Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing

Meichun Hsu*
Digital Equipment Corporation
Mountain View, CA 94022
hsu@tpwest.dec.com

Avi Silberschatz
University of Texas
Austin TX 78712
avi@cs.utexas.edu

Abstract

Distributed transaction processing requires the use of a commit protocol to ensure failure atomicity of distributed transactions. Commit protocols are quite expensive since they require several rounds of messages between the coordinator and each of the participating sites, may result in a blocking condition, and in general may require transactions to hold data locked for a long duration of time. We propose an alternative approach to distributed transaction processing based on unilateral commit and persistent transmission. Instead of executing a unit of work as a single distributed transaction as in the traditional transaction execution paradigm, we look for opportunities to execute it as a structured set or a sequence of smaller, possibly single-site atomic transactions. Each such transaction, once executed, is committed independently of other transactions in the task. In addition, a method for rigorously maintaining the linkage between the steps is provided for by a persistent transmission mechanism.

1 Introduction

A transaction is a unit of work performed on shared data which preserves the correctness of data. A transaction, in definition, is atomic, i.e., it preserves serializability, failure atomicity and permanence properties. Transactions are used as a foundation for building reliable distributed database systems. In a distributed database system consisting of multiple sites, every site contains one or more data resource managers, or RM’s, which manage the fragments of the database on that site. A common paradigm for executing a transaction in a distributed database system is for a transaction to be distributed to multiple RM’s. Each RM executes a subquery of the transaction which accesses data fragments stored on that site. Distributed transaction execution thus spans a number of sites (or RM’s) which must be coordinated by a commit coordinator site at the end of a transaction to commit or abort the transaction.

The two-phase commit protocol [2PC][BH87] is used to ensure the failure atomicity and the permanence properties of distributed transactions.

The transaction model offers a powerful abstraction for distributed shared-data computing. The programmers are relieved of complicated and error-prone tasks of concurrency control and recovery by simply bracketing a logical unit of work into a transaction. However, when a logical unit of work involves a number of distributed sites, executing this unit of work as a transaction using a general transaction management algorithms can be very costly. Some disadvantages of the general distributed transaction management algorithms are:

1. Two-phase commit is expensive in general. It involves two round-trip messages between the commit coordinator and each participating site.

2. If the commit coordinator fails during two-phase commit, a blocking condition may occur, degrading system availability. The three-phase commit protocol [CS86] can alleviate the blocking problem; however, it is more expensive than two-phase commit, and none of these protocols can eliminate the blocking condition if a network partition occurs.

3. Long-running transactions can generate excessive locking overhead. Participation in two-phase commit further lengthens the locking time.

4. In a heterogeneous environment, some sites may use Database Management Systems which do not offer two-phase commit, or may not wish to subject their databases to blocking conditions which are potentially generated from two-phase commit participation.

There are many applications where the traditional approach to distributed transaction management is too re

*This work was performed while the author was on the faculty at Harvard University; the research was supported in part by the National Science Foundation under grant NSF-IRI-88-09491.

1Obviously, some optimizations are possible for special cases. For example, the “prepare” and “ready” messages may be piggybacked with requests and answers, and the read-only sites may not have to participate in the second phase of commit. Further more, the final “Ack” messages from the participating site to the commit coordinator may be lazy messages which employ less expensive communication protocols.
restrictive. We thus argue that alternatives to the traditional execution paradigm will be useful and necessary.

It may be possible to organize a unit of work (which we will also call a task) spanning multiple sites into a number of “1-site” steps, each of which can be executed and “committed” locally. Each 1-site step updates only database fragments residing on a single site. For example, in executing a funds transfer task which involves a debit site and a credit site, the debit site may initiate the task (i.e., the transfer), debits an account in its database, sends a credit-subquery to the credit site, and “commits” the debit “step” locally. The credit-subquery arrives at the credit site, which executes and commits the credit “step”. This is in contrast with the traditional paradigm, where the funds transfer task is to be executed as a single distributed transaction spanning two sites and involves two-phase commit. If the decomposed execution does not compromise database integrity, then the explicit use of two-phase commit for processing this funds transfer task can be avoided.

