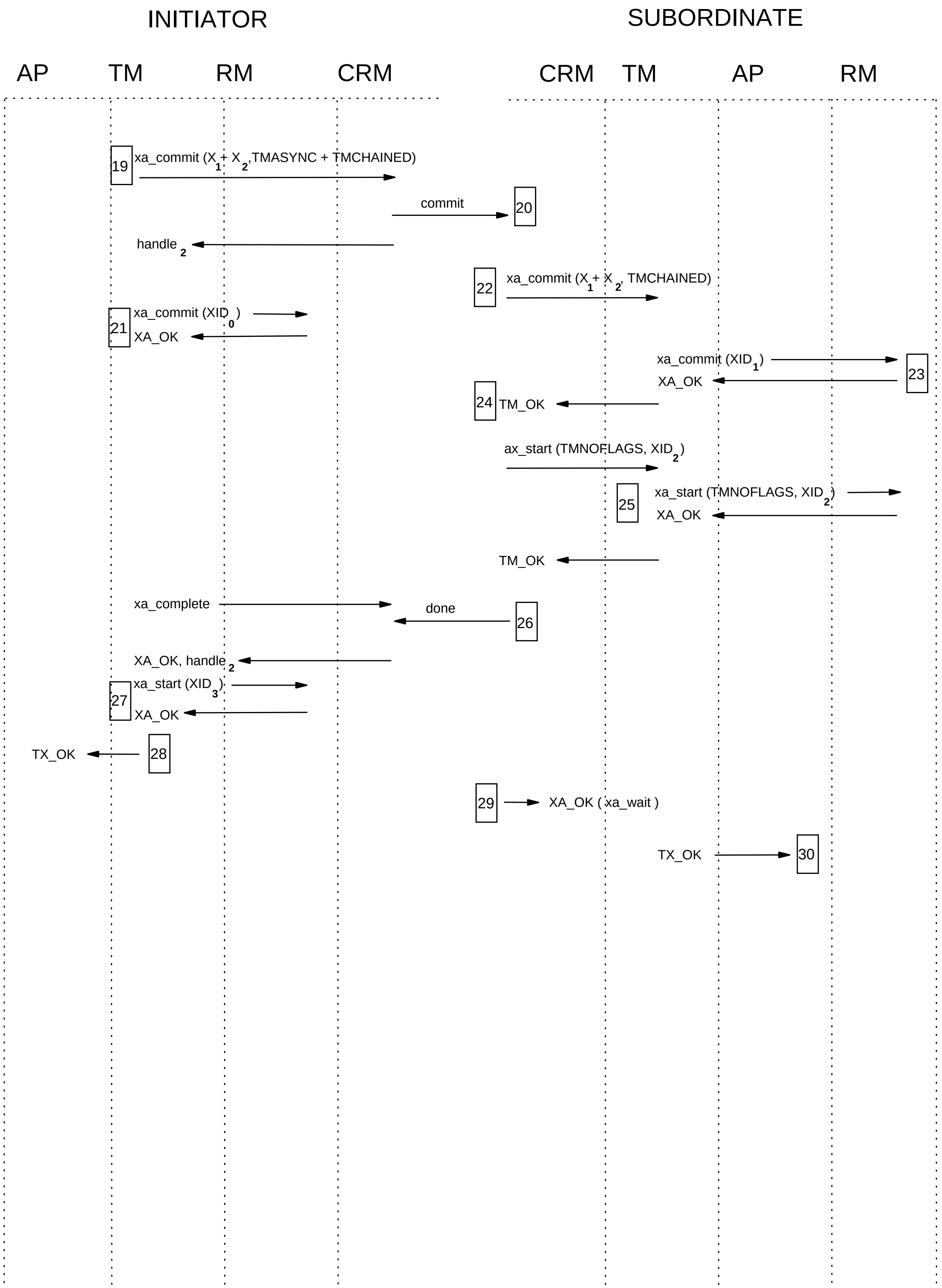
B.3 Peer-to-Peer Two-Phase Commit Chained Transaction

This scenario shows peer-to-peer transaction commitment for a transaction mandatory (chained transaction) dialogue.

