DECdta - Digital's Distributed Transaction Processing Architecture

By Philip A. Bernstein, William T. Emberton, Vijay Treba

Abstract

Digital's Distributed Transaction Processing Architecture (DECdta) describes the modules and interfaces that are common to Digital's transaction processing (DECtp) products. The architecture allows easy distribution of DECtp products. In particular, it supports client/server style applications. Distributed transaction management is the main function that ties DECdta modules together. It ensures that application programs, database systems, and other resource managers interoperate reliably in a distributed system.

Introduction

Transaction processing (TP) is the activity of executing requests to access shared resources, typically databases. A computer system that is configured to execute TP

- Atomicity. Either all of the transaction's operations execute, or the transaction has no effect at all.
- Serializability. The set of all operations that execute on behalf of the transaction appears to execute serially with respect to the set of operations executed by every other transaction.
- Durability. The effects of the transaction's operations are resistant to failures.

A transaction terminates by executing the commit or abort operation. Commit tells the system to install the effect of the transaction's operations permanently. Abort tells the system to undo the effects of the transaction's operations.

For enhanced reliability and availability, a TP application uses

applications is called a TP system.

A transaction is an execution of a set of operations on shared resources that has the following properties:

transactions to execute requests. That is, the application receives a request message (from a display, computer, or other device), executes one or more transactions to process the request, and possibly sends a reply

to the originator of the request or to some other party specified by the originator.

TP applications are essential to the operation of many industries, such as finance, retail, health care, transportation, government, communications, and manufacturing. Given the broad range of applications of TP, Digital offers a wide variety of products with which to build TP systems.

DECtp is an umbrella term that refers to Digital's TP products. The goal of DECtp is to offer an integrated set of hardware and software products that supports the development, execution, and management of TP applications for enterprises of all sizes.

DECtp systems include software components that are specialized for TP, notably TP monitors such as the ACMS and DECintact TP monitors, and transaction managers such as the DECdtm transaction manager. [1] [2] DECtp systems also require the integration of general-purpose hardware products (processors, storage, communications, and terminals) and software products (operating systems, database systems, and communication

gateways). These products are typically integrated as shown in Figure 1.

DECdta-Digital's Distributed Transaction Processing Architecture

Applications on DECtp systems can be designed using a client/server paradigm. This paradigm is especially useful for separating the work of preparing a request from that of running transactions. Request preparation can be done by a front-end system, that is, one that is close to the user, in which processor cycles are inexpensive and interactive feedback is easy to obtain. Transaction execution can be done by a larger back-end system, that is, one that manages large databases and may be far from the user. Back-end systems may themselves be distributed. Each back-end system manages a portion of the enterprise database and executes applications, usually ones that make heavy use of the database on that back end. DECtp products are modularized to allow easy distribution across front ends and back ends, which enables them to support client /server style applications. DECtp systems thereby simplify programming and reconfiguration in a distributed system.

Digital's Distributed Transaction Processing Architecture (DECdta) and explains how DECdta components are integrated by distributed transaction management.

Current versions of DECtp products implement most, but not all, modules and interfaces in the DECdta architecture. Gaps between the architecture and products will be filled over time. DECtp products that currently implement DECdta components are referenced throughout the paper.

TP Application Structure

By analyzing TP applications, we can see where the need arises for separate DECdta components. A typical TP application is structured as follows:

Step 1: The client application interacts with a user (a person or machine) to gather input, e.g., using a forms manager.

Step 2: The client maps the user's input into a request, that is, a message that asks the system to perform some work. The client sends the request to a server application to process the request.

A request may be direct or queued. If direct, the client expects a

defines the modularization and distribution structure that is common to DECtp products. Distributed transaction management is the main function that ties this structure together. This paper describes the DECdta structure

server to process the request right away. If queued, the client deposits the request in a queue from which a server can dequeue the request later.

Step 3: A server processes the request by executing one or more transactions. Each transaction may

a. Access multiple resources

b. Call programs, some of which may be remote

c. Generate requests to execute other transactions

d. Interact with a user

e. Return a reply when the transaction finishes

Step 4: If the transaction produces a reply, then the client interacts with the user to display that reply, e.g., using a forms manager.