Following the above discussion, we propose an approach to distributed transaction processing based on unilateral commit and persistent transmission. Instead of executing a unit of work, or a task, as a single distributed transaction as in the traditional transaction execution paradigm, we look for opportunities to execute it as a structured set of smaller, possibly single-site, atomic transactions, which we will call steps. Each step, once executed, is committed independently of other steps in the task, thus the term unilateral commit. In addition, a method for rigorously maintaining the linkage between the steps is provided for by a persistent transmission mechanism. Informally, the persistent transmission mechanism ensures that if any step in the task is executed and committed, then all steps of the task will be executed and committed. We call this paradigm distributed transaction processing the unilateral commit paradigm, or UCP.

Although executing a task T using the unilateral commit paradigm appears intuitively appealing, the execution semantics is different from that of the traditional paradigm where the entire task T is modeled and executed as a single atomic transaction. We will discuss the execution semantics in Section 3. In particular, we will define a virtual transaction semantics for tasks, which allows the unilateral commit paradigm to preserve serializability and failure atomicity at the task level. A task executed using UCP satisfies the virtual transaction semantics if the execution produces the same effect as the execution of the task as a single atomic transaction. Obviously, not all tasks will satisfy the virtual transaction conditions when executed using the unilateral commit paradigm. Weaker semantics for the paradigm will have to be explored.

The remainder of the paper is organized as follows. Section 2 introduces persistent transmission. Section 3 describes the unilateral commit paradigm in detail. Section 4 describes implementation of the persistent transmission protocol. Section 5 offers example applications of virtual transactions executed using UCP. Section 6 summarizes the paper and points out future research.

2 Persistent Transmission

In this section we introduce a type of (abstract) inter-site communication channels which are persistent, i.e., data sent via such channels are guaranteed to survive system crashes or communication link failures. We call such communication channels persistent pipes.

A persistent pipe is a communication channel for which two types of operations can be performed:\footnote{To distinguish operations on persistent pipes from other communication primitives, we prefix the operations using a letter P.}

1. \texttt{P\_Send (Pipe\_id, OP, argument\_list);}
2. \texttt{e = P\_Receive (Pipe\_id);}

Operations performed on a persistent pipe are issued from an atomic transaction. The transaction semantics of the persistent pipe operations is one of the most important aspects of the persistent pipe abstraction. If the transaction issuing the \texttt{P\_Send} operation is committed, then it is guaranteed that the message sent by this operation will be transmitted through the pipe. If the transaction issuing the \texttt{P\_Receive} operation is committed, then the message received by this operation is permanently removed from the pipe. If the transaction issuing a \texttt{P\_Send} or a \texttt{P\_Receive} operation fails (i.e., is aborted), then it has no effect on any of the pipes it operated on. With persistent transmission, the linkage between the steps of a task can be properly maintained.

Persistent transmission can be implemented by a combination of message logging, which guarantees that no system failure will cause a message to be lost from the message buffer, and a reliable transport protocol, which guarantees detection of missing messages and their retransmission. A message is logged (i.e., reliably captured) if and only if the atomic transaction which sends the message commits. If a failure occurs after a transaction is committed but before the destination site receives the messages, a recovery protocol makes sure that the missing messages will be retrieved from the log at the sending site and retransmitted and executed at the destination site upon recovery. Therefore message logging simply participates in the regular logging activities associated with committing a transaction.

To allow the transaction to commit unilaterally, i.e., without waiting for an explicit agreement from the destination site on the transmission, the sender site must send out the message only after the sending transaction has committed at the local site. If no failure occurs, the message will arrive at the destination site in due course, and the destination site will execute and commit a transaction which \texttt{receives} the message. If failure occurs, the recovery protocol ensures proper retransmission. Implementation of persistent transmission is discussed in a later section.

The persistent pipe abstraction is inspired by the \texttt{queue} abstraction discussed in [BHM\textsuperscript{90}]. In [BHM\textsuperscript{90}], a detailed
3 Unilateral Commit

We first provide an overview of the unilateral commit paradigm using an example task \( T \) (a unit of work) which involves four sites. Let \( S_0 \) be the initiator site. \( T \) needs to access and update data resources on sites \( S_0, S_1, S_2, S_3 \). Let the fragment of \( T \) which executes on site \( S_0 \) be denoted as \( T_0 \), and those executed on the remote sites \( S_i \) as \( T_i \), then the structure of \( T \) can be represented as follows:

\[
\begin{array}{c}
T_0 \\
T_1 \\
T_2 \\
T_3 \\
\end{array}
\]

If \( T \) is executed as a distributed transaction, the general distributed transaction execution mechanism requires that a distributed commit be carried out at the end of \( T \), involving data resource managers (RM's) on all four sites.

