ELSEVIER

Journal of Systems Architecture 43 (1997) 229–243

JOURNAL OF SYSTEMS ARCHITECTURE

Reconciling chained and unchained transactional support for distributed systems

George Samaras [a,b,*], Andrew Citron [a], Ajay Kshemkalyani [a]

[a] *IBM Distributed Systems Architecture, IBM Corporation, P.O. Box 12195, Research Triangle Park, NC 27709, USA*
[b] *Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus*

Abstract

Distributed transactions require transaction processing support either from their communications protocols or from protocols at some higher layer. One of the earliest "industrial-strength" distributed transactional support systems was provided by SNA (LUG.1 and LUG.2). As OSI 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. OSI/TP supports both chained and unchained transactions. This makes the coexistence of applications requiring both kinds of transaction processing support on a platform that supports only chained transactions problematic and reduces the interoperability of communication protocols providing transactional support. This paper addresses the above problems by identifying the extra functionality provided by unchained transactions over chained transactions. It then shows how a protocol/platform (i.e., IBM's SNA) that provides only chained transactional support can provide the functionality of unchained transactions, and without using the notion of chained or unchained transactions at the API. The paper also describes an efficient way by which protocols providing only chained support can provide the functionality of unchained transactions using the X/Open TX API based on OSI TP's Chained and Unchained Functional Units.

Keywords: Context management; Transaction processing; Databases; Distributed processing; LU6.2

1. Introduction

A transaction is defined to provide the properties of atomicity, consistency, integrity, and durability (A.C.I.D.) for any work it performs [10,5]. Distributed transactions require transaction processing (TP) support, also termed *transactional support*, either from their underlying communication protocols (e.g., LU6.2's SNA), or from some layer above the communications protocols (e.g., Tuxedo's Transaction Manager above TCP/IP). The transactional support available to the distributed transaction depends on the platform and/or the various communication protocols used. Until recently, only protocols such as

* Corresponding author. Email: csamara@turing.cs.ucy.ac.cy.

SNA [6] and TMF [18] were used to support distributed transactions in homogeneous systems. Even though OSI TP [20] represents ongoing work for supporting transaction processing in an open environment, the variety of transactional support provided by various protocols/platforms will continue to exist.

A specific transaction is delimited by a *transaction boundary*, viz., the beginning and end of the transaction. This notion is important because it relates to the concept of a transaction ID (TRANID), and in certain protocols defines the transaction state of the different distributed resources.

Two dominant paradigms to demarcate the boundaries of a transaction have emerged *chained* and *unchained* transactions. The X/Open [19,1] API, based on OSI, [13] exposes both paradigms. Existing proprietary/open communication protocols do not necessarily support both chained and unchained transaction processing. SNA's LU6.2 provides only chained support, while OSI/TP supports both chained and unchained. The non-uniformity of transactional support poses the two problems of TP support mismatch and incompatibility of TP protocols. The first problem arises when a distributed application requiring a particular kind of transactional support is not able to run because the underlying communication protocol/platform does not provide that support. This makes the coexistence of applications requiring both kinds of transactional support over a network problematic. The second problem can arise when a distributed application is not able to perform protected work because the participating nodes are in different networks, each of which supports different communication protocols/platforms that possibly provide no common TP support. This reduces the interoperability of transactional support across protocols/platforms.

Enhancing the communication protocols/platforms to support both chained and unchained transactional paradigms is not a practical solution because of its high cost and migration problems. The goal of achieving the coexistence of the two transactional paradigms within a network, and the interoperability of networks for transactional support should be attained by minimal changes to the communication protocols/platforms. This paper presents a solution to this problem by showing how the chained transactional model can emulate the unchained transactional model and vice versa.

This paper shows that the emulation of the chained paradigm by an unchained transaction can be performed without any changes to the line flows of the communication protocols. The emulation of the unchained paradigm by a chained transaction can be performed by using four features. To demonstrate the viability of this approach and explain the features of the solution, a realistic example is very useful.

A most appropriate example is SNA's LU6.2 which is the dominant protocol used for distributed transaction processing in the industry. Therefore, the solution is explained with reference to SNA's LU6.2. SNA's LU6.2 traditionally supports only chained transactions using the presumed-nothing (PN) feature [9] but has been enhanced with these four features to support unchained transactions.

- The first feature involves the notion of context and *context management* [2], which is a way of locally managing various activities within application programs, and resources with low overhead.
- The second feature provides for a nonblocking commit (sync point) call or a threading facility.
- The third feature enhances the transactional support of a communication protocol/platform to permit a participating application program to specify that it does not wish to participate in a transaction if it does not perform any work. This feature requires a 1-bit indicator on a communication protocol line flow and is provided by LU6.2's presumed abort design [11,15,9,8].
- The fourth feature provides a means of sending "out-of-band" or "expedited data" [7].

Two solutions that use the first two features are presented to let an application perform some work at a "chained" remote site in between two global transactions, and have that work excluded from those transactions. However, these solutions require more flows than does the OSI TP solution. Another solution that uses all four existing features is as efficient as the OSI TP solution and it uses SNA's API, not X/Open's TX. If emulating the unchained paradigm requires using the X/Open API, the first three of the above existing features are used in addition to a small change to the local processing and a 1-bit change to a commit line flow.