The following notes describe the flows in the scenario opposite:

1. The CRM in the subordinate calls the TM to suspend the transaction's association in the thread of control because of a blocking receive done by the subordinate AP.
2. The TM in the subordinate suspends the association with the local RM in the thread of control. The TM does not call the CRM because it uses dynamic registration and it has not registered.
3. The AP in the superior calls the TM to commit the transaction.
4. The TM in the superior ends the association with the local RM.
5. The TM ends the suspended association with the CRM. The CRM responds read-only so that it will not be included further for the root XID.
6. The TM in the superior asks the CRM to prepare the transaction branch in the subordinate.
7. The CRM sends the prepare message to the subordinate.
8. The TM in the superior prepares the local RM.
9. The CRM in the subordinate calls the TM to resume the association with the thread of control.
10. The TM in the subordinate resumes the association with the local RM. The TM does not resume the association with the CRM because it has selected dynamic registration.
11. The CRM gives a *take commit* indication to the subordinate AP.
12. The AP in the subordinate calls the TM to commit the transaction.
13. The TM in the subordinate ends the association with the local RM. The TM does not call the CRM because it never registered.
14. The TM in the subordinate calls the CRM to wait for the prepare indication.
15. The CRM in the subordinate propagates the prepare to the TM.
16. The TM prepares the transaction branch in the local RM in the subordinate. The CRM is not prepared because it never registered for the transaction branch.
17. The TM in the subordinate logs the ready decision.
18. The ready response is sent to the superior.


INITIATOR
SUBORDINATE
AP
TM
RM
CRM
CRM
TM
AP
RM
cm_receive
ax_end (TMSUSPEND, XID1)
1
xa_end (TMSUSPEND, XID1)
XA_OK
2
TM_OK
3
tx_commit
xa_end (TMSUCCESS, XID0)
XA_OK
4
xa_end (TMSUCCESS, XID0)
XA_RDONLY
5
6
xa_prepare (XID1 ,TMASYNC)
prepare
7
handle 1
9
ax_start (TMRESUME, XID0)
xa_start (TMRESUME, XID1)
XA_OK
10
8
xa_prepare (XID0)
XA_OK
TM_OK
11
take_commit
tx_commit
12
xa_end (TMSUCCESS, XID1)
XA_OK
13
xa_wait (TMSUCCESS, XID1)
14
15
ax_prepare (XID1)
xa_prepare (XID1)
XA_OK
16
TM_OK
17
xa_complete
ready
18
XA_OK, handle 1


INITIATOR
SUBORDINATE
AP
TM
RM
CRM
CRM
TM
AP
RM
19
xa_commit (X_1 + X_2,TMASYNC + TMCHAINED)
commit
20
handle $_2$
22
xa_commit (X_1 + X_2, TMCHAINED)
21
xa_commit (XID $_0$)
XA_OK
xa_commit (XID_1)
XA_OK
23
24
TM_OK
ax_start (TMNOFLAGS, XID_2)
25
xa_start (TMNOFLAGS, XID_2)
XA_OK
TM_OK
xa_complete
done
26
XA_OK, handle $_2$
27
xa_start (XID_3)
XA_OK
TX_OK
28
29
XA_OK (xa_wait)
TX_OK
30