With persistent pipes available between sites, \( T_0 \) is executed on site \( S_0 \) as before; however, subqueries \( T_1 \) to \( T_3 \) which update data resources on remote sites, are sent from \( S_0 \) to remote sites via the persistent pipes between them. \( S_0 \) "commits" \( T_0 \) locally as soon as all "send" commands issued to persistent pipes have been acknowledged. By "acknowledgement" we mean that the "send" call to the persistent pipe service at site \( S_0 \) has returned; this does not imply that any communication with remote sites has been involved. In fact, as will be shown later, "sent" to a persistent pipe is implemented as a local operation. Once a subquery arrives at a remote site and is received by the appropriate remote process, the process carries out the update and commits its update on that site.

3.1 Definition of the Unilateral Commit Paradigm

Let \( S \) be a set of sites, where each site \( S_i \) stores a database fragment \( D_i \). Given a task \( T \), a UCP-decomposition of the task is defined to be a decomposition of \( T \) into a tree of subtasks (or steps) \( T_j \)'s, with a root subtask \( T_0 \), such that each \( T_j \) accesses only one particular data fragment \( D_i \). We denote a particular UCP-decomposition of a task \( T \) as \( T \). Thus \( T \) corresponds to some tree of subtasks of \( T \). If \( T_j \) accesses data fragment \( D_i \), then we say that site \( S_i \) is the execution site of \( T_j \). Let \( S_0 \) be the execution site of \( T_0 \). \( S_0 \) is also called the initiator site of \( T \) and \( T_0 \) the initiator step of \( T \).

An UCP-execution of \( T \) in \( T \) is defined to be an execution of the subtasks in \( T \) based on an execution structure defined below. If a task is executed using UCP, we call it a UCP task.

It is assumed that a persistent pipe has been initialized between every pair of sites in \( S \). The initiator site of \( T \) executes \( T_0 \) as follows:

```
Begin Transaction T_0;
update local database fragment;
for each child subtask T_j of T_0 updating data fragment at site s, i do;
P_Send(pipe,i, T_j, arguments);
Commit Transaction;
```

Note that the messages for invoking child steps of the initiator step are sent by the initiator site if and only if the step is committed. A message is guaranteed to be received and executed by a transaction at the destination site (i.e., the "child" site) exactly once.

At each destination site \( s_i \), an agent for the persistent pipe executes as follows:

```
Do forever:
Begin Transaction;
e = P_Receive(pipe,i);
call routine to do operation T_j in e;
for each child subtask T_k of T_j updating data fragment at site s, i do;
P_Send(pipe,i, T_k, arguments);
Commit Transaction;
```

A task is terminated when all the leaf subtasks are committed. Execution of a task using UCP corresponds to the execution of a tree of local transactions (i.e., transactions updating data fragments on a single site) where the parent and the child are linked together by a message sent from the parent to the child through a persistent pipe. In some applications, simpler structures (e.g., a two-level tree) may be adequate.

The programming interface in UCP consists of transaction primitives (i.e., Begin/Commit Transaction) and primitives on the persistent pipes (i.e., P_Send and P_Receive). In addition to these primitives, other services for programming a distributed task using UCP may also be supported. The message identifier, for example, can embed a task id which identifies the task for which the message is generated. The first transaction executed on be half of the task can generate the task id. Auditing programs can be written to trace the steps of a task by examining the messages sent and received through the persistent pipes.

Note that in our definition of UCP we have left the precise meaning of the notion of a site unspecified. A "site" can be a node in a distributed homogeneous database system (e.g., a node in System \( R^* \)), or it can be a node in a distributed heterogeneous database system (e.g., a particular database system in a network of database systems each belonging to a different organization unit and each running on a different hardware/software platform.). A site may in turn consist of multiple physical machines interconnected through an internal network. What is required is that each
"site" is capable of executing local, physical transactions accessing data stored on it.