Section 2 presents a model of the system and of distributed transactions. Section 3 describes context management by the application. Section 4 defines the chained and unchained transactional paradigms, discuss their functional difference, and illustrates how LU6.2 can provide the functionality of unchained transactions by giving three solutions. Section 5 describes how chained and unchained transactions can coexist even if the underlying transactional support is of one type only. In particular, it shows how SNA's LU6.2 can support the functionality of unchained transactions using the chained/unchained interface of X/Open even though SNA LU6.2 is traditionally viewed as providing only chained transaction support. Section 6 gives concluding remarks.

2. Distributed transaction execution

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.

A user's *application program* initiates or participates in a distributed computation, which consists of a set of transactions executed serially. A transaction consists of a set of operations that are executed to perform a particular logical task, generally making changes to data resources such as databases or files (the work performed on behalf of a transaction is also known as *protected work*). The changes to these resources must be committed or aborted atomically before the next transaction in the series can be initiated. This is achieved by a *commit* protocol.

A distributed computation is associated with a tree of processes [1] that is created as the application executes. The process tree links the processes that perform the transactions of the distributed computation. Processes may be created at remote nodes (and even locally) in response to the data access requirements imposed by the application. Consequently, a creator-createe relationship exists between the processes. The tree may grow as new sites are accessed by the transactions. Subtrees may disappear either in response to application logic or because of site and communication link failures.

The dominant models supporting distributed transactions are the hierarchical model, such as that usually associated with client/server computing and the peer-to-peer model that supports a more general model of computation [6]. In the client/server model, all the transactions (i.e., $t_1, \ldots, t_6, t_7$) constituting the distributed computation t are initiated and committed by the root process. The server processes participate in the computation by executing requests from their client. In the peer environment, however, each process has the same privileges and rights as any other process in the process tree. Any program can initiate or commit a transaction. Two programs can initiate work independently with or without any communication between them. This is in contrast to the hierarchical model, where the client starts the transaction and the servers wait until they get re-

[1] In the rest of the paper, whenever we refer to a process, we are not necessarily referring to a process as defined by the operating system. A process may in fact be a lightweight *thread*, a *context*, or a thread of control (XOPEN model).

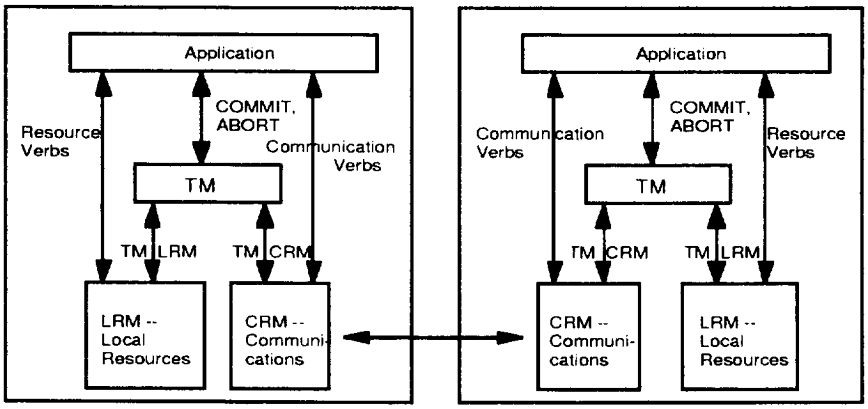


Fig. 1. Components involved in a transaction and a transaction commit tree.

quests from clients or other servers. The coordinator-subordinate relationship is established at the beginning of commit processing and endures only for the current transaction.

As shown in Fig. 1(a) process participating in a transaction accesses local resources such as data bases and files. A remote request is sent via the communication network to a remote process, which can access either local resources or additional remote resources.

Once the computations of a transaction are completed, the application instructs the *transaction manager (TM)* of its site to initiate and coordinate the commit protocol. Two types of components participate in the commit protocol under the coordination of the TM at each site: *Local Resource Managers (LRMs)* such as database and file managers, which have responsibility for the state of their resources only, and *Communication Resource Managers (CRMs)*, which manage communication resources such as conversations that enable a TM and local processes to communicate with remote TMs and remote processes. 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* is the TM acting on behalf of the process that initiates a commit operation; a subordinate is either an LRM or a remote TM that is represented at the site of the coordinator through a CRM. Remote TMs may also have subordinate LRMs and TMs. The coordinator coordinates the final outcome of the commit processing by arriving at a COMMIT or ABORT decision and propagating that decision to all subordinates. Subordinate TMs propagate the decision to their subordinate TMs or LRMs. The commit operation employs the well-known two-phase commit (2PC) protocol [4,12].

3. Context management

Context management is a local application-support mechanism that permits applications to manage logically separate pieces of work within a single locus of execution (e.g., a thread or a process) [2,17]. Context management can be used for a wide range of functions such as addressing "loopback" in distributed database operations, and enabling a server to serve a large number of clients. Multiple contexts of an application can exist within a single locus of execution (process or thread) without involving most of the operating-system process/thread overhead, such as delays in process start-up, forking and dispatching multiple processes, locking mechanisms for accessing shared tables, and extra storage. Two needs for context management as defined above are described next.

