Distrib. Syst. Engng **5** (1998) 1–11. Printed in the UK PII: S0967-1846(98)88318-6

Context management and its applications to distributed transactions*

Ajay D Kshemkalyani†‖, George Samaras‡¶ and Andrew Citron§+

† Department of Electrical & Computer Engineering and Computer Science, PO Box 210030, University of Cincinnati, Cincinnati, OH 45221-0030, USA
‡ Department of Computer Science, University of Cyprus, PO Box 537, Kallipoleos 75, 1678 Nicosia, Cyprus
§ IBM Corporation, PO Box 12195, Research Triangle Park, NC 27709, USA

Received 9 December 1996

Abstract. An emerging paradigm that handles multiple loci of control in a system allows multiple program threads to work on the same task, each thread to work on a different task, or a thread to work on multiple tasks for greater design flexibility or due to system constraints such as real-time demands and a high load on tasking. We use the definition of *context* to capture the notion of logical locus of control. The *context* of the work being currently executed must be identifiable uniquely by the application, the resource managers and the transaction manager because each *context* represents different work. in this paper, we define context management by defining a local context manager and its user interface. We then show why the notion of *context* is required to solve the problems that arise in local and distributed transaction processing due to the emerging paradigm. We present solutions to these problems in transaction processing using the proposed context management.

1. Introduction

Currently, operating systems that handle multiple applications provide a separate operating system level locus of execution for each individual application program. The locus of execution is the process for single-threaded processes provided by operating systems such as DOS and VM, and it is the thread♯ for multithreaded processes provided by operating systems such as UNIX††, OS/2‡‡ [6, 13], Windows NT§§ and Windows 95§§ [9]. Operating systems support the client–server model of computing by dispatching a separate server process or thread to handle a new request from the clients.

There are two recent trends which indicate that the existing support provided by operating systems is unsuitable for a range of application programs. First, as applications grow in number and get more distributed, the number of applications a server can support becomes limited by the operating system constraints such as the number of processes/threads allowed within the system. Secondly, as applications become more numerous and response times become critical for real-time systems, the servers cannot afford the overhead of process start-up and switching, or forking and dispatching, and the overhead of locking mechanisms for access to shared tables for each new application. A new processing paradigm is now evolving to overcome the above problems and to provide more flexibility to distribute tasks across processes/threads. The emerging paradigm is as follows. A process or thread can concurrently support multiple client applications, and a server application can be distributed across multiple processes and/or threads. This paper proposes how this new paradigm can be supported, and discusses its interaction with transaction processing. Note that this paradigm is applicable to all operating systems, and can be exploited by all applications. We focus on distributed transaction processing as an application because it represents an important and growing class of applications, and it was our involvement in distributed transaction processing that triggered this work.

In the paradigm outlined above, each initiated transaction supported by a thread(s) is explicitly associated

* This paper is a revised and expanded version of a paper by the same title, by G Samaras, A Kshemkalyani and A Citron, that appeared in *Proc. IEEE Conf. Distributed Computing Systems (May 1996)* pp 683–91.

‖ E-mail address: ajayk@ececs.uc.edu
¶ E-mail address: cssamara@turing.cs.ucy.ac.cy
+ E-mail address: citron@vnet.ibm.com

♯ For commit processing of a transaction, the locus of execution is still the process, not the thread. This is a drawback of existing transaction processing design for multithreaded systems.

†† UNIX is a registered trademark in the United States and other countries licenced exclusively through X/Open Company Limited.

‡‡ OS/2 is a trademark of the IBM Corporation.

§§ Windows, Windows NT and Windows 95 are trademarks of Microsoft Corporation.

with a *context*. Thus, a context becomes a logical locus of control and represents a transaction and its associated resources. The above notion of context is similar to X/Open's notion of 'thread-of-control' [19]. With the new processing paradigm, multiple transactions can be associated with a thread, representing multiple contexts per thread. However, at any instant, only one of these contexts will be active. The application and the system should be able to specify and determine: (1) the context currently being worked on by the thread, and (2) all the resources associated with the processing of any context. There is an explicit need to coordinate the contexts within and across threads and processes, and to coordinate access to resources by multiple contexts (within a thread, across threads, and across processes). This is achieved through a *context manager (CM)* mechanism and its associated user interface defined in this paper.

The contribution of this paper is that we define context, the CM and its application programming interface (API), and show how context is used for local and distributed transaction processing. We highlight the role of context in a multithreaded, real-time, operating system environment with a high load on tasking, provide different practical styles of transaction management using context management, and show how to use context management to solve deadlocks, protocol violations, and loopbacks in distributed transaction processing, as well as to reconcile chained and unchained support for distributed transaction processing. We have implemented a prototype of the CM for use with the SNA LU6.2 Syncpoint Services [8]. The VM operating system Shared File System has been enhanced to provide a version of the context management support described here.

This paper is organized as follows. Section 2 describes the system model. Section 3 examines the requirements for a new notion of context, and defines and describes the operation and usage of context, along with a new CM user interface. Two examples of the use of context management in a multithreaded environment are given. Section 4 describes the problems that arise in distributed transaction processing using the new paradigm, namely assigning multiple transactions to multiple threads. It then shows how to use context management to solve the problems. Section 5 shows how context management is used to provide the functionality of unchained transaction processing to those systems that provide only chained transaction processing. Section 6 concludes.

