A Critical Analysis of the Transaction Internet Protocol

Tim Kempster* , Colin Stirling, Peter Thanisch

Technical Report ICSA-46-99

DIVISION *of* INFORMATICS

August 5, 1999

Abstract

Recently, the Transaction Internet Protocol (TIP) became an Internet standard. TIP is intended to facilitate electronic commerce transactions in which the customer may need to acquire a package of goods and services from several different enterprises, each enterprise having its own autonomous transaction processing system. The TIP protocol attempts to ensure the atomicity of the transaction: either all enterprises commit the transaction or all enterprises abort it.

As protocols go, TIP has some attractive features and some not so attractive features, so groups of enterprises planning to use TIP should be aware of the potential pitfalls. In this report, we provide a formal framework in which to reason about the behavioural properties and related optimizations of TIP. Within this framework we analyse problems for which TIP is intended to be a solution and discuss how TIP fits in to the broader transaction processing and e-commerce picture.

*Rm 2602, JCMB, Kings Buildings, University of Edinburgh, Mayfield Road, Edinburgh, EH9 3JZ, Scotland Tel +44 131 650 5139, Fax +44 131 667 7209. This research was supported by EPSRC grant GR/L74798.

Contents

1 Introduction

The spectacular growth of Internet electronic commerce and the hype that surrounds it is well documented[11]. The bulk of these transactions are relatively simple, involving a customer and a single enterprise. A good example is the very successful online bookshops.

When two or more enterprises are involved in providing a *package* of goods or services, electronic commerce becomes considerably more complex. Not surprisingly, where such arrangements do exist, they are based on long-term bilateral relationships between a fixed sets of enterprises. In other words, so far, Internet commerce has merely been a more convenient way of doing shopping, that is "visiting" one enterprise at a time to buy goods or services. However, the Internet has the potential to allow a radically different approach to the buying and selling of goods and services by allowing enterprises to come together on an ad hoc basis to provide, collectively, a package that can be acquired by the customer in "one stop". By making this complexity transparent to the customer, the enterprises can appear to be a single, virtual, enterprise. Such an arrangement can be very transient, possibly exploiting special market conditions.

Whenever a transaction is distributed between several autonomous transaction processing systems, there immediately arises the problem of ensuring the "atomicity" of the transaction. For example, making sure the customer does not end up with the hotel booking but no flight, in the classic holiday booking example. The solution to this problem is to use a commit protocol [4]. However, the nature of these virtual enterprises, combined with the way in which the customer interactively puts the package together, create the need for a lightweight, flexible approach to commit processing.

TIP (Transaction Internet Protocol) is a distributed commit protocol published by Microsoft and Tandem as an Internet draft. It defines an approach that makes the commit protocol independent of the communication protocol used. It is intended that TIP will be adopted by the Internet Engineering Task Force (IETF) as an Internet standard. In any case its adoption by Microsoft and Tandem will likely affirm it as a de facto standard.

Potential implementers of TIP and application programmers using TIP services need to be aware of the problems which TIP solves and the problems that it has been deliberately designed to side step. The omission of discussion of these potential problems means that an implementer or user of TIP services is forced with a set of choices as to how TIP might be incorporated within their overall e-commerce solution. In this report, we provide a framework in which to analyse behavioural properties of TIP. This allows us to identify potential problem areas, which arise from the implementers' set of choices.

The report is structured as follows. The next section describes the way parties are enlisted into a TIP transaction. Section 3 describes how the TIP protocol ensures a consistent commit or abort outcome amongst the parties involved. Section 4 presents a more formal model of the protocol. Section 5 extends the model to take into account failure. In Section 6 we use our model to prove that TIP transactions are atomic. Section 7 analyses some optimisations present in TIP. Section 8 discusses issues of trust and security. Section 9 details some potential pitfalls in TIP transaction processing. Section 10 proposes a centralised variant of TIP which solves some of these pitfalls. Finally, section 11 offers concluding remarks and future work.

2 Growing A TIP Transaction Tree

A transaction manager (TM) provides transaction services to applications. Typically, each application participating in a distributed transaction will invoke the services of its local TM. Although each TM will manage many transactions simultaneously, the states of each transaction do not affect each other[1]. Therefore we focus on the life cycle of a single transaction and think of the TMs at each of the sites at which the transaction executes as managing only this transaction. A TM, therefore should be thought of as the state of the sub-transaction held on behalf of the local application.

[1] One transaction can affect the behaviour of another indirectly through locks it might have acquired on resources.

TIP makes use of transaction identifiers (TIDs). A TID labels and locates a transaction at a particular TM. TIP specifies a TID as the concatenation of the IP address of a machine where the TM is located, a port number to connect to the TM process at that machine and finally the local transaction identifier assigned to the transaction by that TM. TIP calls these TIDs TIP URLs. During the lifetime of a transaction TIDs are Internet wide unique identifiers.

2.1 Push

Figure 1 below describes the first enlistment method, known as PUSH. Before any TIP commands can take place, an application must open a TIP connection to its local TM. This is achieved with a `tip_open` call (steps 1 and 2).

If application A, wishes to enlist a remote application B then A contacts its local TM, X and requests that the transaction be pushed to B's TM Y using `tip_push`. Application A must supply the whereabouts of TM Y with this request. X then contacts Y and pushes the transaction (steps 5 and 6)[2]. Y creates a new sub-transaction and returns a TID to X for this new sub-transaction. After the push the connection enters the enlisted state with X the superior and Y the subordinate. X now notifies application A that the push has taken place (step 7), supplying A with the TID obtained from Y. Whenever application A makes a request to application B to carry out some work on the transaction the request carries with it the TID returned in step 7. Application B can now register this work with its local TM Y using the TID. This request, response and register dialogue is shown in steps 8–11.

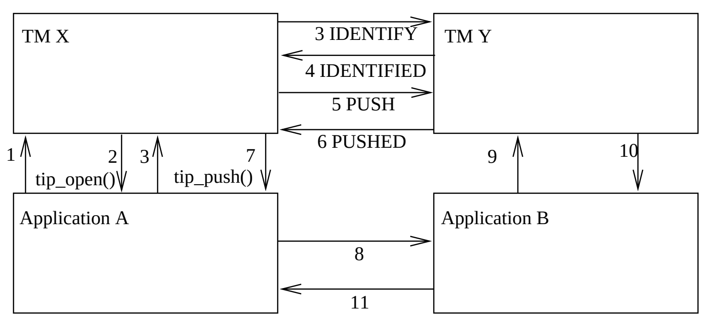

Figure 1: A Push

2.2 Pull

The second method by which an application A can enlist a subordinate application B in a transaction is called Pull see Figure 2.