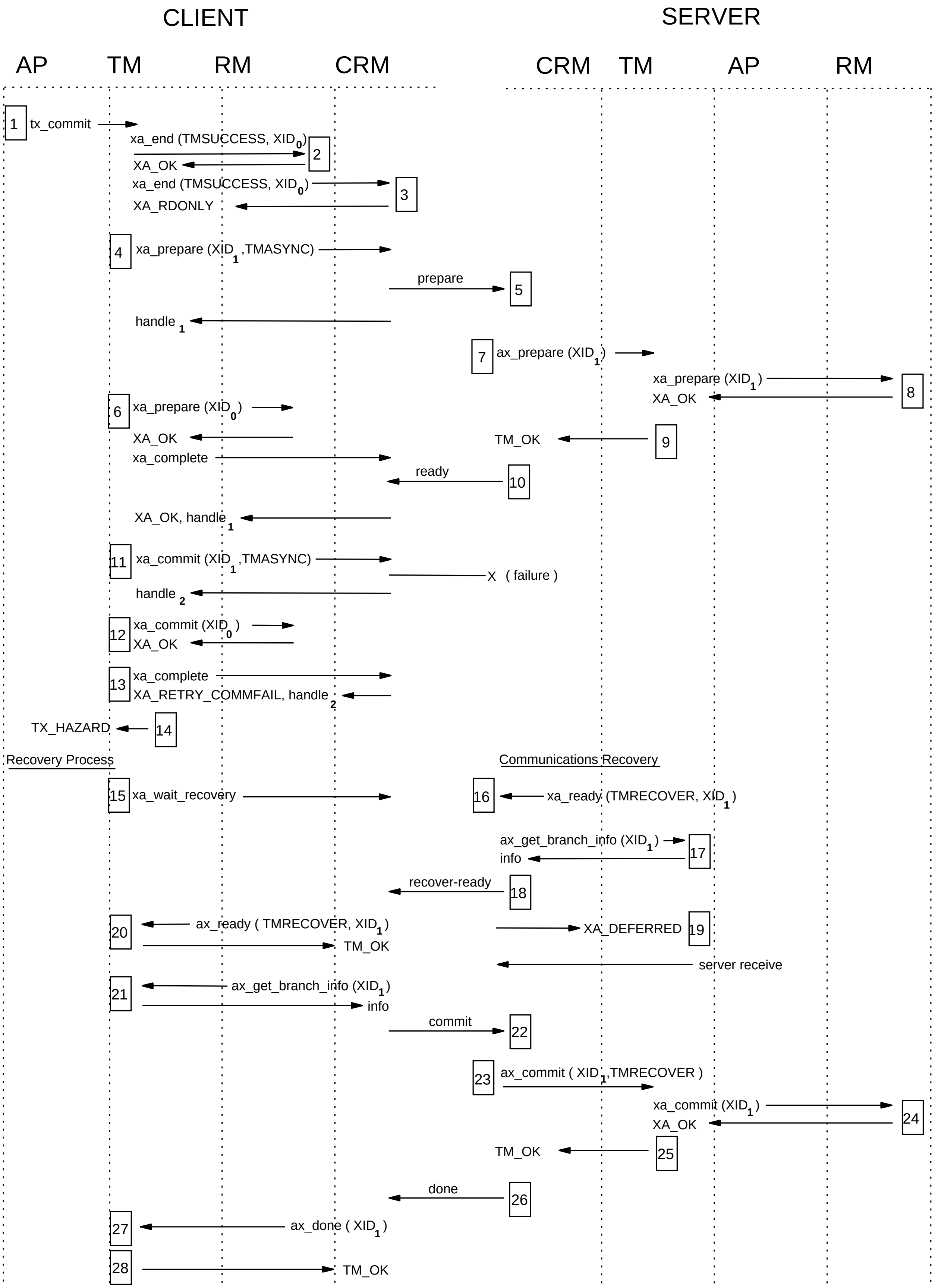
19. The coordinator TM logs the ready decision and asks the CRM to commit the transaction branch in the subordinate. The TM passes the branch's XID and a new XID for the chained transaction in the subordinate.
20. The commit message is sent by the CRM to the subordinate.
21. The TM in the superior commits the local RM.
22. The CRM in the subordinate propagates the commit decision to the TM. The CRM passes the branch's XID and the new one for the chained transaction.
23. The TM commits the transaction branch in the local RM in the subordinate.
24. The TM in the subordinate logs the commit decision. This is a forget transaction branch decision. The TM returns to the CRM. The CRM starts the chained transaction branch in the TM in the subordinate.
25. The TM starts the chained transaction's association with the local RM in the subordinate.
26. The commit response is sent to the superior.
27. The coordinator TM logs the commit decision. This is a forget transaction decision. The TM starts the association for the chained transaction in the local RM.
28. The TM responds to the superior AP informing it that *tx_commit*( ) was successful.
29. The CRM in the subordinate responds to the TM's *xa_wait*( ) call.
30. The TM responds to the subordinate AP informing it that *tx_commit*( ) was successful.

B.4 Network Failure - Client Decided, Subordinate Initiated Recovery

This scenario shows client/server interaction global transaction commitment with a communication failure after the prepare phase of the two-phase commit procedure is complete. In this example, the server initiates recovery when communications are re-established.

The following notes describe the flows in this scenario:

1. The client AP calls the TM to commit the transaction.


CLIENT
SERVER
AP
TM
RM
CRM
CRM
TM
AP
RM
1 tx_commit
xa_end (TMSUCCESS, XID$_0$)
2
XA_OK
xa_end (TMSUCCESS, XID$_0$)
3
XA_RDONLY
4 xa_prepare (XID$_1$,TMASYNC)
prepare
5
handle $_1$
7 ax_prepare (XID$_1$)
xa_prepare (XID$_1$)
8
XA_OK
6 xa_prepare (XID$_0$)
XA_OK
TM_OK
9
xa_complete
ready
10
XA_OK, handle $_1$
11 xa_commit (XID$_1$,TMASYNC)
X (failure)
handle $_2$
12 xa_commit (XID$_0$)
XA_OK
13 xa_complete
XA_RETRY_COMMFAIL, handle $_2$
TX_HAZARD
14
Recovery Process
Communications Recovery
15 xa_wait_recovery
16
xa_ready (TMRECOVER, XID$_1$)
ax_get_branch_info (XID$_1$)
17
info
recover-ready
18
20
ax_ready (TMRECOVER, XID$_1$)
TM_OK
XA_DEFERRED
19
server receive
21
ax_get_branch_info (XID$_1$)
info
commit
22
23 ax_commit (XID$_1$,TMRECOVER)
xa_commit (XID$_1$)
24
XA_OK
TM_OK
25
done
26
27
ax_done (XID$_1$)
28
TM_OK

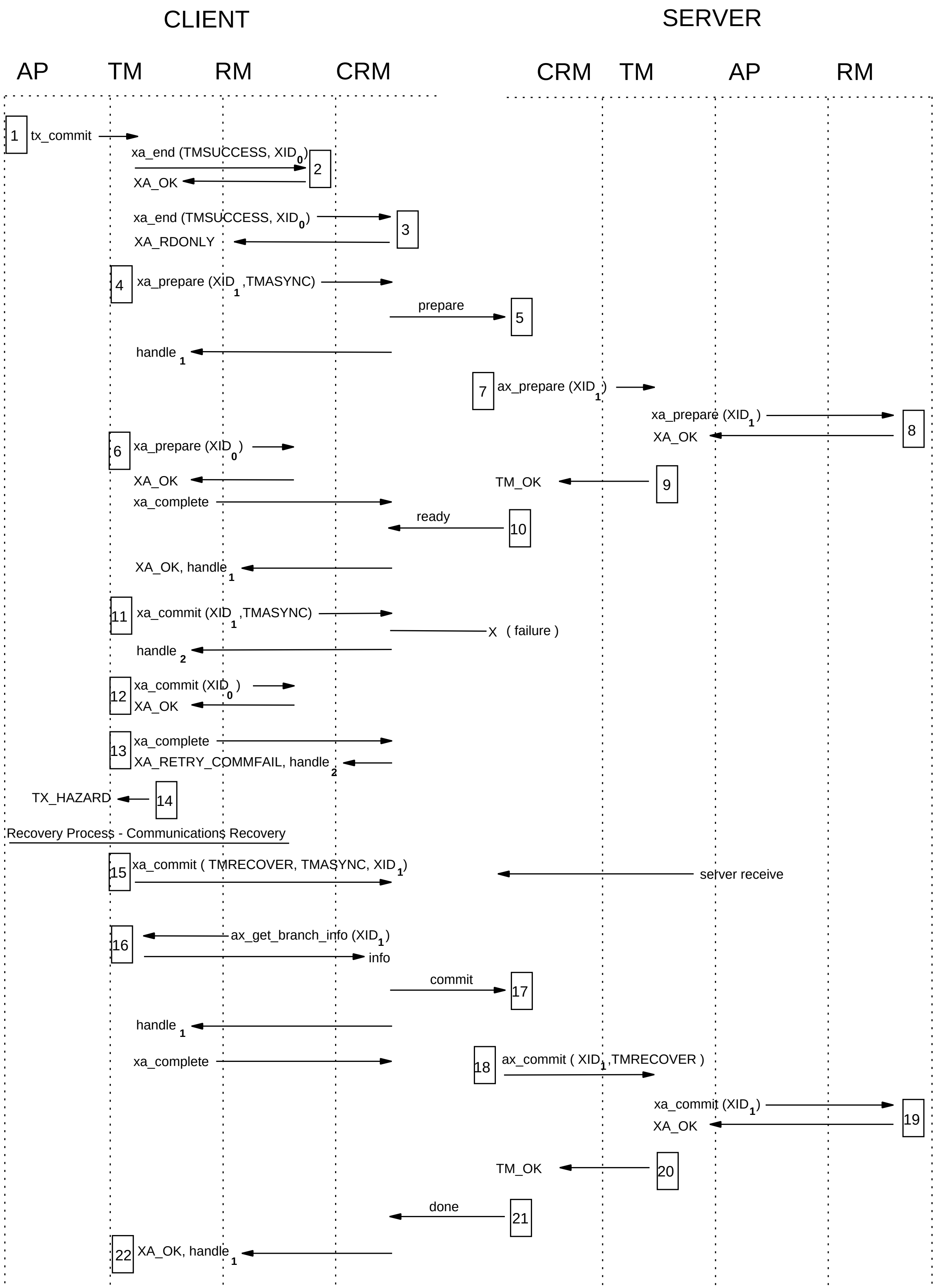
2. The TM ends the association with the local RM.
3. The TM ends the suspended association with the CRM. The CRM responds read-only so that it will not be included further for the root XID.
4. The TM in the client asks the CRM to prepare the transaction branch in the server.
5. The CRM sends the prepare message to the subordinate (server).
6. The TM in the client prepares the local RM.
7. The CRM propagates the prepare to the remote TM.
8. The TM prepares the transaction branch in the local RM in the server. The CRM is not prepared because it never registered for the transaction branch.
9. The TM in the server logs the ready decision.
10. The ready response is sent to the superior.
11. The coordinator TM logs the ready decision and asks the CRM to commit the transaction branch in the server.
12. The TM in the client commits the local RM.
13. The CRM in the client indicates that it cannot commit now because of communications failure.
14. The coordinator TM does not log the commit decision because it is incomplete. The TM returns a heuristic hazard to the client AP because the TM cannot guarantee commitment.
15. The CRM in the recovery process in the client node is given control by the TM.
16. The TM initiates recovery in the server for the prepared transaction.
17. The CRM in the server requests its saved information from the TM.
18. The CRM sends a recover-ready message to the superior.
19. The CRM in the server informs the TM that the recovery decision is deferred.
20. The CRM in the superior asks the TM for the status of the transaction. The TM responds that it should be committed.
21. The CRM retrieves its saved information for the transaction branch.
22. The commit message is sent by the CRM to the subordinate (server).
23. The CRM in the server propagates the commit decision to the TM.
24. The TM commits the transaction branch in the local RM in the server.
25. The TM in the server logs the commit decision. This is a forget transaction branch decision.
26. The commit response is sent to the superior.
27. The CRM in the client informs the TM that the commit was successful in the subordinate branch.
28. The coordinator TM logs the commit decision because the transaction is now fully committed. This is a forget transaction decision.