2. System model

A distributed system consists of a set of computing nodes linked by a communications network. The nodes of the system cooperate with each other in order to process distributed computations. For the purpose of cooperation, the nodes communicate by exchanging messages via the communications network using a dedicated end-to-end logical connection called a *conversation*.

A multiprogramming/multiprocessing operating system runs at each node. A process which is an executing program has a single address space and a single thread of control for the program [18]. The state information for the process consists of page tables, swap images, file descriptors, outstanding I/O requests, and saved register values. Multiple programs are handled by maintaining and switching between processes. If threads or lightweight processes are supported by the operating system, then the threads of a process concurrently execute within the same address space. Each thread uses a separate program counter, a stack of activation records, and a control block which contains information necessary for thread management. Most of the information that is part of a process is shared by all the threads executing in the same address space. This reduces the overhead in creating and maintaining information, and the information that has to be saved when switching between threads of the same process.

A *transaction* program (TP) is any program that requires that its effects are *atomic*, i.e. either all the effects of the program persist or none persist, whether or not failures occur [5, 11]. Each TP has a unique identifier denoted TRANID. Atomicity of a transaction is guaranteed by a *commit protocol*. A *distributed transaction* is a transaction for which the program code and/or data accessed is distributed on different nodes in the networked system [5]. The execution of a distributed transaction requires a *distributed commit protocol* to ensure atomicity. An *application*, also known as an application program, consists of several (possibly distributed) transactions that may be interleaved with nontransactional program code. An application usually denotes a user-level program. An application may be executed by single or multiple processes or threads. In addition, the application may be distributed and may be a server application or a client application. We use the terminology 'server application or server' and 'client application or client' when the nature of the application needs to be emphasized.

Once the code of a transaction is executed, the application instructs the *transaction manager (TM)* of its node (site) to initiate and coordinate the commit protocol. The 'logical locus of control' from which the application issues the commit command is the entity that identifies to the TM the transaction to be committed. At each node, the local *resource managers (RMs)*, such as database and file managers, and *communication resource managers (CRMs)* participate in the commit protocol. The RMs/CRMs commit only those resources that are associated with the current 'logical locus of control' and transaction. The CRM embodies the communication protocol and provides a local view of the remote processes and remote TMs. The TMs that participate in the commit processing include one *coordinator* and one or more *subordinates*. The coordinator coordinates the final outcome of the commit processing by issuing a commit or abort, that is propagated to all subordinates. Subordinate TMs propagate the decision to their subordinate TMs or local RMs. The commit operation employs the well known *two-phase commit (2PC)* protocol [5, 12, 17], sketched next.

The 2PC protocol ensures that all participants commit if and only if all can commit successfully. The 2PC consists of two serial phases: the *voting* phase and the *decision* phase. During the voting phase, the coordinator

asks all the other participants to prepare to commit (using the prepare flow), and they reply YES or NO. During the decision phase, the coordinator propagates the outcome of the transaction to all participants (using the commit/abort flows): if all participants voted YES, the commit outcome is propagated; if any participant voted NO, the abort outcome is propagated to those participants that voted YES. Each participant in the transaction commits or aborts the effects of the transaction based on the outcome propagated by the coordinator and acknowledges the completion to the coordinator. The entire 2PC is triggered by issuing the COMMIT command at the coordinator. However, an individual branch of the spanning tree rooted at the coordinator can be processed for the voting phase by issuing a Prepare_For_Syncpt call along it [8].

3. Context

Definition 1. A *context* is a logical locus of control that is unique and local to the operating system at a node. The context is shared by the application, TM and RMs to manage their resources, and relate their resources to the resources owned by other RMs within the system. A context represents a grouping of resources within the system, needed to perform a particular function in a logical locus of control or to show the inter-relationship between diverse resources.

Each dispatchable entity is associated with a context. The notion of context is supported by the system by way of defining, storing, and recognizing a context identifier, discussed in section 3.1. The application, TM, and RMs share this notion of context. The usage of context is user-defined; the extent to which a context is shared by the operating system dispatchable entities (processes or threads), the address spaces, and data spaces, is controlled by the application design.

The *scope* of a task defines the set of resources required for the task. When an application issues commit, the scope defines the set of related local resources that participate in the commit processing. The notion of context helps group the set of resources more flexibly.

3.1. Requirements

There are several emerging trends that require the notion of context.

(I) Currently, a process or thread is associated with at most a single logical locus of control. This paradigm is proving inadequate for some applications because:

- the client–server paradigm requires an executing server application to accept multiple incoming requests. For example, the asynchronous RPC style of distributed programming uses this model [1]. The following difficulties arise if a different thread/process is used for each request. (a) First, as applications grow in number and become more distributed, the number of transactions a server can support becomes limited by the operating system constraints such as the number of processes/threads allowed within the system. For example, OS/2 can support 4096 threads. (b) Second, as response time becomes critical for real-time systems, the servers cannot afford the overhead of process start-up, switching, dispatching or forking, locking mechanisms for accessing shared tables, and extra storage for each new transaction. A thread should be able to support multiple transactions and temporarily suspend work on a long-running request to process work for another request. This minimizes demand on operating system resources, while allowing greater parallelism in servicing requests. The result is better response time and/or better throughput.
- A message routing program in a large database system acts as a router based on the content of the message. A database system typically uses long-lived programs that handle transactions from more than one end-user or transaction at a time and that can activate other conversations based on the database activity and the input.

In both of these cases, the same thread or process should accept the various incoming routing requests (loci of execution) rather than have separate threads handle the various routing requests, for better efficiency. A new paradigm that allows a server process/thread to support multiple transactions is required.