Each of the above steps involves the interaction of two or more programs. In many cases, it is desirable that these programs be distributed. To distribute them conveniently, it is important that the programs be in separate components. For example, consider the following:

- The presentation service that operates the display and the application that controls which form to display may be

  One may want to off-load presentation services and related functions to front ends, while allowing programs on back ends to control which forms are displayed to users. This capability is useful in Steps 1, 3d, and 4 above to gather input and display output. To ensure that the presentation service and application can be distributed, the presentation service should correspond to a separate DECdta component.

- The client application that sends a request and the server application that processes the request may be distributed. The applications may communicate through a network or a queue.

  In Step 2, front-end applications may want to send requests directly to back-end applications or to place requests in queues that are managed on back ends. Similarly, in Step 3c, a transaction, T, may enqueue a request to run another transaction, where the queue resides on a different system than T. To maximize the flexibility of

distributed.

distributing request

management, request management should correspond to a separate DECdta component.

- Two transaction managers that want to run a commit protocol may be distributed.

For a transaction to be distributed across different systems, as in Step 3b, the transaction management services must be distributed. To ensure that each transaction is atomic, the transaction managers on these systems must control transaction commitment using a common commit protocol. To complicate matters, there is more than one widely used protocol for transaction commitment. To the extent possible, a system should allow interoperation of these protocols.

To ensure that transaction managers can be distributed, the transaction manager should be a component of DECdta. To ensure that they can interoperate, their transaction protocol should also be in DECdta. To ensure that different commit protocols can be supported, the part of transaction management that defines the protocol for interaction with remote transaction managers should be separated from the part that coordinates transaction execution across local resources.

Interoperation of transaction managers and resource managers, such as database systems, also affects the modularization of DECdta components. A transaction may involve different types of resources, as in Step 3a. For example, it may update data that is managed by different database systems. To control transaction commitment, the transaction manager must interact with different resource managers, possibly supplied by different vendors. This requires that resource managers be separate components of DECdta.

The DECdta Architecture

Having seen where the need for DECdta components arises, we are now ready to describe the DECdta architecture as a whole, including the functions of and interfaces to each component.

Most DECdta interfaces are public. Some of the public interfaces are controlled by official standards bodies and industry consortia; i.e., they are "open" interfaces. Others are controlled solely by Digital. DECdta interfaces and protocols will be published and aligned with industry standards, as

In the DECdta architecture, the former is called a communication manager, and the latter is called a transaction manager.

appropriate.

DECdta components are abstract entities. They do not necessarily map one-to-one to hardware components, software components (e.g., programs or products), or execution environments

(e.g., a single-threaded process, a multithreaded process, or an operating system service). Rather, a DECdta component may be implemented as multiple software components, for example, as several processes. Alternatively, several DECdta components may be implemented as a single software component. For example, an operating system or TP monitor typically offers the facilities of more than one DECdta component.

The following are the components of DECdta:

- An application program is any program that uses services of DECdta components.
- A resource manager manages resources that support transaction semantics.
- A transaction manager coordinates transaction termination (i.e., commit and abort).
- A communication manager supports a transaction communication protocol between TP systems.
- A presentation manager supports device-independent interactions with a presentation device.

DECdta components are layered on services that are provided by the underlying operating system and distributed system platform, and are not specific to TP, as shown in Figure 2.

- A request manager facilitates the submission of requests to execute transactions.

Application Program

We use the term application program to mean a program that uses the services provided by other DECdta components. An application program could be a customer-written program, a layered product, or a DECdta component.

In the DECdta architecture, we distinguish two special types of application program: request initiators and transaction servers. A request initiator is a DECdta component that prepares and submits a request for the execution of a transaction. To create a request, the request initiator usually interacts with a presentation manager that provides an interface to a device, such as a terminal, a workstation, a digital private branch exchange, or an automated teller machine.

A transaction server can demarcate a transaction, interact with one or more resource managers to access recoverable resources on behalf of the transaction, invoke other transaction servers, and respond to calls from request initiators.

For a simple request, a transaction server receives the request, processes messages with the user, usually through the request initiator.

In principle, a request initiator could also execute transactions (not shown in Figure 2). That is, the distinction between request initiators and transaction servers is for clarity only,

and does not restrict an application from performing request initiation functions in a transaction. Architecturally, this amounts to saying that request initiation functions can execute in a transaction server.

Resource Manager

A resource manager performs operations on shared resources. We are especially interested in recoverable resource managers, those that obey transaction semantics. In particular, a recoverable resource manager undoes a transaction's updates to the resources if the transaction aborts. Other recoverable resource manager activities in support of transactions are described in the next section. In the rest of this paper, we use "resource manager" to mean "recoverable resource manager."