B.5 Network Failure - Client Decided, Coordinator Initiated Recovery

This scenario shows client/server interaction global transaction commitment with a communication failure after the prepare phase of the two-phase commit procedure is complete. In this example, the client initiates recovery when communications are re-established.

The following notes describe the flows in the scenario opposite:

1. The client AP calls the TM to commit the transaction.
2. The TM ends the association with the local RM.
3. The TM ends the suspended association with the CRM. The CRM responds read-only so that it will not be included further for the root XID.
4. The TM in the client asks the CRM to prepare the transaction branch in the server.
5. The CRM sends the prepare message to the subordinate (server).
6. The TM in the client prepares the local RM.
7. The CRM propagates the prepare to the remote TM.
8. The TM prepares the transaction branch in the local RM in the server. The CRM is not prepared because it never registered for the transaction branch.
9. The TM in the server logs the ready decision.
10. The ready response is sent to the superior.
11. The coordinator TM logs the ready decision and asks the CRM to commit the transaction branch in the server.
12. The TM in the client commits the local RM.
13. The CRM in the client indicates that it cannot commit now because of communications failure.
14. The coordinator TM does not log the commit decision because it is incomplete. The TM returns a heuristic hazard to the client AP because the TM cannot guarantee commitment.
15. The TM initiates recovery in the superior for the prepared transaction.
16. The CRM requests its saved information from the TM.
17. The commit message is sent by the CRM to the subordinate (server).
18. The CRM propagates the commit decision to the remote TM.
19. The TM commits the transaction branch in the local RM in the server.
20. The TM in the server logs the commit decision. This is a forget transaction branch decision.
21. The commit response is sent to the superior.
22. The CRM informs the coordinator TM that the commit was successful in the subordinate branch. The coordinator TM logs the commit decision because the transaction is now fully committed. This is a forget transaction decision.