(II) The notion of context provides useful functionality allowing process-oriented systems as well as thread-based systems the flexibility needed in today's complex and demanding environment. For operating systems that allow threads to be spawned or processes to be forked, it is desirable to allow a server application to divide the incoming requests however it chooses to. Some applications might want all threads to work on the same request. Other applications might want each thread to work on a different request or to work on multiple requests. A context management service needs to allow each thread to identify each of the contexts it is associated with. The context must be independent from the operating system's task dispatching mechanism.

The above requirements express the emergence of environments where:

- a server process or thread can accept requests from different end users, and is allowed to suspend work on one request to work on a different request, or
- multiple server processes/threads are working on related work representing the same context of the application.

The application needs a way to inform the TM and the RMs which task the application is working on at a particular time, and the TMs and RMs need to coordinate to have a common understanding of which context is currently under execution, for the following reasons.

- To group together logically related work and separate logically unrelated work.
- Each request is likely to be a part of a different atomic transaction. The work a server process does on behalf of one transaction must commit or abort independently from other unrelated work that the server was handling.
- The security authorization of each request can be different. An application must make sure that the system's security manager and other RMs cooperate to ensure that the access granted at any particular instant is proper for the end user application that is currently being worked on.

For the above, a thread_id or process_id does not suffice to identify the logical locus of control; rather a *context_id* is required. The notion of context allows the management of multiple logical instances of the same transaction program within a single process or single thread, as well as the management of a single instance of a transaction program across several processes/threads. Our context management scheme thus provides the much greater flexibility of allowing the coordination of context not just between the TM and transactional RMs, but between all system components including the TM, transactional RMs, nontransactional RMs, and the applications. With context management, the commit issued by the application applies to the context from which it was issued; not to the thread that issued it. Currently, for commit processing, the locus of control is confined to the process level and is not allowed to the thread level, even though multithreaded environments are common. This is an existing shortcoming of transaction processing design for multithreaded systems. Our solution of context can accommodate any future extensions of the scope of a commit to the thread or even to a context within a thread or across threads.

3.2. Context management

Context management is a local application-support mechanism that permits applications to manage logically separate pieces of work within or across processes or threads [2]. When an application uses context management, the CM keeps track of the various contexts, allows the application to create and set a context for work, and allows an application to switch contexts when appropriate. The CM shares the notion of contexts with the RMs and the application.

A new incoming conversation is assigned a new context and the program's current context is set to the new context. When a new outgoing conversation is allocated to a partner program, the conversation is assigned the current context of the program.

While context management provides additional functions such as flexibility and avoidance of process switching to the application program, it also imposes an extra burden on the application. The application has to keep track of the progress done in each context, and switch contexts to meaningfully exploit the features of context management. An application can perform context management by exploiting the functions provided by the CM using a suite of calls, described subsequently.

A server process that performs context management has special responsibilities [3]. The server process must correctly indicate the end user context on behalf of which it is working. When many threads are acting on behalf of the same context, the server process must make sure that the work of all the threads is completed before kicking off 2PC processing. Similarly, when a thread does work on behalf of multiple independent contexts, it must be ensured that all the work in the relevant context is completed before kicking off 2PC processing.

The notion of context provides a logical separation of work done by an application; each logically separate piece of work is done in a separate context. Context is local to a system node. Different tasks of a distributed transaction, that each represent different loci of control and are executed at different nodes, have a different context even though they belong to the same transaction. Furthermore, loosely coupled executables within a transaction that are executed on the same physical node may have separate contexts. Specific examples of such loosely coupled contexts that arise in distributed transaction processing are discussed in section 4. Clearly, when an application program is involved in multiple transactions at a time, each transaction is done in a separate context at the application. The same context at a node may be involved in multiple transactions only sequentially. The application assumes the responsibility of keeping track of its various contexts, coordinating the data spaces of the various contexts, and switching between contexts.

Context manager associations. The CM administers the contexts independently of the operating system's task dispatching mechanism. At each node, the context is identified by a *context_id*; a CM maintains a context table that stores the following information per context_id:

- context_id, unique to a node
- (thread_id, process_id), unique to a node
- a boolean indicating whether this context is currently being worked on by this (thread_id, process_id).

The TM maintains the association between the TRANID and the context_id, the RM maintains the association between the resources it manages and the context_id, i.e. the database resource manager keeps the association between the database changes, lock information, and the context_id, the communications resource manager keeps the association between the connection_id and the context_id, the security manager keeps the association between the validated user_id and context_id; likewise for the managers of other subsystems. The various RMs can query the security manager for the user_id of the current context.

We define a context management interface that: (1) implicitly externalizes the creation, coordination and deletion of contexts, and (2) allows the application and managers of various subsystems such as the TM, RM, and security manager to associate their resources with a context and coordinate the usage of all resources related to a particular context.

3.2.1. Context management calls. In an environment where there is no one-to-one association (but rather a many–many association) between threads and contexts, explicit context management calls are required to control the association between threads and contexts. This is particularly important for transaction processing because the TM and RMs must identify the TRANID on a context basis and not on a thread basis. The thread_id, TRANID, accounting and security information are all associated with a context.

Transaction management in a multithreaded environment. When a thread (more generally, a *locus of execution*) is created, it is assigned a context—the context can be a new one or an inherited one. We describe three transaction program design patterns that use threads [2]. The design patterns use context management calls to start and manage the threads within the context management framework. Typically, server applications would follow these design patterns.