Application A opens a connection to the TIP gateway by contacting its local TM, X (step 1) and receives a TID for the transaction (step 2). This TID not only contains the ID of the transaction at X but also the means by which X can be contacted (an IP address and port number).

If an application B wishes to carry out some work on the transaction, as a subordinate of A, it can ask its TM Y to pull into A's TM X. In order to do this it must first obtain, by some means, the TID returned to A[3] in step 2. The passing of the TID is represented in step 3.

Application B now presents Y with the TID and asks Y to PULL into the transaction, using `tip_pull`. TM Y uses the TID to contact X and pulls into the transaction, possibly setting up a connection first (steps 5–8) and returns control to application B steps 7,8 and 9. An asynchronous version `tip_pull_async` exists, where control is returned to the application straight away and then

[2] If no connection exists between X and Y a connection is first established, (steps 3 and 4).

[3] In most cases A will pass the TID along with a request for B to carry out some work on the transaction, but in general any method whereby B receives the TID is possible.

the application notified later when the PULL completes. Application B may now contact A to inform A that it is enlisted (step 11).

One of the main differences between push and pull is that in pull a TID is made available in some way to a subordinate which then *may* pull into the transaction. Any application[4] that acquires the TID *may* pull into the transaction whereas if a TM pushes to another TM the enlistment is directed to only that TM.

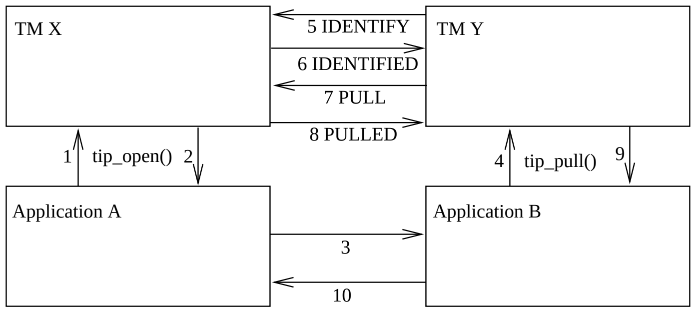

Figure 2: A Pull

2.3 Enlistment using Push and Pull

Once an application has registered the work it carries out as part of a transaction with its local TM, transaction semantics are enforced. At some point the initiator application will try to terminate the whole transaction. All it need do is contact its local TM to do this and the TIP protocol will then ensure that all TMs decide a consistent abort or commit outcome. Conversely, if a local application unilaterally decides to abort its part of the transaction it will contact its local TM. This TM will then use the TIP protocol to ensure all TMs and their associated local applications are informed of the global abort decision. Thus once enlisted in a transaction an application can delegate all responsibilities of commit processing to its TM. This significantly simplifies the application programmers task.

In the example of Figure 3, a TM R pushes a transaction to two other applications B and C. In order for B to fulfill the requests of A, B further pushes the transaction to D. In order for C to fulfill the requests of A it requests E and G to pull into the transaction and it pushes the transaction to F.

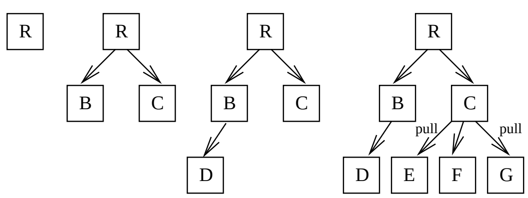

Figure 3: Growing a transaction tree in TIP. By default transaction are pushed unless otherwise stated.

Notice that the growing of the transaction tree and thus the shape of the tree is completely controlled by the local applications cooperating in the transaction. This means the application that originated the transaction only has indirect control over the shape of the tree in terms of the identities and number of subordinates at each level. The decision to enlist a TM is not made in the TIP protocol but instead by the applications.

[4] We will see in a later section how this can be made more secure.

It is possible for the same transaction to arrive at a TM via different paths. If this happens a TM has two or more superiors for the same transaction. In TIP there is no way to know that the two enlistment actions refer to the same transaction[5]. We therefore model this situation by viewing subordinate TMs that are enlisted multiple times as separate TMs, one for each enlistment. In Figure 3 TMs F and D may reside on the same physical site in the same TM implementation but we view them as distinct TMs. For this reason transaction enlistment always gives rise to a tree structure[6] : each TM has at most one superior for any given transaction.

3 Terminating the Transaction Tree

In the previous section we saw how TIP enlists TMs into a transaction. We now give an informal account of how TMs communicate in order to arrive at a consistent global commit or abort decision at each TM in the transaction. In the next section we will give a more precise account of this process.

In contrast to the enlistment process, local applications play little part in the termination of a transaction. The applications interact in the following ways. When the initiator application decides to terminate the transaction it contacts its local TM. The TM then executes the TIP protocol to determine the global outcome of the transaction. Each local TM eventually discovers the outcome and passes this on to its local application.

If any enlisted application wishes to abort the transaction it may do so and the TIP protocol will ensure that the transaction is aborted at each enlisted TM.

We now describe the steps taken when the initiator application wishes to commit the transaction. The initiator application contacts its local TM to start the commit process. A two-phase commit is entered into between every superior and each subordinate. The steps of this two-phase commit are as follows

- If a local application can commit the transaction a PREPARE command is sent to each of its subordinates. Each subordinate replies PREPARED if it is prepared to make its local work permanent, the state of the connection between superior and subordinate then changes to prepared. Before replying PREPARED a subordinate TM must carry out a two-phase commit with each of its own subordinate TMs. Thus a PREPARED response is not propagated up until all subordinate TMs have replied PREPARED.
- If the local application cannot commit the transaction, its TM responds with ABORTED to any PREPARE request and also sends an ABORT message to all its subordinates. If a TM receives an ABORTED response from any subordinate or an ABORT message from its superior, it sends an ABORT to all its non-aborted subordinates and replies ABORTED to its superior. In TIP a TM cannot ABORT from an enlisted state until it is asked to PREPARE or ABORT[7].
- Once all the subordinate TMs of the initiator TM, R, have replied PREPARED, R moves into a committed state and issues a COMMIT command to each of its subordinate TMs. Once a TM receives a COMMIT command it moves to a committed state and propagates the commit command down to each of its subordinates and replies COMMITTED once all subordinates have replied COMMITTED.