We have also left open in our specification of UCP whether the UCP decomposition of a task is determined a priori or dynamically during run time. In many applications, given a task T, its actual decomposition may depend on the arguments and the state of the database. For example, a funds transfer request may consist of only two steps, a debit step or a credit step, if none of the customers involved are defense contractors. However, it may involve three steps, a debit, a credit and a notify step, if one of the customers is a defense contractor. In other words, it may be too restricted to require that the UCP structure of a task be completely known before run time.

3.2 Semantics of UCP

Execution of a task T using UCP in general will not guarantee atomicity of the task. We examine failure atomicity, permanence, and serializability properties as follows.

UCP guarantees the following:

- If the root step $T_0$ of $T$ commits then all child subtasks of $T_0$ will be executed and committed.
- If a subtask $T_i$ of $T$ commits, then all child subtasks of $T_i$ will be executed and committed.
- Each subtask $T_i$ of $T$ is executed atomically.

By simple induction, UCP guarantees that if the root step of a task commits, then the task eventually commits and is made permanent. In other words, the task cannot be "rolled back" once its root step commits. This failure semantics fact is not as peculiar as it might sound. In many applications, "aborting" a task almost never occurs. Even if a task logically fails (e.g., the funds transfer request is denied), it may still leave a permanent "footprint" in the system (i.e., the first step of the funds transfer request is committed with a database update indicating that such a request arrived and failed to logically take effect, and the first step stops short of sending out any message, and therefore the entire task terminates at the time the first step terminates). Therefore the only time a task is (physically) rolled back is when the root step of the task is cancelled before it is committed and it is not resubmitted. All other steps must eventually execute. If a non-root task is aborted and cannot be retried (e.g., a program error is encountered in the procedure which handles the step), the message which caused this step to be spawned must be routed to an error-handling step which eventually must commit. We will further contrast this failure semantics of UCP with that of sagas [GS87] or migrating transactions [KR88] in a later subsection.

UCP in general does not guarantee serializability. In some applications, it may be argued that serializability is not needed. The general subject of UCP-task semantics which is weaker than serializability is a topic for future research. In the following subsection, however, we briefly describe a more stringent class of UCP-tasks which do preserve serializability at the task level. This class of UCP-tasks is called virtual transactions.

3.3 Virtual Transaction Semantics

We describe a simple class of virtual transactions which preserve serializability.

A virtual transaction is a task which is UCP-decomposed into a two-level tree, and its UCP-execution against the other concurrent UCP-executions of tasks in the system preserves serializability at the task level. In other words, it can be verified that the UCP-execution of the tasks is equivalent to some serial execution of the same set of tasks.

The formal characterization of serializability in the UCP context is briefly described as follows. We generalize the notion of a database state to be a promised state, where a promised state is a database state plus all the persistent messages yet to be received and executed upon. A materialization of a promised state in some sequence $\lambda$ is defined to be a database state which would be realized if all the persistent messages in the promised state are received and executed in order defined by $\lambda$. Given a consistent initial database state with empty persistent pipes, a promised state is reachable if it can be the result of executing, in any sequence, the root steps and some number of the child steps of a set of 2-level UCP tasks in this initial state. These UCP tasks preserve virtual transaction semantics if the materialization of any such reachable states in any order $\lambda$ is equivalent to a state resulted from executing the same set of UCP tasks in some serial manner.

The verification of serializability in this context is relative to a system of tasks, and in general adding a new type of tasks to the system necessitates a reevaluation of this property if virtual transaction semantics are to be preserved for the tasks. A formal characterization of the serializability property of UCP-decomposition in a narrower context has been presented in [TH90].

3.4 Related Work

As discussed in the previous section, the persistent pipe abstraction is inspired by the queue abstraction proposed in [BHM90]. The notion of a queue has long existed in commercial transaction systems (e.g., see [McGr77], [Gray78] [DEC88]). The focus of [BHM90] is on fault tolerance for user input and system replies. It uses queues to offer reliable request-reply processing, and utilizes some specific features (namely, the semantics of a persistent registration operation) proposed for the queues. It is proposed also in that paper that queues can be used to implement a chained transaction model, where a user request is processed by a sequence of atomic transactions by the server(s) before providing a reply to the client. The chaining mechanism is almost identical to the execution structure of a subtask we have described in the previous subsection, except that instead of the use of a queue object, which may be residing in either the sending site or the receiving site (and even a third site), we have used the two ends of a persistent pipe, ensuring that operations on the pipes are always local. In this context, the

---

2Note that the UCP paradigm does not intrinsically offer virtual transaction semantics; it is the knowledge of the semantics of the application that enables one to verify virtual transaction semantics.
UCP mechanism can in fact be used as an implementation for enqueuing elements onto remote queues with only local operations.