The following auxiliary calls are needed to support threaded applications, as well as applications that are single-threaded but process more than one context at a time.

• EXTRACT_CURRENT_CONTEXT(context_id): This function allows the TM, RM, or the application to find out which context is currently active. The context_id is a return parameter.

• SET_CONTEXT(context_id): This function allows an application that can run for more than one context (e.g. a program that processes many independent incoming requests) to inform the system which context the application wants to work on.

• START_NEW_CONTEXT(context_id): This function allows the system scheduler and applications to start new work that is independent of other work they are processing. The context_id is a return parameter.

Program design pattern 1. A server receives new work and kicks off a thread to handle the new work. A new context is implicitly created using START_NEW_CONTEXT whenever new work is accepted. This is a typical approach for an RPC server, or an OS/2 LU6.2 TP that issues RECEIVE_ALLOCATE and then waits for the next incoming work. The context management function to support this is as follows.

• START_THREAD_AND_HANDOFF_CONTEXT(context_id): This function starts a new thread and disassociates the main (old) thread from the newly created context. If no parameter is specified, the context_id that is handed off is extracted from the environment of the current context using EXTRACT_CURRENT_CONTEXT. The forked thread is associated with the new context.

Program design pattern 2. A server kicks off a thread but the forking thread continues to work on the same context. This is typical of an application that can take advantage of the parallelism that lightweight threads provide. The CM function associated with this is as follows.

• START_THREAD_AND_SHARE_CONTEXT(context_id): This function starts a new operating system thread. If no parameter is specified, the context_id to be shared is extracted from the environment of the current context using EXTRACT_CURRENT_CONTEXT. Any one of the threads can initiate the commit operation. The commit operation affects all resources allocated to this context. It is up to the application's design to ensure that all threads are ready for commitment. If some threads are not ready, the commit call may return a state-check, or it may backout, or accidentally commit work in-progress. A simple approach is to have the main thread issue COMMIT, after all forked threads report they are ready (using an OS wait/post mechanism for example).

Program design pattern 3. The main thread receives a new work request, and then instead of forking a new thread, it hands the work to an existing thread. For performance reasons, it is better to avoid creating a new thread; using prestarted threads that are waiting for work is faster.

• HANDOFF_CONTEXT(context_id): This function gives exclusive ownership of a context to an existing thread and posts the thread to inform it a new context is available. If a context is not explicitly specified as a parameter, then the current context, identified using EXTRACT_CURRENT_CONTEXT, is handed off. Any one of the existing threads that are waiting for work using GET_NEW_CONTEXT (see below) is given the context.

• SHARE_CONTEXT(context_id): This function permits shared ownership of a context with an existing thread and posts the thread to inform it a new context is available. If a context is not explicitly specified as a parameter, then the current context, identified using EXTRACT_CURRENT_CONTEXT, is shared. Any one of the existing threads that are waiting for work using GET_NEW_CONTEXT (see below) gets to share the context.

• THREAD_DONE_WITH_CONTEXT(context_id): This function allows a thread to disassociate itself with a context and get ready to be involved in a new context. If no other thread is associated with the context, an implicit commit is attempted. If a context is not explicitly specified as a parameter, then the thread disassociates itself from the current context, identified using EXTRACT_CURRENT_CONTEXT.

• GET_NEW_CONTEXT(context_id): This function allows a thread to wait for a coordinator thread to issue HANDOFF_CONTEXT or SHARE_CONTEXT. Blocking and nonblocking flavours are useful. The context_id is a return parameter.

GET_NEW_CONTEXT and SET_CONTEXT are the two ways in which a thread can explicitly change the context to get involved in existing work. START_NEW_CONTEXT creates a new context for new work. HANDOFF_CONTEXT, THREAD_DONE_WITH_CONTEXT and START_THREAD_AND_HANDOFF_CONTEXT disassociate the issuing thread from a context.

3.2.2. Examples

Example 1. Thread handles multiple contexts. Table 1 gives example 1 in which a single thread at the server switches context to handle requests from two clients.

Example 2. Threads handle a transaction using different/same contexts. Example 2 given in table 2 deals with one client and one server. The server has prestarted a number of threads. This design can be used to optimize the performance of a database client/server application that issues multiple SQL queries that open

Table 1. Example 1. Thread handles multiple contexts.

Time	Client 1	Server	Client 2
1	BEGIN TRANSACTION request		
		START_NEW_CONTEXT(C1) BEGIN TRANSACTION begin work for client 1; send reply	
	get reply		
2			BEGIN TRANSACTION request
		START_NEW_CONTEXT(C2) BEGIN TRANSACTION begin work for client 2; send reply	
			get reply
3	request		
		SET_CONTEXT(C1) (server switch to Client 1's context C1) do some work for context 1; send reply	
	get reply		
4			request
		SET_CONTEXT(C2) (server switch to Client 2's context C2) do some work for context 2; send reply	
			get reply
5	COMMIT (client 1 commits)		
		SET_CONTEXT(C1) Change context to C1. Issue COMMIT to TM. TM commits all work associated with current context. (TM queries CM to get current context before processing commit. It then associates current context with the TRANID and orders all RMs to commit work associated with that TRANID and context.)	
6			COMMIT (client 2 commits)
		SET_CONTEXT(C2) change context to C2. Issue COMMIT to TM. TM commits all work associated with current context. (TM interacts with CM as in previous step.)	

multiple cursors† and fetches data from each cursor [4]. A separate connection is used for each cursor to allow a fetch to be done on each open cursor independent of the data flowing on other connections. The server hands off work to the threads which are already initialized, and are waiting for incoming work. The waiting threads are not involved in work for any context. Once the thread is given the context and the connection associated with that context, the thread has exclusive use of the connection. In the example, multiple threads work on a single transaction using different contexts. There are two independent connections created at the server. By default, each connection starts a new context.