- Pre-started application programs can accept more than one incoming conversation. Each incoming conversation is independent of the other incoming conversations accepted by the application program. Server application programs need the ability, and an API, to switch between the contexts represented by the incoming conversations.
- A distributed transaction can loop back to a site that already has been "infected"with that transaction. In this situation, the TRANID does not uniquely identify the unit-of-work instance at the site. Further, if the application program has a conversation with another application program on its own node, the combination of TRANID and conversation correlator (an identifier for the conversation – the same concept as *branch ID* in OSI/TP) is also not unique. So a third identifier is needed. The process_ID does not suffice because a process can accept multiple incoming conversations and have multiple logical threads of control. The context ID identifies the logical thread of control. Context ID, TRANID, and conversation correlator uniquely identify a unit-of-work instance at a node [2].

Context ID is similar to process ID except that when one operating system process acts on behalf of many different incoming conversations, the operating-system process ID is not a unique identifier.

When an application program uses context management, the context manager keeps track of the various contexts, allows the application program to create and set a context for work, and allows an application program to switch contexts when appropriate. The context manager shares the notion of contexts with the RMs and the application program. This is similar to X/Open where all the RMs and TMs have to share the notion of "thread-of-control".

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

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. A locus of execution can create a new context using *Create_Context*. The active context can be changed to another context using *Set_Context*. This results in the locus of execution changing to the set context.

The notion of context provides a logical separation of work done by an application program; each logically separate piece of work is done in a separate context. Context is local to a system, and distributed work done for the same transaction is not part of the same context. When an application program is involved in multiple transactions at a time, each transaction is done in a separate context at the application program. The same context may be involved in multiple transactions sequentially. The application program assumes the responsibility of keeping track of its various contexts, coordinating the data spaces of the various contexts, and switching between contexts.

Context management enables an application program to do work in different contexts, each of which has a separate identity. Work performed by an appli-

[2] The RPC model does not use the notion of context, because it offers less parallelism in processing; i.e., one of the two communicating partners is suspended at any time.

cation program in one context is not considered to be work performed by any other context of that application program.[3]

4. Chained and unchained transactions

4.1. Evolution

The notions of chained and unchained transaction processing are confusing because they are used as resource characteristics as well as API choices. This section clarifies the notions, and furthermore shows that the distinction between chained and unchained transaction processing is a somewhat arbitrary and artificial way of expressing a more elementary property of transaction processing.

4.1.1. Chained transactional support

SNA's LU6.2 transactional processing support was shipped in 1981. A protected communication resource participating in a transaction was placed in the next transaction as soon as the previous one was committed. This was an optimization that worked well because (a) it was essential to avoid sending the extra data representing the new TRANID and (b) it allowed different partners to initiate the next transaction. This manner of using the protected conversation led to the notion of a chained resource. A protected resource is called *chained* if and only if the end of participation in a global transaction implicitly places the resource in the next global transaction and the new TRANID is somehow made available to it. Any work performed by the protected resource is always part of a global transaction.

4.1.2. Unchained transactional support

In the mid 80s to early 90s, several other transaction processing systems, e.g., TMF, Tuxedo and DEC's ACMS, were built, and their implementation influenced OSI TP's definitions for transaction processing [13]. OSI/TP's unchained transactions came about because in many transaction processing systems, it is necessary for the TM to broadcast the transaction ID to the various resource managers, so the resource managers know to associate the thread of control with the transaction ID. In these systems, the TM waits for the application to instruct the TM to broadcast a begin-transaction message. It is therefore possible for a thread of control to be outside a global transaction. From this idea, OSI/TP transaction-capable dialogs were given the ability to be used inside or outside a global transaction, and the notion of unchained resources formed. A protected resource is called *unchained* if it fulfils the following two requirements:

(1) The resource does protected work on behalf of a global transaction only when it explicitly receives notification of the beginning of the transaction and the new TRANID.

(2) The resource does only unprotected work or work that is not part of a global transaction outside the boundaries of a global transaction, i.e., Begin_Transaction, End_Transaction (or commit).

Building on the idea that it was possible for a thread of control doing protected work to be outside a global transaction for a while, the X/Open API designers designed the TX interface (an API) that surfaced the chained/unchained paradigm to the application program [19]. OSI TP provided both chained and unchained resources (dialogues) in its specifications definition.

4.2. Unchained transactions: A functional requirement, or a proposed solution?

4.2.1. Unchained transactions

The difference between chained and unchained transactions is the manner of demarcating the boundaries of a transaction by the application. An applica-

[3] We will use "application" henceforth to mean application program.

tion is said to follow the *unchained* paradigm if it fulfils the following two requirements:

(1) The application does protected work on behalf of a global transaction only when it explicitly specifies the beginning (and end) of the transaction.

(2) The application can do only unprotected work, or work that is not part of the global transaction, outside the boundaries of a global transaction, i.e., Begin_Transaction, End_Transaction (or commit).

4.2.2. Chained transactions

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

Unchained transactions offer more flexibility to the application in coordinating activities with the partner program but they permit chance of error causing violation of A.C.I.D. properties. (Clearly, the notion of chained and unchained resources is an artifact from implementations of chained and unchained resources because it deals with how and when RMs are notified of TRANIDs. It does not affect the functionality or requirements of applications. More discussion on chained and unchained resources follows in the next section.)