The proposed unilateral commit paradigm has many similarities with current literature on extended transaction models (e.g., [KR88] [GS87] [PKH88] [FZ89] [KS90] [DHL90]). In particular, the notion of decomposing a task into a sequence of transactions is extensively studied in the context of the saga abstraction ([GS87] [KR88]). However, we have a different focus and approach in this research. The primary motivation behind our approach is to tackle the issue of whether it is feasible and desirable to reduce the use of distributed commit protocols, especially in heterogeneous systems, while still preserving certain reliability semantics. We also advocate a position which restricts the failure atomicity to be "if the first step commits, then the task will eventually commit". The saga approach focuses primarily on the notion of compensating transactions, advocating the idea that the programmer can specify, for each step, a compensating step, and the system can automatically administer the dispatching of compensating steps should any step fail in the task. Some discussions on this issue have also been offered in Section 2.3.

In [DHL90], an approach was also proposed which includes triggers into the structure of a transaction or a task. These more sophisticated structures for transactions are beyond the scope of interest in our approach. At the same time, our approach relies on a concrete implementation plan (i.e., based on persistent transmission) to link atomic steps together.

The tree-structured decomposition of a task bears resemblance to that of nested transactions ([Moss81] [Lisk85]). However, the UCP execution paradigm is very different from the nested transaction model. In the nested transaction model, a transaction is structured as a tree of subtransactions where the root is the top-level transaction. The nested transaction model enables intra-transaction parallelism to be better expressed, and provides enhanced reliability management: if a subtransaction fails (e.g., the system on which the subtransaction is executed fails,) it can be aborted by its parent without affecting other parts of the transaction tree. However, in the nested transaction model, the top-transaction together with its offsprings forms a unit of atomicity. Committing a distributed nested transaction still requires the use of two-phase commit protocols in general, and all locks are held until all subtransactions have "committed". In contrast, each step in the UCP tree is executed physically as an atomic transaction, not a subtransaction. The nested transaction model can be effective in systems where closely-coupled cooperation exists (thus all sites are willing to participate in two-phase commit) and serializability at the task level is desirable but not achievable by way of virtual transactions. The Camelot system [Spec87], for example, implements a nested transaction model when linking multiple servers together through Remote Procedure Calls ([Neb81] [BN84] [LS87]) in servicing a user "transaction". We are exploring a different niche in studying the applicability of UCP.

4 Implementation of Persistent Transmission

In this section we describe implementation of persistent transmission. Note that the mechanism has much in common with the techniques used in communication systems to preserve reliable and ordered transmission of packets; the latter, however, does not typically preserve persistence upon failures, and therefore is not sufficient for our purpose.

Let the total number of communicating sites in the network be denoted as |S|. Each site S, maintains a monotonically increasing sequence counter step-number, denoted as sn(S), that uniquely identifies the latest step committed at this site. In addition, S maintains two vectors, alpha[i] and beta[i], each vector containing |S| entries. The j-th entry of alpha[i][j] is denoted as alpha[i][j][j]. alpha[i][j] contains the sn of the step most recently executed at Si, which has sent Si a message. beta[i][j] contains the sn of the step that is executed at Si from which Si has most recently received a message.

4.1 Transmission Algorithm

The persistent pipe algorithm consists of two parts: one part is executed while the transaction is still in progress; the other executed as part of transaction commit. We start with the P.Send and P.Receive operations:

P.Send(Pipe_id, OP, argument list):
enter message into log;
end;

P.Receive(Pipe_id):
wait till Pipe_id not empty;
e = remove_next_message(Pipe_id);
  /* message e is removed from the pipe */
  enter e into log;
return (e);
end;

At commit time, the commit processing at a site Sj for a step Tj must also "commit" the persistent pipe operations:

Commit(Tj):
Tj.sn := sn(Sj)++; /* assign sn to Tj */
enter commit record (and its sn) into log;
for each msg in log of Tj for S_i do:
  include Tj.sn and alpha[i][j] into msg;
  alpha[j][i] := Tj.sn;
SEND(msg); /* SEND actually sends msg */
end;

Note that when the transaction commits, so have the P.Send and P.Receive operations associated with the transaction.