For performance reasons, it is better to have all the threads using the same context. Even though each connection has its own context, the server can understand, through application-specific logic, that the requests from the single client that are coming over different conversations pertain to related work. So the server could choose to hand the same context to each of the threads. The example would change as follows. In step 1, the server would issue SHARE_CONTEXT instead of HANDOFF_CONTEXT. In step 2, the server would simply issue SHARE_CONTEXT instead of START_NEW_CONTEXT and HANDOFF_CONTEXT.

† Each cursor is a pointer to a row within each query result.

3.3. Interfaces between CM, TM and RMs

The TM and protected RMs normally associate a thread with a TRANID. Thread_id and TRANID, along with accounting and security information are the associations maintained for a 'context'. In an environment where an application can start a thread to take care of part or all of a transaction, the TM and the protected RMs need to share the current context of the thread that is executing on the protected resource. The mechanism that allows the TM and

Table 2. Example 2. Multithreaded server application using a different thread for each context. Each context is part of the same distributed transaction.

Time	Client 1	Server
0		Two threads are prestarted and are waiting to do work on behalf of a context by issuing to the main thread GET_NEW_CONTEXT(context_id)
1	BEGIN TRANSACTION SQL request OPEN_CURSOR A (one protected connection with the server)	START_NEW_CONTEXT(C1) BEGIN TRANSACTION HANDOFF_CONTEXT(C1) (This satisfies thread_one's GET_NEW_CONTEXT. Thread_one is now processing SQL request.)
2	SQL request OPEN_CURSOR B (one protected connection with the server) fetch from A fetch from B	START_NEW_CONTEXT(C2) HANDOFF_CONTEXT(C2) (This satisfies thread_two's GET_NEW_CONTEXT. Thread_two is now processing SQL request.)
3	COMMIT (If both connections are protected, the local TM will initiate commit processing on both connections.)	The two server threads will receive the commit message. Each thread issues COMMIT to TM. TM commits all work associated with current context of each thread. TM queries CM to get current context before processing commit. It then associates current context of each thread with TRANID and requests all RMs to commit work associated with that TRANID and context.

the RMs on a local system to share a common context_id is a matter of implementation. Two design choices we considered are given below.

Design 1. Each time an RM, including the security RM, is invoked, it can query the current context using the CM's EXTRACT_CURRENT_CONTEXT. The RM then looks up its own tables to determine if it is already involved in work for this current context. If it is not already involved in work for the current context, the RM adds an entry to its internal tables. The entry includes the context_id, and whatever other information the particular RM needs. Section 3.2 described how the context_id is used by the various RMs.

Design 2. The CM provides a broadcast mechanism. With this approach, the CM broadcasts the current context_id to all 'interested' RMs. The broadcast occurs each time an OS dispatchable unit (thread or process) changes the context it is working on. The context change occurs because the application issues a SET_CONTEXT call to the CM. The RMs do internal housekeeping when notified that the active context has switched from one context_id to another. The switch does not imply that the work has committed or aborted, but rather that a context has temporarily suspended execution. This usage of SET_CONTEXT is akin to X/OPEN's *xa_end(suspend)*.

The mechanism for determining which RM is 'interested' in the broadcast is also an implementation issue. There can be either dynamic registration where an RM calls the CM to request to be notified whenever the context associated with a thread or process changes, or the CM can support automatic inclusion of RMs based on system definition.

The choice of query versus broadcast is a performance issue. In a particular environment, queries might entail less overhead than broadcasts. In the prototype developed for OS/2, a query mechanism was used. The CM maintained a table with process_id, thread_id, context_id, and a flag for the active context_id per thread. On receiving a query, the CM could reply with the context currently associated with the thread/process of interest to the query.

3.4. X/Open's thread of control API and the context management API

We have seen that an important part of context management is the ability to allow the application, TM, RM, and other resources to share contexts and identify the current context. The interface between the TM and RM, as defined by X/Open [19], allows the *xa_end(suspend/resume)* call from the TM to the RM to inform the RM about which thread of control is currently performing work. However, the X/Open specification does not address how the TM identifies the active thread of control because X/Open does not provide any handle or API to identify the current thread of control. X/Open has recognized the importance of the drawback and in a recent X/OPEN request for comments (RFC) [20], the need to address the problem was clearly identified.

Our proposed context management mechanism provides the solution to this problem.

Sections 4 and 5 describe how context management can solve complex issues in distributed transaction processing that arise due to the way multiple transactions are assigned across processes/threads.

4. Distributed transaction management

Each new incoming request accepted by a transaction program (TP) is handled by a new instance of the TP. Multiple such TP instances can in some cases share the same TRANID but may be in the same or different thread or process in the proposed processing paradigm. Each TP instance is a different locus of control and context is necessary to identify it. We show that the TRANID, and thread_id or process_id are not enough to identify the TP instance. If context_id is not used, there is a possibility of deadlocks [10] or protocol violations during commit processing, due to the combination of the new processing paradigm and 'loopback', discussed next. Note that these problems occur even if the commit protocols are nonblocking and cope with failures during their execution. The problems are more obvious in the peer-to-peer communication model where the commit can be initiated by any partner in the transaction tree and more than one transaction is in progress at the same time. The notion of context plays an important role in solving these problems cleanly. We will discuss the problems in transaction processing that exist without using context, and the solutions offered by context management. We also review the use of context management in reconciling communication protocol support between chained and unchained transactions [16].