4 Modeling TIP Protocol Execution

The execution of a TIP protocol can be modeled as a tree. The vertices of the tree represent the TMs. An arc from vertex X to Y, models a superior to subordinate connection between the TMs

[5] This is because the format of a TID is defined by each TM and so there is no way to compare TIDs and deduce that they pertain to the same transaction.

[6] We chose to model TIP transactions as a tree structure rather than a directed graph, since the TIP specification was ambiguous in this respect.

[7] The application can however still abort releasing any resources once it has informed its TM of the abort decision.

X and Y. The end points of the arcs are "coloured" to represent the states of the connection at each TM. The different state colours are initial **i**, enlisted **e**, prepared **p**, committed **c** and aborted **a**[8]. Figure 4 shows a tree. The initiator application has issued a command to start to commit the transaction. The subtree with TMs B and D, have connections that have entered their **p** state.

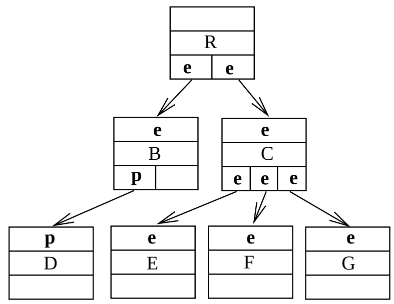

Figure 4: A TIP Transaction tree

We now give a set of rules which manipulate the coloured trees of our TIP transactions. The rules provide a transition semantics for all possible TIP transaction trees and describe the possible termination sequences of the transactions.

Each rule manipulates an arc representing the state of a connection in a TIP transaction tree. We write $X(x) \to Y(y)$ if an arc exists between vertices X and Y and the arc is coloured x at X and y at Y. Each rule transforms these arcs provided certain conditions hold. We write rules as

$$\textbf{Name}\ \frac{X(x) \to Y(y)}{X(x') \to Y(y')}\ \text{CONDITION}$$

meaning rule **Name** transforms a connection of type $X(x) \to Y(y)$ to $X(x') \to Y(y')$ provided condition CONDITION holds. In most of our rules $x = y$ and $x' = y'$.

We define the multi-set $OUT(X) = \{x \mid \exists Y(y), X(x) \to Y(y)\}$, the multi-set of states of all outbound connection from X. We define the singleton or empty set $IN(X) = \{x \mid \exists Y(y), Y(y) \to X(x)\}$, to be the state of the inbound connection, or empty if there is no inbound connection.

Applications must ensure that a TM does not enlist new TMs after it has entered its prepared state or if it has decided the outcome of the transaction. We will return to this point later. For now it will be useful to define

$$\text{grown(X)} \stackrel{def}{=} (\mathbf{p} \in IN(X)) \vee (\mathbf{a}, \mathbf{c} \in IN(X) \cup OUT(X))$$

Applications must also ensure that if a TM X enlists another TM Y in a superior subordinate relationship then either X is the initiator application's TM, i.e. $X = R$, or X has an enlisted superior[9]. This can be expressed with the predicate

$$\text{enlisted(X)} \stackrel{def}{=} (\mathbf{e} = IN(X)) \vee (X = R)$$

The first three rules grow the transaction tree enlisting TMs.

$$\textbf{Connect}\ \frac{Y\ is\ new}{X(\mathbf{i}) \to Y(\mathbf{i})}$$

$$\textbf{Push}\ \frac{X(\mathbf{i}) \to Y(\mathbf{i})}{X(\mathbf{e}) \to Y(\mathbf{e})}\ \neg\text{grown(X)} \wedge \text{enlisted(X)}$$

[8]The TIP specification[10] groups the initial, committed, and aborted states into a single state they call idle. A connection starts in an idle state and once aborted or committed it returns to idle. We use the states committed, aborted and initial in order to differentiate between a connection that has terminated and one that is new.

[9]It is possible to envisage an enlistment scheme where TMs enlist one another into a forest of trees, each tree having a different root TM. Eventually these trees merge into a single tree with one root. Our rules require modification for this more general case.

$$\textbf{Pull}\ \frac{X(\mathbf{i}) \rightarrow Y(\mathbf{i})}{X(\mathbf{e}) \leftarrow Y(\mathbf{e})}\ \neg\text{grown(Y)} \wedge \text{enlisted(Y)}$$

The **Connect** rule establishes a new arc between two TMs. This models the situation where TIP creates a new connection between two TMs or reuses an old connection that has terminated. The **Push** and **Pull** rules move initial connections into enlisted connections.

To move a connection to a prepared state we require all outbound connections to be in a prepared state. We can write this rule as follows.

$$\textbf{Prepare}\ \frac{X(\mathbf{e}) \rightarrow Y(\mathbf{e})}{X(\mathbf{p}) \rightarrow Y(\mathbf{p})}\ \forall z \in OUT(Y),\ z = \mathbf{p}$$

When a TM is sent a PREPARE message it might respond with an ABORTED message. This might be because the local application cannot guarantee to do the work asked of it. The rule for this is given below. It might seem that the lack of a condition for this rule means that a TM can spontaneously abort from an enlisted state. In normal operation a TM will generally try to commit but we must allow it to abort if it needs to.

$$\textbf{Abort I}\ \frac{X(\mathbf{e}) \rightarrow Y(\mathbf{e})}{X(\mathbf{a}) \rightarrow Y(\mathbf{a})}$$

If a PREPARE command causes a TM X to abort then this abort decision must be propagated to all prepared subordinate and superior TMs of X. The rule for this is given below.

$$\textbf{Abort II}\ \frac{X(\mathbf{p}) \rightarrow Y(\mathbf{p})}{X(\mathbf{a}) \rightarrow Y(\mathbf{a})}\ \exists \mathbf{a} \in IN(X) \cup OUT(X)$$

Once all TMs are prepared from the bottom up the transaction can commit. The commit rule is given below, it propagates the commit decision down the tree.

$$\textbf{Commit}\ \frac{X(\mathbf{p}) \rightarrow Y(\mathbf{p})}{X(\mathbf{c}) \rightarrow Y(\mathbf{c})}\ (\forall x \in IN(X),\ x = \mathbf{c}) \wedge (\forall x \in OUT(X),\ x = \mathbf{c} \vee x = \mathbf{p})$$

The example in figure 5 below shows the start of a possible execution of the TIP protocol.