Persistent transmission illustrates the usefulness of end-to-end protocol design [SRC84]. Our implementation relies on message logging, and bears some resemblance to the approach suggested in [SY89].

290
Should the transaction abort, the undo procedure will include returning any "P.Received" message back to the pipe where it came from, and no message will be sent out.

Upon receiving a message m at site S, from site S, site S, executes the RECEIVE procedure outlined below. m carries the alpha element, denoted as m.alpha, and the sn of the step which sent this message, denoted as m.sn. If there is any missing message between S, and S, as indicated by a comparison between m.alpha and beta[j][i], S, asks site S, to retransmit the missing messages.

```c
RECEIVE(m,i): /* receive a msg m from S, i*/
if m.alpha > beta[j][i] { /* some messages missing */
    request for retransmission;
    exit;
} else { /* no messages missing */
    enter m into log:
    beta[j][i] = m.sn;
    insert m into appropriate pipe buffer;
}
end;
```

It is assumed that the procedure is executed atomically at the receiver site. Note that there is no need to force out the log entries immediately at the receiving site.

If a site S, receives a retransmission request from S, S, processes the request by retrieving missing messages from its log. We omit the details of this procedure here.

### 4.2 Recovery Algorithm

We assume that the local fragment of the database state at S, along with alpha[i] and beta[i] is periodically checkpointed and saved in stable storage. When a failed site S, recovers, S, first restores the last checkpoint and then rolls forward the local database state, the pipe buffer, alpha[i], and beta[i], based on events saved in the local log. This completes its local recovery, and S, is now operational, i.e., S, can now start processing transactions.

However, there may be lost persistent messages between S, and other sites when S, failed. Although the normal processing of persistent transmission protocol will discover these missing messages in due course, it would speed up the detection of such missing messages if S, at this point takes the initiative to broadcast, in null messages, its alpha vector to other sites that are up, and solicit from all sites some messages, null or non-null. The normal procedure for persistent message processing will ensure that all missing messages are recovered. The recovery procedure is outlined below:

```c
Recovery(s):
    restore checkpointed token state,
    alpha[s] and beta[s];
    for each entry in local log do:
        if log entry is database operation of committed step { perform database operation; }
        if log entry is message m received { beta[s][m.sender] = m.sn;
            store m in appropriate pipe buffer;
        } if log entry is message m P.Received by a committed step {
            remove m from pipe buffer
        } if log entry is message m P.Sent by a committed step sn to site S_j {
            alpha[s][j] = sn;
            sn[s] = sn; /* restore sn[s]; */
        }
    /* site s becomes operational at this point */
    for each remote site S, j in S do:
        formulate a null message mj;
        append sn[s] and alpha[s][j] to mj;
        SEND(mj);
        /* check that site j is up-to-date */
        /* j will ask for retransmission if needed */
    end;
```

The procedures can handle link failure easily as well. If a link between S, and S, has failed, messages that are lost during the failure can be detected by the RECEIVE procedure when subsequent messages from S, to S, are received after the repair. However, the responsiveness of this detection scheme depends on the frequency of persistent messages between S, and S,. To remedy this, each site should periodically send null messages using its alpha vector to ensure timely retransmission.

### 5 Example Applications of UCP

In this section we examine two applications which can use UCP and persistent pipes to reduce distributed transaction overhead. The first case is drawn from the technique of Data Value Partitioning. The second is to support the implementation of directory operations in distributed data migration systems. These applications also serve to illustrate the need for a better integration between communication and transaction systems in a distributed environment.