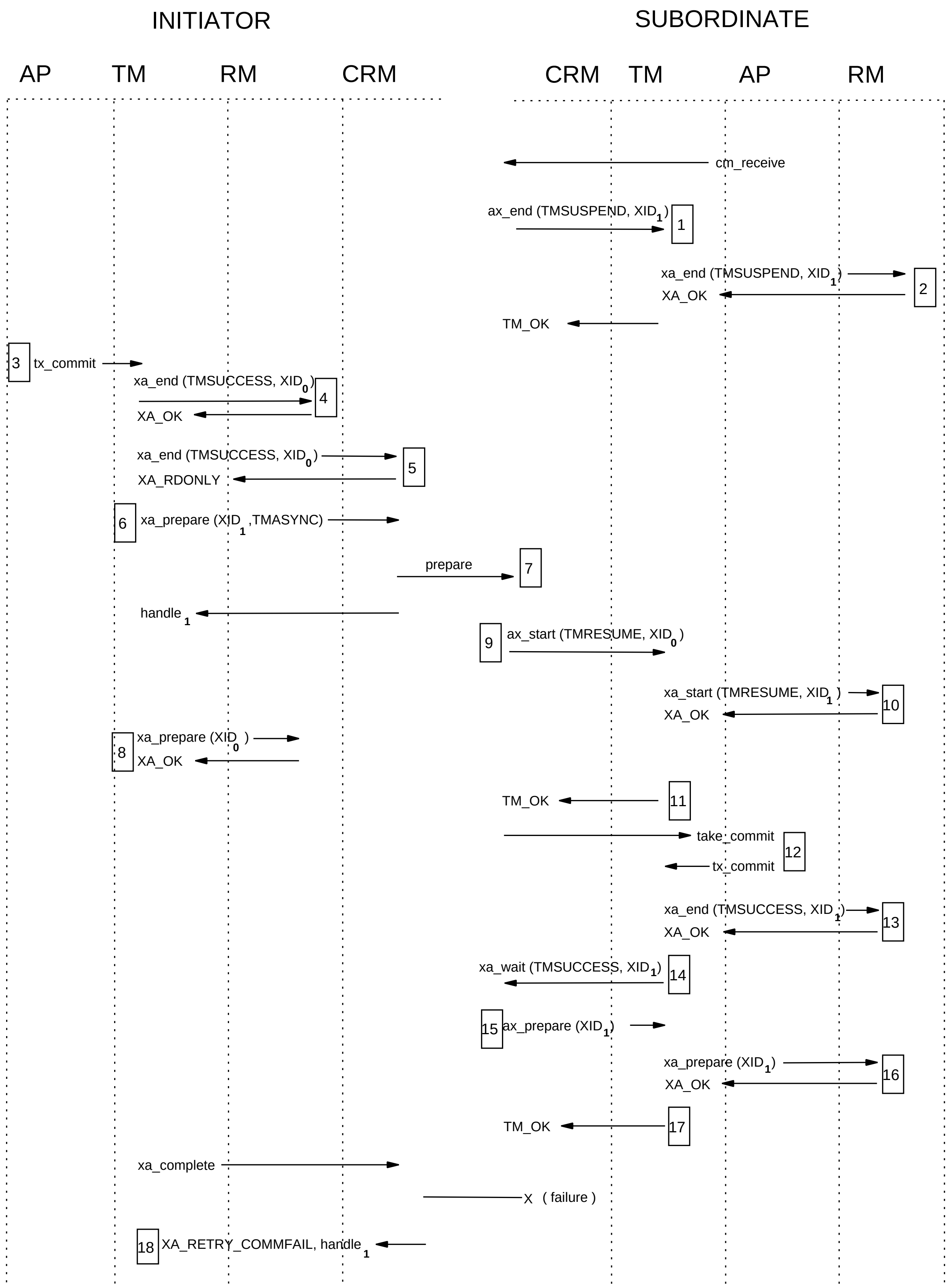

CLIENT
SERVER
AP
TM
RM
CRM
CRM
TM
AP
RM
1
tx_commit
xa_end (TMSUCCESS, XID0)
2
XA_OK
xa_end (TMSUCCESS, XID0)
3
XA_RDONLY
4
xa_prepare (XID1,TMASYNC)
prepare
5
handle 1
7
ax_prepare (XID1)
xa_prepare (XID1)
8
XA_OK
6
xa_prepare (XID0)
XA_OK
TM_OK
9
xa_complete
ready
10
XA_OK, handle 1
11
xa_commit (XID1,TMASYNC)
X (failure)
handle 2
12
xa_commit (XID0)
XA_OK
13
xa_complete
XA_RETRY_COMMFAIL, handle 2
TX_HAZARD
14
Recovery Process - Communications Recovery
15
xa_commit (TMRECOVER, TMASYNC, XID1)
server receive
16
ax_get_branch_info (XID1)
info
commit
17
handle 1
xa_complete
18
ax_commit (XID1,TMRECOVER)
xa_commit (XID1)
19
XA_OK
TM_OK
20
done
21
22
XA_OK, handle 1