5 Failure, Recovery and Logging

In our model a rule changes the state of a TIP TM connection. This requires a state change at both the TIP TMs associated with this connection. Although modeled as an atomic event a handshake between the TMs must take place in order for both TMs to change state. We can break down the event of transforming $X(x) \rightarrow Y(x)$ to $X(x') \rightarrow Y(x')$ into two steps. Firstly, X sends a messages along the connection to Y requesting it to move to state x'. When Y receives this it moves to state x' and then sends an acknowledgement back. When X receives the acknowledgement it too moves to state x'. We call this a *forward handshake*. A *backward handshake* is also possible where the sender changes state before the message is sent. All our rules employ a forward handshake with the exception of **Commit**.

Failures may occur when a TM at one end of a connection crashes or the connection between two TMs fails. TIP does not differentiate between a lost connection due to communications failure or a lost connection because the remote end of a connection failed. We model failure as a loss of connection and defer the discussion of logging and recovery of crashed TMs until later.

5.1 Connection Failure

Failure can happen at any time between any of the TMs in the system. When a TM detects a failure on a connection it carries out steps depending on the last known state of the connection before the failure. We cannot always assume that the state of the connection at the remote TM is

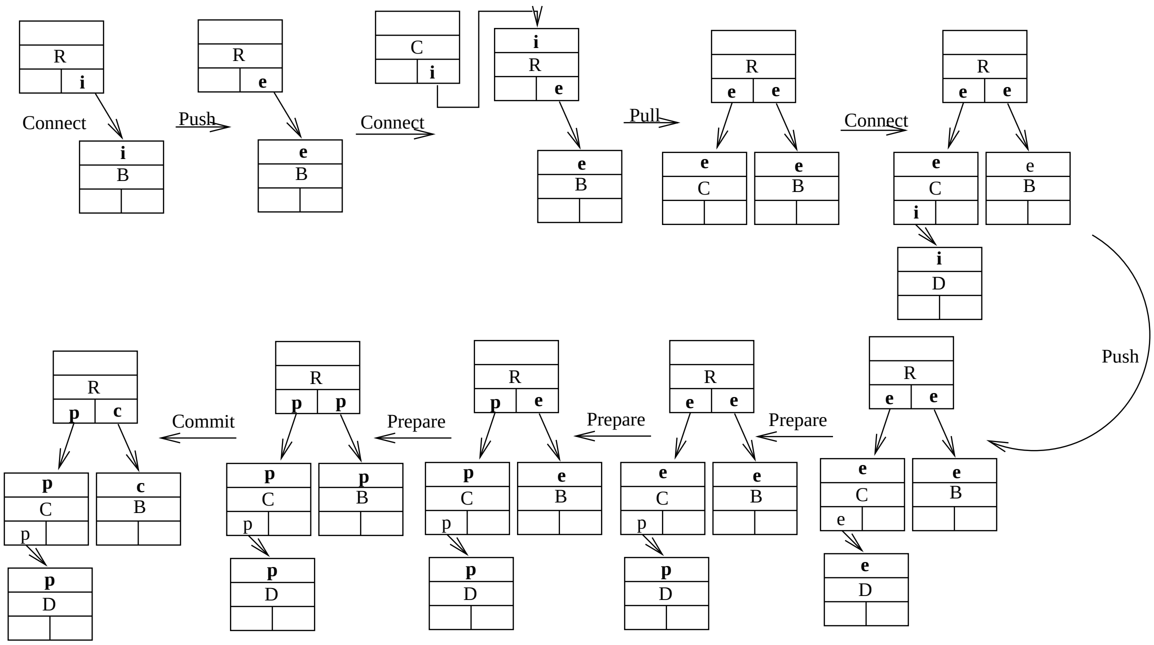

Figure 5: An execution of TIP

the same as the state of the connection at the TM where the failure is detected. This is because the failure may have happened during the handshake process when the two TMs are changing state. In the case of a forward handshake the subordinate might be in a more advanced state than the superior. In TIP, handshakes are of the forward variety unless otherwise stated.

To model connection failure, we introduce some new notation. We say $X(x) - \; \rightarrow Y(y)$ if a previously established connection between X in state x and Y in state y has failed. We can express the failure of a connection in the following rule where x ands y range over the states **i**, **e**, **p**, **c** and **a**.

$$\textbf{Failure}\ \frac{X(x) \longrightarrow Y(y)}{X(x) - \; \rightarrow Y(y)}$$

In TIP the actions taken by a TM after failure depend on the state of the connection at the TM prior to the failure. If the state of the connection is **i**, **c** or **a** no action need be taken[10] . If a TM X is in the **e** state the transaction is aborted. We model this using the rule

$$\textbf{EAbort I}\ \frac{X(\mathbf{e}) - \; \rightarrow Y(\alpha)}{X(\mathbf{a}) - \; \rightarrow Y(\alpha)}\ \textbf{EAbort II}\ \frac{X(\beta) - \; \rightarrow Y(\mathbf{e})}{X(\beta) - \; \rightarrow Y(\mathbf{a})}$$

These two rules model the case where a superior aborts an enlisted connection to a subordinate or vice versa, when it detects a connection failure to the remote TM. The exact state at the remote end of the connection is unknown so we denote it α which could be **e**, **p** or **a** in the first rule and β which could be **i**, **e** or **a** in the second rule.

If the superior TM connection failed in state **p** then it should try to reconnect to its subordinate[11]. After recovering the connection the superior sends a RECONNECT message to the subordinate which responds with a RECONNECTED message and both TMs re-enter the **p** state.

If a subordinate TM connection failed in state **p**, on recovery it periodically attempts to contact its superior in order to resolve the outcome of the transaction. The subordinate issues a

[10] Although the TIP specification states that no action need be taken if a connection fails in the **c** state this is not quite true. In order for the superior to be able to "forget" the transaction and deallocate the memory associated with it, it must be sure that all its subordinates have committed. A recovering subordinate in the **c** state may need to inform its superior that it has committed

[11] In fact, if the superior TM is going to abort the transaction reconnection is optional.

QUERY command. The superior then attempts a RECONNECT/RECONNECTED handshake re-establishing the connection in the **p** state, or else, if the superior had aborted the transaction it replies with QUERYNOTFOUND moving the connection to the the **a** state. We model this behaviour with the following three rules.

$$\textbf{Rec}\ \frac{X(\mathbf{p}) - \rightarrow Y(\mathbf{p})}{X(\mathbf{p}) \rightarrow Y(\mathbf{p})}$$

$$\textbf{Query I}\ \frac{X(\mathbf{c}) - \rightarrow Y(\mathbf{p})}{X(\mathbf{c}) \rightarrow Y(\mathbf{c})}$$

$$\textbf{Query II}\ \frac{X(\mathbf{a}) - \rightarrow Y(\mathbf{p})}{X(\mathbf{a}) \rightarrow Y(\mathbf{a})}$$

The **Rec** rule models the behaviour of either the subordinate or the superior reconnecting. The **Query I** rule models the case where the subordinate queries the superior and finds out the transaction has committed. The **Query II** rule models the case where the subordinate queries the superior and finds out that the transaction has aborted. Note a subordinate will only attempt to query its superior when it is in the **p** state.