4.2.3. Unchained transactions: A proposed solution for a functional requirement

There have been recent requirements that SNA's LU6.2, which traditionally provides "chained" transactional support, should provide unchained transactional support similar to OSI TP's unchained dialogs. However, this is a loosely stated requirement because the essential functionality difference between unchained and chained transactions is not whether or not the application needs to specify the beginning and end of each transaction. *The essential functionality difference is whether or not the application can use the same protected dialog to do protected work and other work (unprotected work or protected work outside the global transaction of the application).* The precise requirement on SNA's LU6.2 is that it should be possible for an application to have a protected conversation, in which the partner did protected work, or unprotected work that was outside the global transaction that local application was in. Without the notion of context management LU6.2's protected conversation could not be used by the application to do any protected work outside the global transaction associated with that conversation.

4.3. SNA LU6.2's counter-solutions to unchained transactions

Fortunately, recent enhancements to LU6.2 have equipped it to meet the function offered by OSI TP's unchained transactions. Understanding the requirement was particularly important because in a truly peer-to-peer environment such as LU6.2, where any peer can initiate or terminate a transaction, and where some resource managers can or cannot be used outside a global transaction, adding the unchained support is not particularly pretty, simple, or easy to use.

LU6.2 can provide the user the function implied by unchained transactions by building on the functions already added to LU6.2. LU6.2's solution does not use the notion of chained and unchained transactions nor the chained/unchained API required by OSI TP and X/Open. The solution uses context management [2,17], and LU6.2's presumed-abort optimization [15,9,8] and expedited data [7] In fact, LU6.2's solution goes beyond the requirement and provides more than does the chained/unchained paradigm. LU6.2 can allow unprotected work and protected work outside the global transaction to occur concurrently with protected work in a global transaction, using the same protected conversation. We present the more flexible solution, and present two more solutions that refine it to progressively give the exact semantics of unchained transactions.

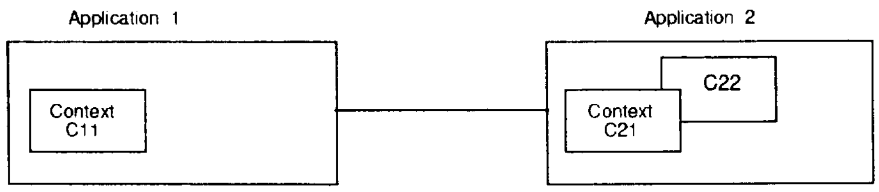


Fig. 2. Functionality of unchained transactions using "chained" resources. (This can be achieved using multiple contexts within Application 2.)

The following notation will be used for identifying contexts: *Cij* is the *j*th context of application *i*. We will consider applications 1 and 2, which are at sites 1 and 2, respectively, as shown in Fig. 2. Applications 1 and 2 communicate through CRMs (not shown in Fig. 2). Application 1 can do protected work in context *C*11 with context *C*21 of Application 2. Application 1 can also do unprotected work in the same context with a context other than *C*21 of Application 2.

4.3.1. Solution 1

This solution uses local context management, and multithreading or nonblocking support. Consider the following scenario: Application 1 executing in Context *C*11 has a protected conversation with Application 2 in Context *C*21. Application 1 sends a message to Application 2 indicating "change context and do work × outside of the global transaction". Application 2 then begins a new context *C*22, does the work, (commits it if the work is part of some transaction), and replies to Application 1 "the work is done". Application 2 switches context to *C*21 and resumes work in the global transaction. When Application 1 issues the commit, the protected work that was done in the original context *C*21 at Application 2 gets committed, but the work done in the other context *C*22 is not affected by these 2PC flows.

$4n$ flows, where n is the number of server processes participating in the global transaction, are required for committing the protected work. The work done concurrently by Application 2 outside this global transaction using the same conversation did not contribute to any message overhead. The solution permits work that is part of a global transaction, as well as work (protected or unprotected) outside that global transaction simultaneously to use the same conversation; hence it is more general than the chained/unchained paradigm. Note that Application 2 may of its own accord switch context and initiate protected work (with a different partner) or unprotected work.

4.3.2. Solution 2

This solution uses local context management, and multithreading or nonblocking support to achieve the semantics of unchained transactions. Unprotected work or work outside a global transaction is serially performed after protected work, on the same conversation.

Consider the following modification of the previous scenario: Application 1 executing in context *C*11 has a protected conversation with Application 2, context *C*21, and commits a transaction. Application 1 sends a message to Application 2 indicating "Change context and do work × outside of the global transaction". Application 2 then begins a new context *C*22, does the work (commits it if the work is part of a transaction), and replies to Application 1: "the work is done". Application 2 switches context to *C*21. Application 1 meanwhile does protected work on another branch of the transaction tree. Then Application 1 issues the commit. Application 2, context *C*21, can vote read-only in the voting phase if all applications in its subtree also vote read-only [4]. It can vote read-only even if protected work was done, because it was not done in the context of the global transaction. This commit uses only $2n$ flows for Application 2's subtree. However, with OSI TP's unchained transactional paradigm no flows are involved to Application 2 or its subtree. These flows can be eliminated in SNA's LU6.2 by using the recently added "expedited data" and LU6.2 presumed-abort enhancements, described below.