4.1. Loopback

Loopback is a system state in which a transaction reappears at a node that is already involved in the same transaction [8]. Multiple contexts in one commit tree present the same TRANID to the shared RMs. The loopback can be direct when a client invokes a server that happens to reside on the same node. Loopback can be indirect, when a server, say X, is invoked by a client on a different node, which in turn is a cascaded server for a client on the same node as the server X. Indirect loopback can also occur when two different servers on the same node are invoked as part of the same transaction. Figure 1 illustrates a loopback involving three partner programs X, Y, and Z.

Currently, when a loopback occurs, a process or thread is dispatched to handle the second occurrence of the transaction. The two loci of execution have the same TRANID but can be differentiated by using the process_id or thread_id. In 2PC, the TM needs to determine which resource needs to be sent prepare [8]. (Note that the prepare flows correspond to the TP-prepare service of OSI TP [14], and in the X/Open model, they are triggered by *xa_prepare* and *ax_prepare* [19].) A particular resource associated with the locus of control can be identified by process_id (from which TRANID can be deduced) and

Figure 1. Example of loopback. TP_Z and TP_Z′ share the same TRANID, process_id and thread_id.

connection_id (the 'leg' identifier, known in OSI TP as the branch_id and in LU6.2 as the conversation correlator). This is sufficient even if a program has a conversation with another program on its own node whereby both branches of the connection/conversation have the same TRANID and connection_id. In the new processing paradigm, the TM can identify a particular resource associated with the locus of control by context_id, from which TRANID can be deduced, and connection_id. If TRANID or thread_id were used instead of context_id, two loci of control would satisfy (TRANID, connection_id) or (thread_id, connection_id), respectively, when the two loci of execution were allocated to the same thread and had a connection with each other. However, when the context_id is used with the connection_id, the context_id uniquely identifies the locus of execution, with respect to which the connection_id is used to determine which resource(s) should be sent prepare.

The above occurrence of loopback arises in the new paradigm when a peer-to-peer communication model, such as modern SNA (APPN and APPC), is assumed [7]. The peers in the transaction are considered to be 'loosely coupled' [19]. In the peer-to-peer model, the commit initiator can be different from the dialogue initiator, and may reside on the same system. In this case, the (TRANID, connection_id) pair is not sufficient to identify the dialogue on which the prepare or backout should be sent. But in the hierarchical communication model (such as OSI TP [14] and RPC [1], whether blocking or nonblocking), only the dialogue initiator can initiate commit processing or send prepare; a TRANID, which can be deduced from process_id or thread_id, and connection_id are sufficient for the TM to determine the branch on which the CRM should send the prepare. However, context is still needed for other requirements presented in section 3.1.

4.2. Protocol violations and deadlocks

If the scope of the commit is the process and the TRANID and process_id are used to identify the different resources, protocol violations and deadlock [10] occur. If context management is used and the commit scope is the context (using the context_id), such problems do not occur. Consider the configuration shown in figure 1. The distributed transaction involves three partner programs, X, Y and Z. An instance of program P is denoted by TP_P. TP_X has invoked servers TP_Y and TP_Z. The arrows on the lines indicate the direction of invocation of TPs. TP_Y has in turn invoked TP_Z (which is really instance

TP_Z′ that is distinct from TP_Z that was started before it). TP_Z and TP_Z′ share the same TRANID, process_id, and thread_id, and hence the TM cannot distinguish between them. The commit tree is not a spanning tree if its nodes are defined by the TRANID, process_id, and thread_id.

Example 1 of protocol violations/deadlocks. (1) TP_X initiates the commit process. Both the conversation resources representing the connection to TP_Y and TP_Z belong to the current context of TP_X and the prepare message [7] is sent to both TP_Y and TP_Z. (2) In turn, the TM of TP_Y sends prepare to TP_Z′. However, TP_Z and TP_Z′ share the same TRANID, process_id, and thread_id, and hence cannot be distinguished by the TM handling them. The TM of TP_Z receives two prepares for the same transaction, which is interpreted as a protocol violation and results in backing out the transaction.

Using context management, the two incoming requests from client TP_X and client TP_Y result in the creation of two separate contexts (with the same TRANID and possibly same process_id and same connection_id) for TP_Z and TP_Z′, respectively. The commit scope is now the context and the commit tree is now represented by a tree of contexts. This is now a spanning tree. When TP_X initiates the 2PC, the prepare for partner TP_Z is for the context representing the connection from TP_X to TP_Z. The other prepare that arrives from TP_Y is for the context that represents the connection from TP_Y to TP_Z′. Based on context, the TM distinguishes between TP_Z and TP_Z′ and there are no problems in the 2PC.