Handshakes in TIP are based on half duplex communications. Once a message is sent along a channel the channel is switched to listen mode. Microsoft's first implementation of the TIP protocol uncovered some problems where in a certain failure sequence both ends of a channel could be switched to listen mode causing the channel to deadlock.

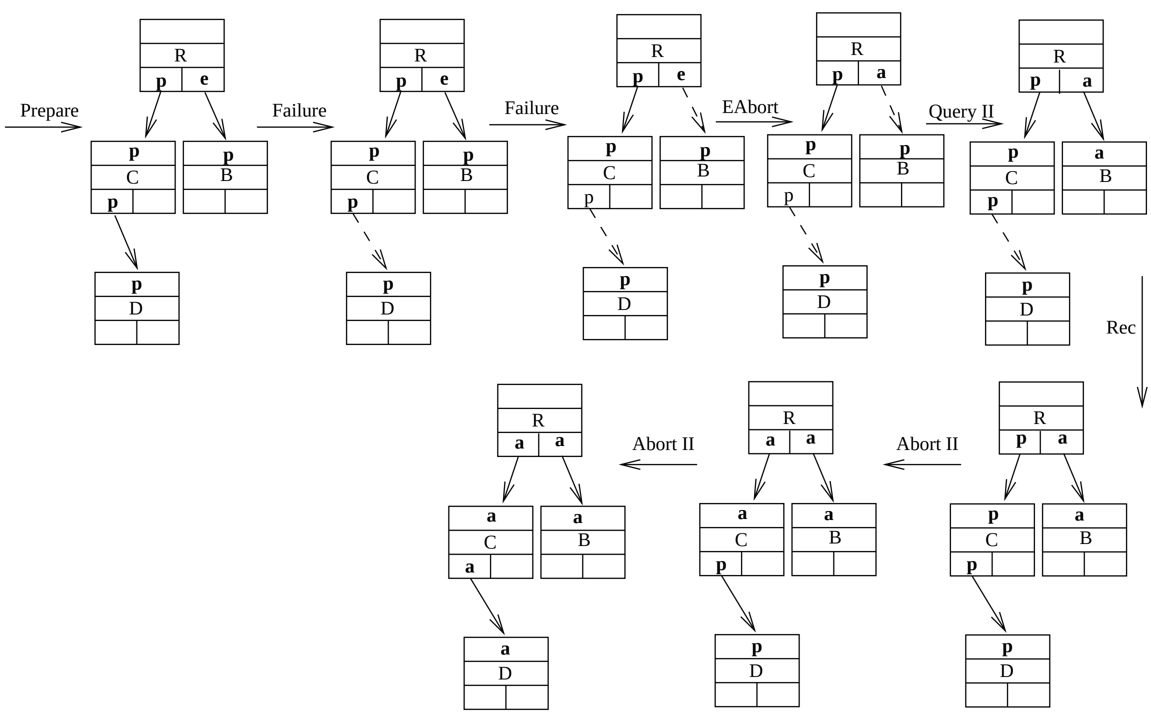

Figure 6: Failure and recovery in TIP

The example of Figure 6 describes the execution of the TIP protocol when two connection failures happen. The first connection lost is $C(\mathbf{p}) \rightarrow D(\mathbf{p})$ and the second failure breaks the connection $R(\mathbf{e}) \rightarrow B(\mathbf{p})$ half way through the handshake between R and B which would have moved that connection to a **p** state at both ends. This leaves the connections in $C(\mathbf{p}) - \rightarrow D(\mathbf{p})$ and $R(\mathbf{e}) - \rightarrow B(\mathbf{p})$ respectively. When R detects the failure of its connection with B it moves its end of the failed connection to state **a**, the **EAbort** rule. B queries R to try to re-establish the connection and receives a QUERYNOTFOUND message moving the connection to $R(\mathbf{a}) \rightarrow B(\mathbf{a})$. Meanwhile C issues a RECONNECT message to D which replies RECONNECTED re-establishing the $C(\mathbf{p}) \rightarrow D(\mathbf{p})$ connection, the **REC** rule. Two applications of the **Abort II** rule then propagates the abort decision to all TMs.

5.2 Crash Failure and Logging

Previously we examined the actions taken by TMs when they loose connections with one another. One cause of a loss of connection might be the failure of a TM at either end of the connection. We have seen that upon recovery a TM may be required to take actions to reestablish the connection depending on the state at its end of the connection before the crash.

If failure in a state, $\mathbf{s}$, requires a TM to take actions upon recovery, the TM is required to force write a log record to stable storage before entering state $\mathbf{s}$[12]. Forced log write are expensive operations and so optimizations have been proposed to reduce the number a protocol requires. TIP implements the popular *presumed abort* optimization[12]. In this scheme, upon recovery, the presumption is that a transaction should be aborted unless there is a log entry to indicate otherwise. This requires a forced log write in the following cases.

1. If a TM receives a PREPARE message from its superior it sends a PREPARE messages to each subordinate it might have. Once they have all replied PREPARED it force writes a `prepare` record and replies PREPARED to its superior. On recovery from a crash if a `prepare` log record exists, the subordinate will reestablish its connection with its superior and determine the transaction outcome. In the case that a TM and all its subordinates are read-only it can reply READONLY and no log record need be written.

2. If a TM receives a COMMIT message from its superior, or in the case that the TM is at the root of the tree and all subordinates have replied PREPARED, a `commit` record is force written and then COMMIT messages are sent to all subordinates[13]. By replying COMMITTED a subordinate promises that it will not request the outcome of the transaction from its superior. Once all subordinate TMs have replied COMMITTED a TM may in turn reply COMMITED to its superior allowing the superior to safely "forget" about the transaction by logging an `end` record. This need not be force written.