4.3.3. LU6.2's presumed abort

With LU6.2's presumed abort, it is possible to leave a partner out of a commit or backout operation if it does not participate during a transaction [7,8,14,15]. An application context is characterized by its potential to participate in the next transaction as *OK to leave out* or *not ok to leave out*. Specifically:

- An application context is *OK-to-leave-out* if it can be suspended until needed. It will be suspended indefinitely until the transaction associated with it receives the new TRANID for further transaction processing. Hence, this application context can participate in applications running in unchained mode without having the associated LU6.2 conversation terminated at the end of each global transaction.
- If an application context is *not OK-to-leave-out* from the next transaction, then the local transaction manager has already received the new TRANID and the application context can start processing independent of its initiator (in a chained mode). This application context is not suspended and is essentially chained.

An application context can specify itself as *OK-to-leave-out* using the Set_Syncpt_Options call. The TM at the partner is informed of this setting on the YES vote in the voting phase of the commit flow and the setting goes into effect from the next transaction. LU6.2 with the presumed abort enhancement implies that the partner that has declared itself *OK-to-leave-out* can be left out of subsequent transactions with this program until it is asked to participate in a transaction with this program. Even in the peer-to-peer structure, the partner application context is prohibited from independently initiating work for the next transaction within this context.

4.3.3.1. Expedited data. *Expedited data*, also known as out-of-band data, support in LU6.2 [7]: Expedited data is defined as being outside of transactional control. An application can use expedited data to do unprotected work with the partner in a context other than the suspended context at the OK-to-leave-out partner. LU6.2's API permits the application to send and receive expedited data by using SEND_EXPEDITED_DATA and RECEIVE_EXPEDITED_DATA verbs, respectively.

Using expedited data, 86 bytes can be sent or received with an API call [5]. For transmitting larger volumes of data, the application has to perform its own segmenting.

4.3.3.2. Solution 3. This solution uses local context management, multithreading or nonblocking support for commit calls, LU6.2's presumed abort, and expedited data.

Consider this scenario, which is a modification of the previous one: Application 1 executing in context

[4] Read-only is a two-phase commit optimization, where an agent votes read-only during the first phase of the synchronization operation, indicating that it had not performed any work during the global transaction and therefore does not care whether the transaction commits or backs out.

[5] This restriction has been placed due to APPN segmentation.

$C11$ has a protected conversation with Application 2's context $C21$ and commits a transaction. During the commit, Application 2's context $C21$ declares itself "OK-to-leave-out". Application 1 sends an expedited data message to Application 2 indicating "Change context and do work $\times$ outside of the global transaction". Application 2 then begins a new context $C22$, does the work, (commits it if the work is part of a transaction), and replies to Application 1: "The work is done" using expedited data. Application 2 switches context to $C21$. When context $C11$ of Application 1 issues a commit, the LU6.2 CRM at Application 1 deduces that the conversation was not used for (transactional) work outside the global transaction of context $C11$ because only the expedited send and receive verbs were used, and this particular conversation would not be sent any 2PC flows. Later, if Application 1 does any subsequent protected work on this conversation and then commits it, the work that was done using the expedited flows is not included because it was done in a different context, $C22$, at Application 2. The work done in context $C22$ may persist even if the global transaction involving Application 1 and Application 2's context $C21$ is backed out. This approach provides the unchained functionality without additional protocol changes and migration problems and it avoids the read-only flows.

As in the previous solutions, Application 2 may of its own accord switch context and initiate protected work (with a different partner) or unprotected work, because only the context that was declared "OK-to-leave-out" is suspended.

This solution is as efficient as OSI TP's solution but it does not use the TX API of X/Open; rather, it uses the already defined LU6.2 API. A drawback of this solution is that the application has to do its own segmenting. If LU6.2 were required to provide the functionality of OSI TP's unchained transactions using X/Open's TX API, it could do so by using nonblocking support, context management, LU6.2's Presumed Abort feature, some changes to the local TM-CRM interface, and a 1-bit change to the Prepare flow. The next section describes this solution.

5. Enhancing chained / unchained CRMs to support X / Open's unchained / chained API

The application specifies the boundaries of a transaction in a manner compatible with the chosen transactional support – chained or unchained, using Set_Transaction_Control of the TX interface or CPIC [19,3], or an enhanced SET_SYNCPOINT_OPTIONS() of LU6.2 [7].

A chained (unchained) application running over OSI TP accesses the Chained (Unchained) Functional Unit [13]. This choice is a one-time decision for a dialog. Hence, it follows that OSI TP classifies its dialog services as chained/unchained resources; a resource is chained or unchained for its lifetime. A chained resource supports chained applications; an unchained resource supports unchained applications.

Although chained/unchained is an application characteristic, it may also be viewed as a resource characteristic; however, the two are independent. A resource should be capable of either chained or unchained support or both (e.g., LU6.2 resources with the OK-to-leave-out enhancement). The chained/unchained classification of resources is somewhat simplistic because it is really the combination of the TM and RM that together supports the application's choice of chained/unchained operation. The application preference and resource manager characteristic interact. Chained versus unchained should be defined based on the application preference, and the resource manager and transaction manager can adapt in such a way that the resource manager's characteristic is largely hidden from the application.