Example 2 of protocol violations/deadlocks. (1) TP_X issues Prepare_For_Syncpt [7] on the branch to TP_Y. (2) TP_Y then issues the same call to TP_Z′. (3) The TM cannot distinguish between TP_Z and TP_Z′ and in an effort to prepare that branch of the tree identified by (TP_X,TP_Y) and (TP_Y,TP_Z′), the TM at Z propagates the prepare along (TP_Z,TP_X). (4) The TM at TP_X will detect a protocol violation in the hierarchical communication model [19], or a deadlock will occur in the peer-to-peer communication model as follows. TP_X will not reply to the prepare request of TP_Z; TP_Z′ (which the TM cannot distinguish from TP_Z) will not reply to the prepare request of TP_Y; TP_Y will not reply to the prepare request of TP_X.

However, if context were used, TP_Z′ would never have sent a prepare to TP_X, and no problems would have occurred. Context is essential to keeping a spanning 2PC tree.

Example 3 of protocol violations/deadlocks. The 2PC protocol has been extensively optimized by reducing the number of messages and force log writes [8, 17]. One such well known optimization is the linear commit [5], otherwise known as the last-agent (LA) optimization. When a partner that initiates the 2PC protocol decides to use the LA optimization, it first chooses the agent that will act as the last agent, then prepares all the other subordinates and finally passes control to the LA (by sending it the YES vote).

(1) Let TP_X choose partner TP_Z as the LA. When TP_X initiates the commit processing, the TM of TP_X will send prepare only to TP_Y. (2) TP_Y will in turn send prepare to TP_Z′. (3) The TM handling TP_Z′ cannot distinguish between it and TP_Z if it does not use context. So it attempts to send prepare to TP_X along (TP_Z,TP_X) because it does not know TP_Z has been chosen as the LA of TP_X.

If the conversation between TP_X and TP_Z is half-duplex, TP_X has send control and TP_Z waits to receive send control in order to send prepare to TP_X. This wait is indefinite because TP_X is blocked waiting for the response to the initial prepare, TP_Y is blocked waiting for a response from TP_Z′, and TP_Z (which the TM at Z cannot distinguish from TP_Z′), is blocked waiting for send control from TP_X. If the conversation between TP_X and TP_Z is full-duplex and a peer-to-peer model is used, TP_Z sends the prepare to TP_X and waits for a reply from TP_X. But TP_X cannot receive the prepare because its TM is blocked waiting for the response to the initial prepare, TP_Y is blocked waiting for a response from TP_Z′, and TP_Z (which the TM at Z cannot distinguish from TP_Z′), is blocked waiting for a response from TP_X. Thus, there is deadlock. If a hierarchical model is used and conversations are full-duplex, a protocol violation is detected by the TM of TP_X. None of these problems would arise if context were used because the TM of TP_Z′ would not send a prepare to TP_X.

5. Reconciling support for chained and unchained transactions

Distributed transactions require transaction processing support either from their communications protocols or from protocols at some other layer. One of the earliest industrial-strength distributed transactional support systems was provided by SNA (LU6.1 and LU6.2) and the vast majority of legacy applications for transaction processing still run over SNA [8]. When a transaction commits in SNA, the involved resources assume they are automatically placed in the next transaction—leading to the notion of chained resources and chained transactional support. As OSI TP [14] began specifying the flows for transactional support, the notion of unchained transactional support as opposed to the chained nature of LU6.2 transactions gradually developed. Unchained transactional support and unchained resources connote that when a transaction commits, the involved resources assume they are not placed in the next transaction unless explicitly informed otherwise (by Begin_Transaction). OSI/TP provides both chained and unchained transactional support. This makes the coexistence of applications requiring both kinds of transaction processing support over an SNA LU6.2 network problematic and reduces the interoperability of communication protocols providing transactional support. We define chained and unchained transactions and show how context management can be used to efficiently provide unchained transactional support in those systems that provide only chained transactional support, as proposed in [16]. Adding chained transactional support to those systems

that have only unchained transactional support is given for completeness.

5.1. Chained and unchained transactions

We recently defined chained and unchained transactions based on the manner in which the application demarcates the boundaries of a transaction [16].

Unchained transactions. An application is said to follow the *unchained* paradigm if it fulfils the following two requirements.

- The application does protected work on behalf of a transaction only when it explicitly specifies the beginning (and end) of the transaction.
- The application can only do unprotected work, or work that is not part of the transaction, outside the boundaries of a transaction specified by Begin_Transaction and End_Transaction (or commit).

Chained transactions. An application is said to follow the *chained* paradigm if and only if the application enters a transaction implicitly when it indicates the end of the previous transaction. Any protected work done by the application is always part of a transaction.

Note that the chained transaction paradigm is a transaction processing mechanism and differs from the chained transaction model [15] that is an extended transaction model for structuring advanced database applications.

5.2. Enhancing unchained transactional support for chained applications

A chained application can be made to run in an environment with unchained transactional support without the services of the CM. The goal is to somehow force the unchained resources to always be in a transaction. When a chained application issues a commit, the TM processes the 2PC protocol and the application is implicitly placed in the next protected unit of work. However, the underlying protocol (CRM) is unchained and assumes that subsequent work is unprotected unless it is informed otherwise. The TM knows that the application is running in chained mode and has to explicitly send a Begin_Transaction(new TRANID) to the unchained partner through the CRM because the application will not issue the Begin_Transaction message. This action of the TM explicitly places the partner in the next unit of protected work. In this way, a chained application can run over an underlying protocol that offers only unchained support.