Figure 7 details the required logging activity in TIP. When transaction trees have a depth greater than one level, as they may do in TIP, logging at each level presents quite a large overhead compared to a flattened transaction tree[12]. Although not specified as part of TIP, Lampson and Lomet[8] provide a technique to further reduce logging overheads.
Message latencies are likely to be much higher in Internet transaction processing than in traditional, tightly clustered transaction processing environments. Thus by Amdahl's law[1], relative savings by reducing logging costs in the two environments will be lower for Internet transaction processing.

6 Atomicity of TIP

In the sequel let w, x, y, z range over states connections may take.

Definition 1 A *directed path* $X_1(x_1) \Rightarrow X_n(x_n)$ exists in a transaction tree if there exists arcs

$$X_1(x_1) \rightsquigarrow X_2(y_1), X_2(x_2) \rightsquigarrow X_2(y_2), \ldots, X_{n-1}(x_{n-1}) \rightsquigarrow X_n(y_{n-1}).$$

Where $\rightsquigarrow$ is $\longrightarrow$ or $- \rightarrow$. The *length* of the directed path is equal to the number of arcs on the path.

[12]The absence of a record can also be used to determine which action to take.

[13]In fact, once a `commit` record has been written a TM can reply COMMITED to its superior.

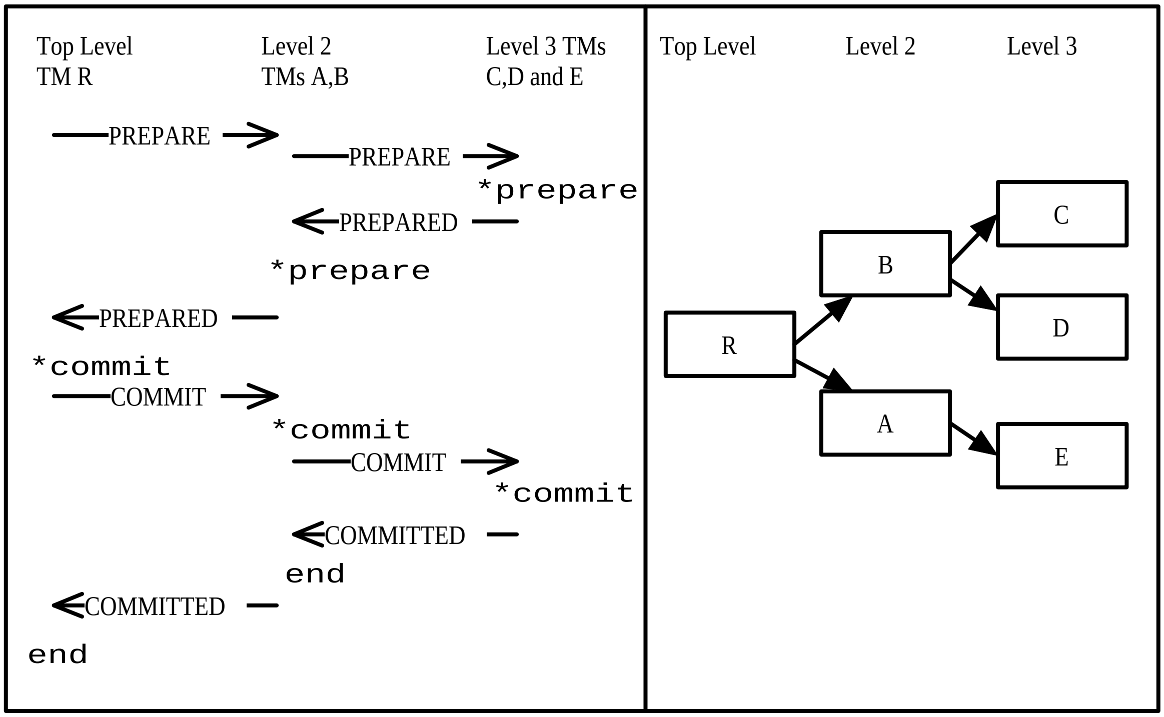

Figure 7: Logging in TIP

In the following we say a TM X first commits when $IN(Y) = \{\mathbf{c}\}$ first holds for any subordinate Y of X. We also say a TM X has prepared once it has received a PREPARED message from each of its subordinates i.e. $\forall z \in OUT(X)$, $z = \mathbf{p}$.

Lemma 1 *TM R is the first to commit.*

Proof: By the condition of the commit rule, **Commit**, and the fact that $IN(X) = \emptyset \Rightarrow X = R$.

□

Lemma 2 *Each TM prepares before all its ancestors.*

Proof: We must prove that, if $\exists x \in OUT(X)$, $x = \mathbf{p}$ and $X(\mathbf{p}) \Rightarrow Y(y)$ then $\forall z \in OUT(Y)$, $z = \mathbf{p}$. Our method of proof is by induction on the length of the path $X(\mathbf{p}) \Rightarrow Y(y)$.

Base Case: Y is a child of X. The **Prepare** rule must have created the connection $X(\mathbf{p}) \rightarrow Y(\mathbf{p})$[14] and a condition of that rule is $\forall x \in OUT(Y)$, $x = \mathbf{p}$.

Induction: Let $X(\mathbf{p}) \Rightarrow Z(\mathbf{p})$ be a directed path of length n. Thus by the inductive hypothesis $\forall w \in OUT(Z)$, $w = \mathbf{p}$. Extend this path with any connection $Z(z) \rightarrow Y(y)$. By the induction hypothesis $z = \mathbf{p}$ and $Z(\mathbf{p}) \rightarrow Y(\mathbf{p})$ must have been created by the **Prepare** rule so $\forall x \in OUT(Y)$, $x = \mathbf{p}$ holds.

It might appear that the above proof technique is invalid because the tree is dynamically growing as new TMs are enlisted. However, once a TM has voted to prepare it may not enlist any other TMs. This means that the inductive hypothesis will continue to hold for paths of length n as we consider paths of length $n + 1$.

□

[14] If $X(\mathbf{p}) - \rightarrow Y(\mathbf{p})$ then, earlier, the **Prepare** rule must have moved this connection to $X(\mathbf{p}) \rightarrow Y(\mathbf{p})$.