#### 5.1 Data Value Partitioning

The Data-value Partitioning approach [SS90] is a method to reduce inter-site coordination on shared data. This method can be applied to data that can be partitioned into smaller pieces such that the pieces themselves can be regarded as instances of the original data items. Example of such data is the number of seats in airline reservation systems. Each of the constituent values is stored at different sites. Every transaction that is to be run in the system is executed at one single site by using data values available at the same site. Only in the event that the locally available data is inadequate for the transaction to execute, the site requests other sites for the values stored in them. If the responses from the remote sites fail to arrive for any reason within a reasonable amount of time, the transaction is aborted.

If a site is requested to transfer data values to another site, the data value transfer task involves at least a sender site and a receiver site, and therefore is a “multi-site” task.
Instead of running the data value transfer task as a single distributed transaction which would involve two-phase commit, we can make use of persistent pipes and execute the data value transfer task as two unilaterally committed transactions.

We illustrate the DVP scheme using two sites S1 and S2. Each site has some data values. A user transaction T2 starts at site S2, and sends a non-persistent message to site S1. Site S1 starts a transaction T1, decrements some data value at S1, and sends some value to site S2 through persistent pipe, and commits T1. T2 receives the data value from the persistent pipe, increments some data value at S2, and commits. These two transactions together implement a virtual DVP transaction: it consists of two atomic steps: the “DVP Trx Init Step” is executed at S1, and the “DVP Trx Remote Step” is the step occurring at S2. The two steps are connected through a persistent pipe.

5.2 Data Migration with Token Transactions

The second example application of virtual transactions is in the management of a distributed shared virtual memory (DSVM) system, as proposed and studied in [TH90] [TH90a]. A DSVM system must implement some form of page locating method in order to service network page faults. This facility must be able to correctly locate the most up-to-date version of a page as the page migrates in the system and as new versions are created or cached. We use the term token table to refer to a site’s knowledge as to what access rights it has on data pages it has cached, and the term locating table for the information on how to get to the sites which have the most up-to-date data pages. Together, they form the token directory.

[TH90] [TH90a] propose to make use of the transaction abstraction to manage dynamic changes in the directory. The idea is to regard data migration activity as a special kind of transaction, called a token transaction that updates the token directory. By updating the token directory in a transaction-atomic fashion, the token directory state will retain consistency in face of failures. This consistency can be exploited to allow the recovery algorithms to be simplified and made more efficient. However, every token transaction updates token directory distributed among several sites, and is a distributed transaction. An examination of the behavior of the token transactions reveals that a token transaction be run as a virtual transaction using UCP, where a token transaction is broken into steps, each step executed as a single-site “physical” transaction.

Figure 2 illustrates the idea. In this example, site S1 is about to transfer an owner token to site S2, and to inform the locator site S-locator to update its locating table to reflect this transition. Site S1 can execute a token transaction as a distributed transaction, involving 3 sites in its commit processing. However, the token transaction can be decomposed into three steps S1 executes the initiating step, which removes the owner token from its token table. Sites S2 and S-locator, upon receiving the child step messages from the persistent pipe, complete the remaining steps of the token transaction. Therefore a token transaction becomes a virtual transaction, consisting of three atomic transactions connected through persistent pipes.

6 Summary

We have presented an alternative paradigm for distributed transaction processing which reduces the need to use the two-phase commit protocol. We discussed some serious drawbacks of the commit protocol and argued that there are applications where the traditional approach is too restrictive. We propose an alternative based on unilateral commit and persistent transmission. UCP is especially attractive since it relies on a site's ability to execute conventional "flat" local transactions and does not require additional capabilities such as ability to execute nested transactions.

In this paper we have only presented a preliminary analysis of the execution semantics of the UCP tasks. The
exact characterization of virtual transaction semantics has been offered only for a limited class of UCP decompositions, namely, the class where the decomposition results in a two-level tree of subtasks. Generalization of these concepts are desirable. Other applications should be identified. Weaker semantics of UCP must also be studied. In particular, we would like to identify classes of UCP-tasks for which general serializability is not necessary, yet for which certain constraints still need to be satisfied. Precise specification and understanding of such weaker semantics can greatly enhance the applicability of UCP.

Additional services will also be studied. For example, the task trace service, which, given a task id, traces the steps that the task has gone through, is desirable and poses technical challenges in its implementation. As another example, when it is detected that a site has failed for some time, the contents of the persistent pipes destined for that site may need to be re-directed to some local contingency service routines. The primitives for specifying such contingency and its implementation need to be examined.

References