5.3. Enhancing chained transactional support for unchained applications

Once a transaction is committed, all chained resources are implicitly in the next transaction. Any work now performed by them is considered part of the next transaction. To emulate the unchained paradigm implies that we somehow need to force the chained resources to consider work that they perform as unprotected until the application issues a Begin_Transaction(TRANID) to these chained resources to include them in the next transaction. Context management can be used to temporarily change from a transactional context to a nontransactional context just as it can be used to change from the context of one transaction to that of another. Using the context to identify the commit scope of a transaction, an application that commits needs to be directed to change context to one not associated with the governing transaction, in order to perform out-of-a-transaction work. To apply this to a distributed transaction, after the coordinator commits and decides to perform work with a chained resource out of the current distributed transaction, it must first direct the chained resource/partner to switch context. This can be done explicitly or implicitly via the first message to that resource. Details of several mechanisms to achieve this are given in [16]. Thus, the use of context allows unchained transactions to run on systems that offer only chained transactional support.

6. Conclusions

Operating systems that support threads within a process need the notion of context to efficiently support the paradigm where a single thread can be concurrently associated with several transactions or where several threads work on the same transaction. This paradigm is particularly useful for greater flexibility in application design and to overcome constraints such as real-time demands and a high load on tasking. The first contribution of this paper was that it defined context management by defining a CM and the primitives in its associated user interface, to support the above paradigm. Systems that do not support threads, but support server processes can also take advantage of the context management services. The context management services permit a transaction processing application to specify which work is to be handled at any instant. The application using these services can then divide the work within and among the threads or processes, and be assured the RMs will know which transaction the work belongs to. While context management fits naturally in the peer-to-peer transactional paradigm, it also allows legacy, process-oriented systems to increase the transactional throughput by allowing multiplexing of transactions within one process.

A second contribution of this paper is that it showed how context management is necessary to solve problems such as deadlocks and protocol violations, and to handle loopback situations that arise in distributed transaction processing when using the paradigm in which a thread could be associated with multiple transactions. The paper then gave solutions to these problems in transaction processing. The paper also showed how context management is used to provide the functionality of unchained transaction processing to those systems that only provide chained transactional processing. A third contribution of this paper is that it showed how the limitations of the X/Open proposal can be overcome by the use of context management in conjunction with their standard API for distributed transaction processing.

We expect context management will become increasingly important to efficiently handle higher tasking demands and real-time demands on the operating system. We believe that our context management API also addresses all the present as well as likely future needs of the distributed transaction processing community. We have implemented a prototype of the CM and its user interface for use with distributed transaction processing. The VM operating system Shared File System has been enhanced to provide several features of context management support described here.

References

[1] Ananda A L, Tay B H and Koh E K 1992 A survey of asynchronous RPC *ACM Operating Syst. Rev.* **26** 92–109

[2] Citron A 1991 Context manager *Proc. 4th Int. Workshop on High Performance Transaction Systems (Asilomar)*

[3] Comer D 1988 *Internetworking with TCP/IP: Principles, Protocols and Architecture* (Englewood Cliffs, NJ: Prentice-Hall)

[4] Date C J 1994 *An Introduction to Database Systems* 6th edn (Reading, MA: Addison-Wesley)

[5] Gray J N 1978 Notes on data base operating systems *Operating Systems—An Advanced Course (Lecture Notes in Computer Science 60)* ed R Bayer, R Graham and G Seegmuller (Berlin: Springer)

[6] OS/2 Technical Library 1994 *OS/2 Warp Control Program Programming Reference* IBM G25H-7102-00

[7] Systems Network Architecture 1993 *Transaction Programmers' Reference Manual for LU Type 6.2* Document Number SC30-3084-5 IBM

[8] Systems Network Architecture 1994 *Sync Point Services Architecture Reference* Document Number SC31-8134-0 IBM

[9] King A 1994 *Inside Windows 1995* (Redmond, WA: Microsoft Press)

[10] Kshemkalyani A D and Singhal M 1994 On characterization and correctness of distributed deadlock detection *J. Parallel Distrib. Comput.* **22** 44–59

[11] Lampson B W 1981 Atomic transactions *Distributed Systems: Architecture and Implementation—An Advanced Course (Lecture Notes in Computer Science 105)* ed B W Lampson (Berlin: Springer) pp 246–65

[12] Mohan C, Lindsay B and Obermarck R 1986 Transaction management in the R* distributed data base management system *ACM Trans. Database Syst.* **11** (4)

[13] Moskowitz D *et al* 1993 *OS/2 2.1 Unleashed, Sam's Publishing* (Englewood Cliffs, NJ: Prentice-Hall)

[14] Information Technology—Open Systems Interconnection—Distributed Transaction Processing—Part 1: OSI TP Model; Part 2: OSI TP Service 1992 ISO/IEC JTC 1/SC 21 N

[15] Ramamritham K and Chrysanthis P 1994 Synthesis of extended transactions using ACTA *ACM Trans. Database Syst.* **19** 450–91

[16] Samaras G, Citron A and Kshemkalyani A 1997 Reconciling chained and unchained transactional support for distributed systems *J. Syst. Archit.* **43** 229–43

[17] Samaras G, Britton K, Citron A and Mohan C 1995 Two-phase commit optimizations in a commercial distributed environment *J. Distrib. Parallel Databases* **3** 325–60

[18] Singhal M and Shivaratri N 1994 *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems* (New York: McGraw-Hill)

[19] X/Open Consortium 1992 *Distributed TP: a) The TX Specification P209 b) The XA Specification C193 6/91 c) The XA+ Specification S201*

[20] Change No. ATT-03 Thread of Control 1995 *X/Open TxRPC CAE Specification*