Definition 2 *Atomicity. We say that the atomicity property holds over a TM tree if for any TMs X and Y in the tree*

- *if* $\mathbf{c} \in OUT(X) \cup IN(X)$ *then* $\mathbf{a} \notin OUT(Y) \cup IN(Y)$ *and*
- *if* $\mathbf{a} \in OUT(X) \cup IN(X)$ *then* $\mathbf{c} \notin OUT(Y) \cup IN(Y)$

Theorem 1 *TIP ensures atomicity.*

Proof: Let X and Y be TMs. We prove that the Atomicity property of definition 2 holds. If $\exists \mathbf{c} \in OUT(X) \cup IN(X)$ then by Lemma 1 R was the first to commit, and when it did, $\forall x \in OUT(R)$, $x = \mathbf{p}$ held. By Lemma 2 all descendant connections of R must also be in state $\mathbf{p}$. For this situation to happen no connection could have been in state $\mathbf{a}$, by the condition of the **Prepare** rule. Once all connections are in a $\mathbf{p}$ state no connection can abort by the conditions of rules **Abort I**, **Abort II**, **EAbort I** and **EAbort II** rules.

Suppose $\exists \mathbf{a} \in OUT(X) \cup IN(X)$ then the **Prepare** rule cannot happen at X for one connection. This means that R will never commit and so neither will any other TM.

□

7 TIP Optimizations

7.1 One Phase Commit Violates Atomicity

A popular 2PC optimization is known as *last agent* [12]. The TIP specification[10] states that a one–phase commit can be performed when

> *"...the sender [*superior*] has 1) no local recoverable resources involved in the transaction, and 2) only one subordinate (the sender will not be involved in any transaction recovery process)".*

TIP supports a one–phase commit by allowing a COMMIT message to be sent from a superior to a subordinate when the superior and subordinate maintain an enlisted connection. Unfortunately, the specification's conditions as to when this may take place are not strict enough to guarantee atomicity, see definition 2. Consider the example of Figure 8.

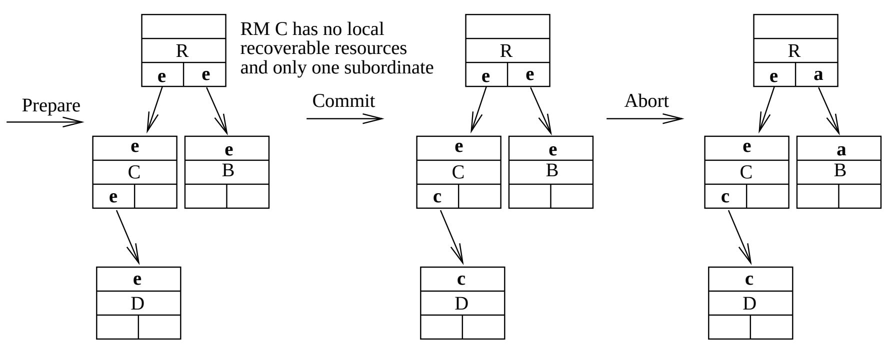

Figure 8: Atomicity Violation

We can rectify this problem by insisting that only the root may delegate commit and only when it has a single subordinate. In the case that this single subordinate has only one subordinate it can again delegate and so on.

In fact the condition of no local recoverable resources could be relaxed. This condition is certainly required if no reply is sent to commit delegation message. In TIP the delegated COMMIT message is replied to, with either COMMITTED or ABORTED. This means the superior, on receipt of this message, can take the appropriate action with local recoverable resources. This does however present a problem. If a failure were to occur and the outcome message returned to the superior were lost the superior can not easily determine the outcome. This is because after sending the outcome the subordinate forgets about the transaction. Changes to the TIP protcol could solve this problem but they are not currently implemented. In the case that a TM has no local recoverable resources it need not force write a prepare record to its log before delegating commitment.

It is unclear how TIP supports the delegation of commitment further than the root of the transaction tree. A TM can determine if it is the root of the transaction by checking that it has no superiors. If it is not the root it can determine if it has no siblings in the following way. If a TM receives a COMMIT message while in the the **e** state without going to through the **p** state it can determine that this COMMIT message must be of the delegate type. Since 80% of distributed transactions are read only[12], with one or two remote subordinates these optimization can represent a significant saving.

7.2 Acknowledgement Schemes

During a transaction, applications enlist one another and issue requests for work to be done on the transaction. This request response dialogue is application specific. It might be something like update a bank balance. A response to such a request would be an acknowledgement that the balance has been tentatively updated. The change is not made permanent until the transaction commits. Alternatively a response to a request might just acknowledge that the request arrived and that the update will be dealt with in due course.

Suppose the initiator application, A enlists another application, B and issues requests for work to be carried out, at B, on that transaction. Application B, once enlisted, tries to carry out the request and in so doing enlists another application C. An interesting question now arises. Should B send an acknowledgement to A to indicate that it has undertaken the work *before* it has received acknowledgements from C or should it always wait for C to acknowledge its own work in turn before passing the acknowledgement up to A. We discuss two options which applications might take.

Strict Acknowledgement Each application acknowledges a request when it has finished the work requested of it *and* when all enlisted subordinate applications have acknowledged their work in turn. In this case the initiator application will eventually receive acknowledgments from all its subordinates. Microsoft's implementation of TIP assumes only this policy.

Lazy Acknowledgement An applications is free to acknowledge a request from its superior before completing the work requested of it. It may also acknowledge the request before the enlistment of subordinates required to complete the request. In this case, when the initiator application receives an acknowledgement it cannot be sure that its subordinates have completed their work or even that all the participants in the transaction have been enlisted.

If strict acknowledgment is used, a transaction cannot exploit the parallelism achieved by working while messages propagate. For instance, if an application acknowledges work requests before carrying out the work requested, that work might well have completed by the time the acknowledgement message is received by the requester. In this way an application can overlap work with message propagation. It might seem that this strategy is flawed because the response might acknowledge work which was not in fact possible to perform. If this is the case then the application still has the option to abort the transaction when asked to prepare.

The increased concurrency of lazy acknowledgement comes at a price. The initiator application will receive lazy acknowledgements and may try to commit the transaction. This will cause each subordinate TM to be sent a PREPARE message. If all the work has taken place by the time the PREPARE message arrives the TM may respond PREPARED. If however work has still not been completed or further subordinates need to be enlisted the subordinate TM is faced with three options.

1. The TM can reply ABORT. We call this situation *rushed abort*
2. The TM can delay the PREPARED response until work has completed, and all its subordinates are enlisted. This might cause other prepared TMs to wait exposing them to blocking in the case of failure.
3. In the case where the TM has not finished enlisting subordinates it might just leave them out of the transaction. This may be possible if for instance it were just collecting prices for a product from available various vendors, as it could use the prices it has collected so far. We call this situation *excluded participants.*

In some cases a superior will need the final results of a work request before it can proceed. We call this constraint *ack with results.* If this is required then a lazy acknowledgement scheme would be inappropriate.