In a TP system, the most

it, and optionally returns a reply to the request initiator. A conversational request is like a simple request, except that while processing the request, the transaction server exchanges one or more

common kind of resource manager is a database system. Some presentation managers and communication managers may also be resource managers. A resource manager may be

written by a customer, a third party, or Digital. Each resource manager type offers a resource-manager-specific interface that is used by application programs to access and modify recoverable resources managed by the resource manager. A description of these resource manager interfaces is outside the scope of DECdta. However, many of these resource manager interfaces have architectures defined by industry standards, such as SQL (e.g., the VAX Rdb/VMS product), CODASYL data manipulation language (e.g., the VAX DBMS product), and COBOL file operations (e.g., RMS in the VMS system).

One type of resource manager that plays a special role in TP systems is a queue resource manager. It manages recoverable queues, which are often used to store requests. [3] It allows application programs to place elements into queues and retrieve them, so that application programs can communicate even though they execute independently and asynchronously. For example, an application program that sends elements can communicate with one that receives

A queue resource manager interface supports such operations as open-queue, close-queue, enqueue, dequeue, and read-element. The ACMS and DECintact TP monitors both have queue resource managers as components.

Transaction Manager

A transaction manager supports the transaction abstraction. It is responsible for ensuring the atomicity of each transaction by telling each resource manager in a transaction when to commit. It uses a two-phase commit protocol to ensure that either all resource managers accessed by a transaction commit the transaction or they all abort the transaction.

[4] To support transaction atomicity, a transaction manager provides the following functions:

- Transaction demarcation operations allow application programs or resource managers to start and commit or abort a transaction. (Resource managers sometimes start a transaction to execute a resource operation if the caller is not executing a transaction. The SQL standard requires this.)

elements even if the two application programs are not operational simultaneously. This communication arrangement improves availability and facilitates batch input of elements.

- Transaction execution operations allow resource managers and communication managers to declare themselves part of an existing transaction.

- Two-phase commit operations allow resource managers and communication managers to change a transaction's state (to "prepared," "committed," or "aborted").

The serializability of transactions is primarily the responsibility of the resource managers. Usually, a resource manager ensures serializability by setting locks on resources accessed by each transaction, and by releasing the locks after the transaction manager tells the resource manager to commit. (The latter activity makes serializability partly the responsibility of the transaction manager.) If transactions become deadlocked, a resource manager may detect the deadlock and abort one of the deadlocked transactions.

The durability of transactions is a responsibility of transaction managers and resource managers. The transaction manager is responsible for the durability of the commit or abort decision. A resource manager is responsible for the durability of operations of committed transactions. Usually, states while recovering from a failure.

A detailed description of the DECdta transaction manager component appears in the Transaction Manager Architecture section.

Communication Manager

A communication manager provides services for communication between named objects in a TP system, such as application programs and transaction managers. Some communication managers participate in coordinating the termination of a transaction by propagating the transaction manager's two-phase commit operations as messages to remote communication managers. Other communication managers propagate application data and transaction context, such as a transaction identifier, from one node to another. Some do both.

A TP system can support multiple communication managers. These communication managers can interact with other nodes using different commit protocols or message-passing protocols, and may be part of different name spaces, security domains, system management domains, etc. Examples are an IBM

it ensures durability by storing a description of each transaction's resource operations and state changes in a stable (e.g., disk-resident) log. It can later use the log to reconstruct transactions'	SNA LU6.2 communication manager or an ISO-TP communication manager.

By supporting multiple communication managers, the DECdta architecture enhances the interoperability of TP systems. Different TP systems can interoperate by executing a transaction using different commit protocols.

A communication manager offers an interface for application programs to communicate with other application programs. Different communication managers may offer different communication paradigms, such as remote procedure call or peer-to-peer message passing.

A communication manager also has an interface to its local transaction manager. It uses this interface to tell the transaction manager when a transaction has spread to a new node and to obtain information about transaction commitment, which it exchanges with communication managers on remote nodes.

Presentation Manager

A presentation manager provides an application program with a record-oriented interface to a presentation device. Its services are used by application programs, usually request initiators.