If an application in unchained mode were to access chained resources (e.g., LU6.2 presumed-nothing conversations), the conversation resources would have to be terminated at the end of the transaction (i.e., after commit time) to preserve the application's unchained mode. If this happens, however, the portion of the transaction tree that is subordinate to those chained resources will also be dismantled, which is not desirable. In this section, we show how an LU6.2 conversation (which is thought of as being chained) is capable of unchained support that is specified through X/Open's TX chained/unchained API. Thus, an unchained application can run on LU6.2 without having to dismantle the subtree after each commit.

The solution presented next is a generalized solution in that it details how any TM and CRM that are traditionally thought of as offering only one type of TP support can be enhanced to support both chained and unchained transactions. Using the presented solution, an application that is written assuming chained (unchained) TP support and using X/Open's API can run over a network that has been enhanced to support both unchained and chained transaction processing. The discussion is directly applicable to LU6.2 with some architecture change.

5.1. Enhancing unchained transactional support for chained applications

Consider an application written in a chained manner executing in a transaction denoted by TRANID, using unchained resources. When the application issues a Commit, the TM processes the 2PC protocol. After the 2PC, the application is implicitly in the next protected unit of work. However, the underlying protocol (CRM) is unchained and assumes that subsequent work is unprotected unless it is told otherwise. The TM that knows that the application running in chained mode will not issue the Begin_Transaction call, it has to explicitly send a Begin_Transaction(new TRANID) to the unchained partner through the CRM. This action explicitly places the partner in the next unit of protected work. Thus, the application running in a chained manner can run over an underlying protocol that offers only unchained support.

5.2. Enhancing chained transactional support (LU6.2 resources) for unchained applications

It needs to be shown that an application can perform unprotected or out-of-a-global-transaction work with its partner in between protected work on the same conversation, when the TP support offered by the underlying protocol (CRM) is for chained transactions only. The CRM that follows the chaining mode should behave as shown in Fig. 3, based on the information provided by the TM, to support both chained and unchained applications.

An application running in unchained mode can exclude its partner that has specified itself OK-to-leave-out from further (protected) work by not issuing Begin_Transaction, and hence not causing the TM to generate, and the CRM to send, a TRANID to the partner. Although this prevents the partner from doing work on behalf of a global transaction, we would like the partner whose context is suspended, to do work outside a global transaction (i.e., unprotected work, or protected work outside the global transaction). This can be achieved by having the partner do work in a different context. Consider the sequence of actions in the following specific example:

1 Applications 1 and 2 do protected work using contexts $C11$ and $C21$, respectively. Application 1 issues an End_Transaction (or commit) for context $C11$ to its TM. This initiates the 2PC flows for context $C11$. On the YES vote in the voting phase of the commit flow, Application 2, context $C21$ indicates using the 1-bit modifier in the YES vote message that it is OK-to-leave-out from the next transaction. Thus, after commitment, Application 2's context $C21$ is suspended until it gets involved in the next transaction by the arrival of a new TRANID from the partner CRM.

2 Then context $C11$ of Application 1 requests

```
Receive data from application:
If (this is the first data is to be sent after a Commit to a partner CRM that has specified its
current context OK-to-leave-out) or (this is the first data is to be sent to a partner CRM):
then query TM for chaining mode of application
    If application is in unchained mode
    then forward data to partner CRM
    else get TRANID from TM and forward to partner CRM before forwarding data
else forward data to partner CRM

        ------------------------------

Receive data from partner CRM:
If (no TRANID precedes data received from partner):
then forward data to local application
else begin new transaction using received TRANID        /* process as usual */

        ------------------------------

Receive Begin_Transaction(TRANID) from TM:
CRM forwards TRANID to the partner CRM
```

Fig. 3. Local modifications to a chained CRM to behave as an unchained CRM.

Application 2 to perform work outside the global transaction, as follows.

- Context $C11$ at Application 1 sends data to its partner, Application 2. The first part of the data contains application data that tells the partner to issue "Create_Context($C2*$)" and "Set_Context($C2*$)". This application data tells the partner to create and make active a new context (e.g., $C22$).
- When site 1's CRM processes the "send" command, it checks with the local TM whether the application is running in chained mode.
- TM at site 1 informs the local CRM that the application is running in unchained mode.
- The CRM at site 1 forwards the data to the partner CRM at site 2 without supplying any TRANID.
- CRM at site 2 receives the data from the partner CRM without any TRANID. The current context $C21$ at the CRM is OK-to-leave-out (therefore suspended) and will not be involved in any global transaction until it is assigned a new TRANID by its parent in the transaction tree. Therefore, the CRM forwards the data received to the local application. Even though context $C21$ is suspended, the nonblocking support permits another application context to receive the data from the local CRM. The application at site 2 issues Create_Context($C2*$) and Set_Context($C2*$), which are the first instructions in the data received from the CRM of Application 1. Create_Context($C2*$) causes context $C22$ to be generated by the context manager. Set_Context($C2*$) causes Application 2 to change its context from $C21$ to $C22$, and perform work for the partner application that is running in context $C11$.