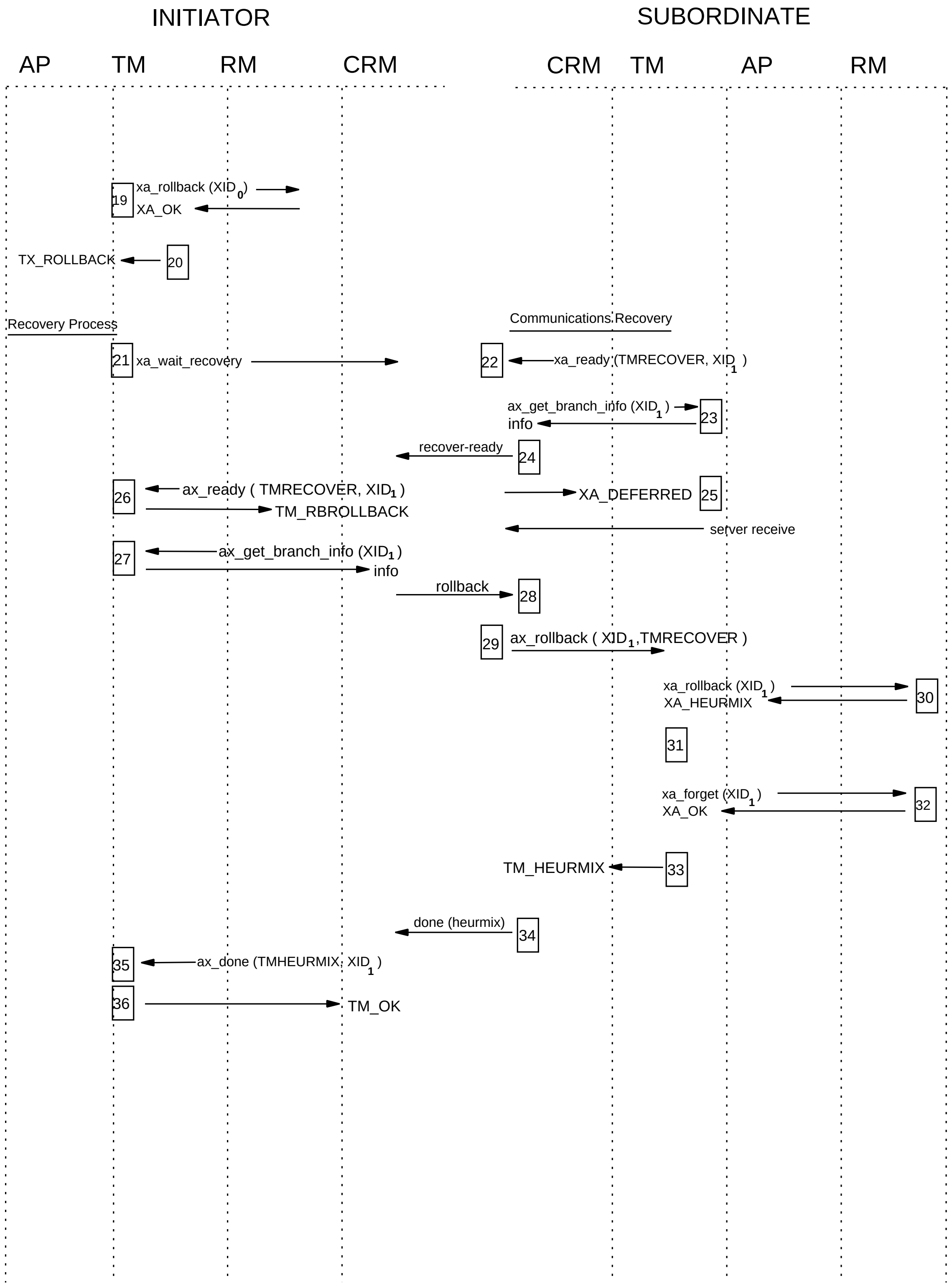
B.6 Network Failure - Heuristic Damage During Subordinate Initiated Recovery

This scenario shows a recovery following a network failure during the prepare phase. After propagating the outcome at the coordinator to the subordinate, the CRM detects heuristic damage in the transaction branch.