A forms manager is one type of presentation manager. Just as a database system supports operations to define, open, close, and access databases, a forms manager supports operations to define, enable, disable, and access forms. A form includes the definition of the fields (with different attributes) that make up the form. It also includes services to map the fields into device-independent application records, to perform data validation, and to perform data conversion to map fields onto device-specific frames.

One presentation manager is Digital's DECforms forms management product. The DECforms product is the first implementation of the ANSI/ISO Forms Interface Management Systems standard (CODASYL FIMS). [5]

Request Manager

A request manager provides services to authenticate the source of requests (a user and/or a presentation device), to submit requests, and to receive replies from the execution of requests. It supports such operations as send-request and receive-reply. Send-request must provide the identity of the source device, the identity of

By using presentation manager services, instead of directly accessing a presentation device, application programs become device independent.

the user who entered the request, the identity of the application program to be invoked, and must input data to the program.

A request manager can either pass the request directly to an application program, or it can store requests in a queue. In the latter case, another request manager can subsequently schedule the request by dequeuing the request and invoking an application program. The ACMS System Interface is an example of an existing request manager interface for direct requests. The ACMS Queued Transaction Initiator is an example of a request manager that schedules queued requests. [1]

Transaction Manager Architecture

DECdta components are tied together by the transaction abstraction. Transactions allow application programs, resource managers, request managers (indirectly through queue resource managers), and communication managers to interoperate reliably. Since transactions play an especially important role in the DECdta architecture, we describe the transaction management functions in more detail.

The DECdta architecture includes interfaces between transaction managers and by Digital's DECdtm distributed transaction manager. [2]

application programs, resource managers, and communication managers, as shown in Figure 3. It also includes a transaction manager protocol, whose messages are propagated by communication managers. This protocol is used

From a transaction manager's viewpoint, a transaction consists of

transaction demarcation operations, transaction execution operations, two-phase commit operations, and recovery operations.

- The transaction demarcation operations are issued by an

  application program to a transaction manager and include operations to start and either end or abort a transaction.
- Transaction execution

  operations are issued by resource managers and communication managers to a transaction manager. They include operations
  - For a resource

    manager or communication manager to join an existing transaction
  - For a communication manager to tell a transaction manager

    to start a new branch of a transaction that already exists at another node
- Two-phase commit operations are issued by a transaction manager to resource managers, communication managers, and through

  manager to prepare, commit, or abort a transaction
  - For a resource manager or communication manager to tell a transaction manager whether it has prepared, committed, or aborted a transaction
  - For a communication manager to ask a transaction manager to prepare, commit, or abort a transaction
  - For a transaction manager to tell a communication manager whether it has prepared, committed, or aborted a transaction
- Recovery operations are issued by a resource manager to its transaction manager to determine the state of a transaction (i.e., committed or aborted).

In response to a start operation invoked by an application program, the transaction manager dispenses a unique transaction identifier for the transaction. The transaction manager that processes the start operation is

communication managers to other transaction managers, and vice-versa. They include operations

- For a transaction manager to ask a resource manager or communication

that transaction's home transaction manager.

When an application program invokes an operation supported by a resource manager, the resource manager must find out the transaction identifier of the application program's

transaction. This can happen in different ways. For example, the application program may tag the operation with the transaction identifier, or the resource manager may look up the transaction identifier in the application program's context. When a resource manager receives its first operation on behalf of a transaction, T, it must join T, meaning that it must tell a transaction manager that it is a subordinate for T. Alternatively, the DECdta architecture supports a model in which a resource manager may ask to be joined automatically to all transactions managed by its transaction manager, rather than asking to join each transaction separately.

A transaction, T, spreads from one node, Node 1, to another node, Node 2, by sending a message (through a communication manager) from an application program that is executing T at Node 1 to an application program at Node 2. When T sends a message from Node 1 to Node 2 for the first time, the communication managers at Node 1 and Node 2 must perform branch registration. This function may be performed automatically case, the result is as follows: the communication manager at Node 1 becomes the subordinate of the transaction manager at Node 1 for T and the superior of the communication manager at Node 2 for T; and the communication manager at Node 2 becomes the superior of the transaction manager at Node 2 for T. This arrangement allows the commit protocol between transaction managers to be propagated properly by communication managers.

After the transaction is done with its application work, the application program that started transaction T may invoke an "end" operation at the home transaction manager to commit T. This causes the home transaction manager to ask its subordinate resource managers and communication managers to try to commit T. The transaction manager does this by using a two-phase commit protocol. The protocol ensures that either all subordinate resource managers commit the transaction or they all abort the transaction.