Observe that no TRANID has been assigned to this work, hence this work is outside the global transaction and it can be unprotected. It can also be protected, but outside the global transaction and unrelated to the original distributed computation. Context $C21$ of Application 2 remains suspended and cannot participate in this work or any other work. Furthermore, note that the applications are designed to cooperate and understand the context management API.

3 To get Application 2 back to the global transaction, Application 1, which is executing in context $C11$, causes Application 2 to execute Set_Context($C21$). Receiving this command can also serve as an indication that no more work is to arrive for context $C22$. Application 2 can then take any appropriate steps to terminate the work performed in context $C22$ (i.e., commit that work if protected), and changes context to $C21$. Application 1 issues a Begin_Transaction in context $C11$. When it sends data to Application 2, the CRM for Application 1 forwards the data to Application 2's CRM along with the TRANID supplied by the local TM for Application 1. The TM at site 2 starts a transaction with the newly arrived TRANID in the current context, viz., context $C21$ of Application 2.

The work performed in step 2 (by Application 2 in context $C22$ with context $C11$ of Application 1) was not included in the commit operation in step 3. because it was not part of the distributed computation associated with contexts $C11$ and $C21$. This was possible because:

- Using the OK-to-leave-out bit in the YES/NO vote of the 2PC protocol, the CRM for Application 2 could specify that the current context was OK-to-leave-out of the next transaction(s); i.e., it would be suspended, until it was told to do some protected work by the partner application.
- The protected work in the global transaction would be identified by the partner sending a new TRANID; hence, until the new TRANID was sent, all work done would be considered unprotected or outside the global transaction. The change in the local CRM processing ensured that any request for work outside the global transaction would not be sent with a new TRANID to the partner.
- The application could perform work even when its context was suspended by changing to a different context that was not suspended; this was facilitated using context management calls and nonblocking support. Thus, suspension of the application was restricted to the context in which the application had declared itself OK-to-leave-out and the nonblocking support enabled the application to regain control in spite of the suspended context.

Observe that to resume global protected work in step 3, it was the application's responsibility to switch context back to the appropriate value before commencing work on behalf of a global transaction.

The application running over the TX interface can change the chaining mode from chained to unchained and vice-versa using *Set_Transaction_Control*. This can be supported by the underlying system supporting only chained transactions and enhanced as described in Fig. 3 by one of the following two mechanisms:

- An application protocol that uses application data can be used between the two partner systems to negotiate the chaining mode. The selected chaining mode goes into effect at the time the next transaction in that context is begun.
- One bit on the Prepare flow conveys to the partner TM/CRM whether the system application wants the chained or unchained mode for the next transaction. If this bit indicates that the application has chosen the chained mode, then the transaction proceeds as usual. If this bit indicates that the application has chosen the unchained mode, then the partner CRM (and its subtree) must vote OK-to-leave-out on the YES vote in the voting phase of the commit protocol in order that the transaction can progress. However, if the partner (and its subtree) do not vote OK-to-leave-out, then the partner and its subtree has to be dismantled because they are implicitly choosing the chained mode despite the request on the Prepare flow to choose the unchained mode.

Thus, using the existing OK-to-leave-out bit on the 2PC flows to specify that the application represented by the CRM was OK-to-leave-out, context management and nonblocking support, applications running in chained and unchained modes can run over protocols that have support for only chained transactions if the following changes are made. SNA's LU6.2 needs to modify some local interaction between the TM and the CRM, and have a new bit on the Prepare flow to support X/Open's TX API. An unchained application written for OSI TP using X/Open's API could then run over LU6.2 and SNA without dismantling the LU6.2 subtree at each commit/abort time.

6. Conclusions

The classification of transaction processing modes into chained and unchained is shown to be one way of expressing the hitherto poorly-understood requirement that protected conversations also allow work outside the global transaction with no additional overhead. LU6.2 protected conversations, which were believed not to allow work outside the global transaction, were demonstrated as being capable of allowing such work. It was shown that simply by using context management and nonblocking support, local platform-specific features, LU6.2 satisfies this requirement (see solution 1 and solution 2). However, these solutions require some flows that are avoided in OSI TP. Therefore, another solution that additionally uses the presumed-abort and expedited-data features of the LU6.2 architecture was presented and shown to be as efficient as the OSI TP solution (see solution 3). Thus, LU6.2 already satisfies the requirement using its own API of expedited data, not the X/Open chained/unchained API.

However, if the somewhat artificial manner of expressing this requirement in the form of chained and unchained transactions were imposed on LU6.2 through X/Open's API, some changes to the TM and XA + interface would be needed along with a new bit on the Prepare flow. In demonstrating the viability of supporting the chained/unchained API over LU6.2, it was shown more generally that an application running in unchained mode can run over any protocol/platform offering only chained transactional support by supporting four features – the OK-to-leave-out bit in the YES/NO vote flow of the 2PC protocol, the transaction_control bit on the Prepare flow, context management support, and nonblocking support – and small changes to the local CRM processing. It was also shown that an application running in a chained mode can run over any protocol/platform offering only unchained transactional support with no changes to the protocol.