The following notes describe the flows in the scenario opposite:

1. The CRM in the subordinate calls the TM to suspend the transaction's association in the thread of control because of a blocking receive done by the subordinate AP.
2. The TM in the subordinate suspends the association with the local RM in the thread of control. The TM does not call the CRM because it uses dynamic registration and it has not registered.
3. The AP in the superior calls the TM to commit the transaction.
4. The TM in the superior ends the association with the local RM.
5. The TM ends the suspended association with the CRM. The CRM responds read-only so that it will not be included further for the root XID.
6. The TM in the superior asks the CRM to prepare the transaction branch in the subordinate.
7. The CRM sends the prepare message to the subordinate.
8. The TM in the superior prepares the local RM.
9. The CRM in the subordinate calls the TM to resume the association with the thread of control.
10. The TM in the subordinate resumes the association with the local RM. The TM does not resume the association with the CRM because it has selected dynamic registration.
11. The CRM gives a *take commit* indication to the subordinate AP.
12. The AP in the subordinate calls the TM to commit the transaction.
13. The TM in the subordinate ends the association with the local RM. The TM does not call the CRM because it never registered.
14. The TM in the subordinate calls the CRM to wait for the prepare indication.
15. The CRM in the subordinate propagates the prepare to the TM.
16. The TM prepares the transaction branch in the local RM in the subordinate. The CRM is not prepared because it never registered for the transaction branch.
17. The TM in the subordinate logs the ready decision.
18. The CRM in the superior indicates that it cannot commit now because of a communication failure.


INITIATOR
SUBORDINATE
AP
TM
RM
CRM
CRM
TM
AP
RM
cm_receive
ax_end (TMSUSPEND, XID1)
1
xa_end (TMSUSPEND, XID1)
XA_OK
2
TM_OK
3
tx_commit
xa_end (TMSUCCESS, XID0)
4
XA_OK
xa_end (TMSUCCESS, XID0)
5
XA_RDONLY
6
xa_prepare (XID1,TMASYNC)
prepare
7
handle1
9
ax_start (TMRESUME, XID0)
xa_start (TMRESUME, XID1)
10
XA_OK
xa_prepare (XID0)
8
XA_OK
TM_OK
11
take_commit
12
tx_commit
xa_end (TMSUCCESS, XID1)
13
XA_OK
xa_wait (TMSUCCESS, XID1)
14
15
ax_prepare (XID1)
xa_prepare (XID1)
16
XA_OK
TM_OK
17
xa_complete
X (failure)
18
XA_RETRY_COMMFAIL, handle1


INITIATOR
SUBORDINATE
AP
TM
RM
CRM
CRM
TM
AP
RM
19
xa_rollback (XID0)
XA_OK
TX_ROLLBACK
20
Recovery Process
Communications Recovery
21
xa_wait_recovery
22
xa_ready (TMRECOVER, XID1)
ax_get_branch_info (XID1)
info
23
recover-ready
24
26
ax_ready (TMRECOVER, XID1)
TM_RBROLLBACK
XA_DEFERRED
25
server receive
27
ax_get_branch_info (XID1)
info
rollback
28
29
ax_rollback (XID1,TMRECOVER)
xa_rollback (XID1)
XA_HEURMIX
30
31
xa_forget (XID1)
XA_OK
32
TM_HEURMIX
33
done (heurmix)
34
35
ax_done (TMHEURMIX, XID1)
36
TM_OK

19. The TM in the superior instructs the local RM to roll back.
20. The TM returns the rollback outcome to the AP.
21. The CRM in the recovery process in the superior is given control by the TM.
22. The TM initiates recovery in the subordinate for the prepared transaction.
23. The CRM in the subordinate requests its saved information from the TM.
24. The CRM sends a recover-ready message to the superior.
25. The CRM in the subordinate informs the TM that the recovery decision is deferred.
26. The CRM in the superior asks the TM for the status of the transaction. The TM responds that it should be rolled back.
27. The CRM retrieves its saved information for the transaction branch.
28. The rollback message is sent by the CRM to the subordinate.
29. The CRM in the subordinate propagates the rollback decision to the TM.
30. The TM rolls back the transaction branch in the local RM in the subordinate. The RM reports back a mixed heuristic outcome.
31. The TM in the subordinate logs the heuristic decision.
32. The TM permits the RM to erase its knowledge of the heuristically completed transaction branch.
33. The TM reports the mixed heuristic outcome to the CRM.
34. The rollback response (mixed heuristic outcome) is sent to the superior.
35. The CRM in the superior informs the TM that the rollback resulted in a mixed heuristic outcome.
36. The superior TM logs the heuristic decision.

Index