In phase 1, the home transaction manager asks its subordinates for T to prepare T. A subordinate prepares T by doing what is necessary to guarantee

by the communication managers. Or, it may be done manually by the application program, which tells the communication managers at Node 1 and Node 2 that the transaction has spread to Node 2. In either that it can either commit T or abort T if asked to do so by its superior; this guarantee is valid even if it fails immediately after becoming prepared. To prepare T,

- Each subordinate for T recursively propagates the prepare request to its subordinates for T
- Each resource manager subordinate writes all of T's updates to stable storage
- Each resource manager and transaction manager subordinate writes a prepare-record to stable storage

A subordinate for T replies with a "yes" vote if and when it has completed its stable writes and all of its subordinates for T have voted "yes"; otherwise, it votes "no." If any subordinate for T does not acknowledge the request to prepare within the timeout period, then the home transaction manager aborts T; the effect is the same as issuing an abort operation.

In phase 2, when the home transaction manager has received "yes" votes from all of its subordinates for T, it decides to commit T. It writes a commit record for T to stable storage and tells its subordinates for T to commit T. Each subordinate for T writes a commit record for T to stable storage and recursively propagates the commit request to for T. When the home transaction manager receives acknowledgments from all of its subordinates for T, the transaction commitment is complete.

To recover from a failure, all resource managers that participated in a transaction must examine their logs on stable storage to determine what to do. If the log contains a commit or abort record for T, then T completed. No action is required. If the log contains no prepare, commit, or abort record for T, then T was active. T must be aborted. If the log contains a prepare record for T, but no commit or abort record for T, T was between phases 1 and 2. The resource manager must ask its superior transaction manager whether to commit or abort the transaction.

An inherent problem in all two-phase commit protocols is that a resource manager is blocked between phases 1 and 2, that is, after voting "yes" and before receiving the commit or abort decision. It cannot commit or abort the transaction until the transaction manager tells it which to do. If its transaction manager fails, the resource manager may be

its subordinates for T. A subordinate for T replies with an acknowledgment if and when it has committed the transaction (in the case of a resource manager subordinate) and has received acknowledgments from all subordinates	blocked indefinitely, until either the transaction manager recovers or an external agent, such as a system manager, steps in to tell the resource manager whether to commit or abort.

A transaction T may spontaneously abort due to system errors at any time during its execution. Or, an application program (prior to completing its work) or a resource manager (prior to voting "yes") may tell its transaction manager to abort T. In either case, the transaction manager then tells all of its subordinates for T to undo the effects of T's resource manager operations. Subordinate resource managers abort T, and subordinate communication managers recursively propagate the abort request to their subordinates for T.

The two-phase commit protocol is optimized for those cases in which the number of messages exchanged can be reduced below that of the general case (e.g., if there is only one subordinate resource manager, if a resource manager did not modify resources, or if the presumed-abort protocol was used to save acknowledgments). [6]

Summary

We have presented an overview of the DECdta architecture. As part of this overview, we model will be made public via product offerings or architecture publications.

Acknowledgments

This architecture grew from discussions with many colleagues. We thank them all for their help, especially Dieter Gawlick, Bill Laing, Dave Lomet, Bruce Mann, Barry Rubinson, Diogenes Torres, and the TP architecture group, including Edward Braginsky, Tony DellaFera, George Gajnak, Per Gyllstrom, and Yoav Raz.

References

1. T. Speer and M. Storm, "Digital's TP Monitors," Digital Technical Journal, vol. 3, no. 1 (Winter 1991, this issue): 18-32.
2. J. Johnson, W. Laing, and R. Landau, "Transaction Management Support in the VMS Operating System Kernel," Digital Technical Journal, vol. 3, no. 1 (Winter 1991, this issue): 33-44.
3. P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems (Reading, MA: Addison-

introduced the components and explained the function of each interface. We also described the DECdta transaction management architecture in some detail. Over time, many interfaces of the DECdta

Wesley, 1987.

4. P. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," Proceedings 1990 ACM SIGMOD Conference on Management of Data (May 1990).

5. FIMS Journal of Development (Norfolk, VA: CODASYL FIMS Committee, July 1990).

6. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," ACM Transactions on Database Systems, vol. 11, no. 4 (December 1986).