Thus, the problems caused by communication protocols/platforms offering different kinds of transactional support can be resolved efficiently by using the strategy described in this paper. In particular, for SNA's LU6.2 with the presumed-abort enhancement, no changes to the protocols are required (unless the chained/unchained API is required) because the LU6.2 architecture already has the features to support this strategy. The solution enables the applications written for unchained transactional support to execute over the large installed base of LU6.2 and SNA networks that offer only chained transactional support.

References

[1] E. Braginsky, The X/Open DTP effort, in: *Proc. 4th Internat. Workshop on High Performance Transaction Systems*, Asilomar, 1991.

[2] A. Citron, Context manager, in: *Proc. 4th Internat. Workshop on High Performance Transaction Systems*, Asilomar, 1993.

[3] Common Program Interface Communications Specification, Working draft of SC31-6180-01, IBM, 1993.

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

[5] T. Haerder and A. Reuter, Principles of transaction oriented database recovery – A taxonomy, *Comput. Surveys* 15 (4) (1983).

[6] Systems Network Architecture LU6.2 Reference: Peer Protocols, Document Number SC31-6808-1, IBM, 1990.

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

[8] IBM Database System DB2, Version 3, Document Number GC26-4886, IBM, 1994.

[9] Systems Network Architecture. SYNC Point Services Architecture Reference, Document Number SC31-8134, IBM, September 1994. It presents in detail IBM's Presumed Nothing commit protocol. Authors: George Samaras, Kathryn Britton, Andrew Citron.

[10] B.W. Lampson, Atomic transactions, in: B.W. Lampson, ed., *Distributed Systems: Architecture and Implementation An Advanced Course*, Lecture Notes in Computer Science, Vol. 105 (Springer, Berlin, 1981) 246–265.

[11] C. Mohan, K. Brriton, A. Citron and G. Samaras, Generalized presumed abort: Marrying presumed abort and SNA's LU6.2 commit protocols, in: *Proc. 5th Internat. Workshop on High Performance Transaction Systems (HPTS)*, Asilomar, 1993; Also available as IBM Research Report, IBM Almaden Research Center, 1991.

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

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

[14] G. Samaras, K.. Britton, A. Citron and C. Mohan, Two-phase commit optimizations and tradeoffs in the commercial environment, in: *Proc. 9th Internat. Conf. on Data Engineering*, Vienna, Austria, 1993.

[15] G. Samaras, K. Britton, A. Citron and C. Mohan, Enhancing SNA's LU6.2 Sync Point to include presumed abort protocol, IBM Tech. Rept. TR29.1751, IBM Research Triangle Park, 1993.

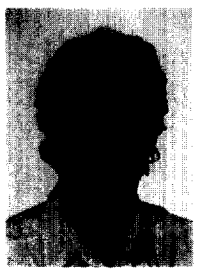
[16] G. Samaras, A.D. Kshemkalyani and A. Citron, Reconciling communication protocol support for chained and unchained transactions, in: *Proc. 2nd Internat. Conf. on Computer Applications to Engineering Systems*, Cyprus, 1993.

[17] G. Samaras, A. Kshemlakyani and A. Citron, Context management and its applications to distributed transactions, in: *Proc. 16th IEEE Internat. Conf. on Distributed Computing Systems*, Hong Kong, 1996.

[18] The Tandem Database Group, NonStop SQL: A distributed, high-performance, high-availability implementation of SQL, in: *Proc. 2nd Internat. Workshop on High Performance Transaction Systems*, Asilomar, 1987.

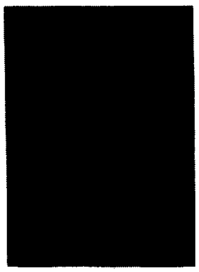
[19] Distributed TP: (a) The TX Specification P209, (b) The XA Specification C193 6/91, (c) The XA+ Specification S201, X/Open Consortium, November 1992, February 1992, April 1993.

[20] F. Upton IV, OSI distributed transaction processing, An overview, in: *Proc. 4th Internat. Workshop on High Performance Transaction Systems*, Asilomar, 1991.



George Samaras received a Ph.D. in computer science from Rensselaer Polytechnic Institute, USA. He is currently an assistant professor at the university of Cyprus. He was previously at IBM Research Triangle Park, USA. He served as the lead architect of IBM's distributed commit architecture (LU6.2 Sync Point). His research interest include transaction processing, databases, mobile computing, object-oriented technology and real-time systems. He also served on several of IBM's internal international standards committees.

Andy Citron holds six software patents related to distributed data processing. He is currently doing multimedia PC software development for the Mwave Digital Signal Processor. Prior to tat he was the lead architect for the IBM's APPC SNA Communication protocols. He works at IBM's Research Triangle Park facility in North Carolina. He holds an MS in Computer systems from SUNY at Binghamton.



Ajay D. Kshemkalyani received the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the M.S. and Ph.D degrees in computer and information science from the Ohio State University, USA, in 1988 and 1991, respectively. He is currently an Advisory Programmer in IBM at Research Triangle Park, North Carolina, USA. He is also an Adjunct Assistant Professor in Electrical and Computer Engineering at North Carolina State University. His current research interests include distributed computing, operating systems, computer architecture, and networking. He is a member of the ACM and a Senior Member of the IEEE Computer Society.