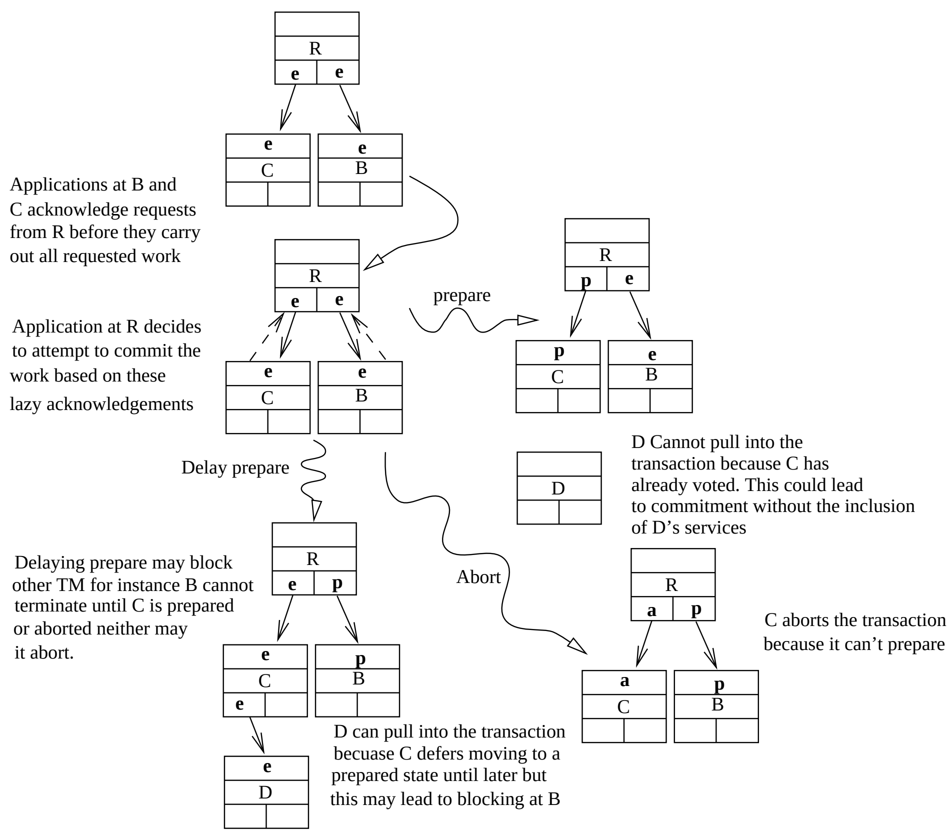

Figure 9: Problems with Lazy acknowledgement

7.3 Read Only Optimizations and Serializability

A very popular commercial two-phase commit optimization is known as read only[12]. This optimization when combined with presumed abort reduces the need for forced log writes. TIP supports this optimization by allowing a TM that does not need to know the outcome of a transaction to reply READONLY to a PREPARE message.

For instance if a TM and all its subordinates only performed reads of their resources then they can reply READONLY and release their read locks, when asked to prepare. In general, this optimization increases concurrency because locks on read only resources can be released earlier. When combined with lazy acknowledgement, concurrency can be increased further still because TMs may be asked to prepare earlier than in strict acknowledgement and so can release their locks earlier still.

Unfortunately combining lazy acknowledgement with read only, lowers transaction isolation levels. Consider an example adapted from[12]. TMs X and Y are subordinates to a common TM R. Both X and Y receive PREPARE messages. Y replies READONLY and releases locks before X has finished with the transaction. X[15] needs to access a resource that Y unlocked, but another unrelated transaction has locked the resource and changed it. When X gains access to the resource it has changed since Y unlocked it. The transaction will thus see an inconsistent view of the resource. One view read by X and a different view read by Y. This level of isolation is often known as READ_COMMITTED[6]. Obviously if strict acknowledgement is used then this scenario cannot arise and SERIALIZABLE isolation is maintained.

7.4 Early Preparation

Suppose application A requests services from applications B and C; see Figure 10. The transaction is pushed to the relevant TMs and work starts at both B and C. Application B pushes the transaction further to applications D and E. Suppose B completes its work receiving the information it needs from D and E and then informs A of completion. It then waits for notification from its TM to terminate the transaction. Application B's TM cannot independently commit the transaction because it has a superior and a sibling who might need to abort. It could however start the first phase of the two phase commit. We call this scheme *early preparation*. B contacts its TM and asks it to carry out *only* the first phase of the two-phase commit with the TMs of D and E. Once the subtree comprising nodes B, D and E have reached a prepared state they must wait. When B's TM receives a PREPARE message from A's TM it can immediately respond PREPARED.

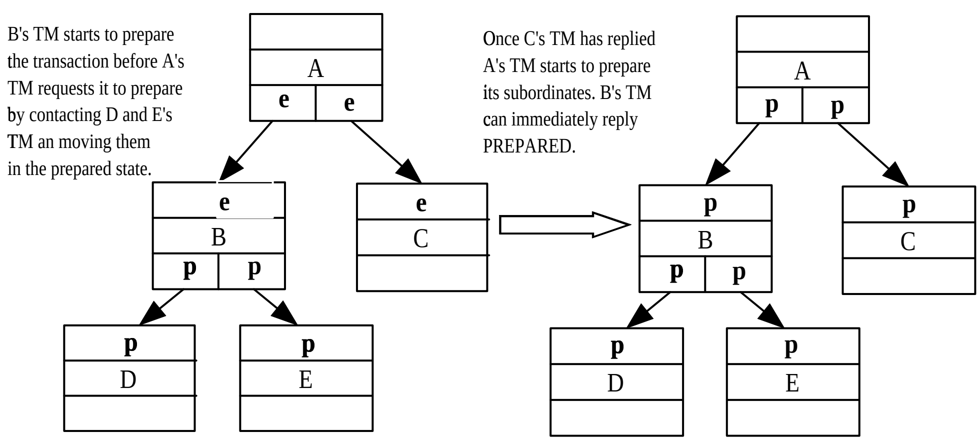

Figure 10: Early Preparation

This scheme could provide greater concurrency. While C is carrying out its work the applications that comprise the subtree B, D and E can perform half their commit processing[16]. If the request to C is long running overlapping processing in this way might present a useful gain. There is a cost associated with this optimization. Firstly, B must know that it will not be asked to carry out further work in the transaction and therefore it is safe to start to prepare the transaction. Secondly, there is the issue of exposure to blocking. Once a TM has replied PREPARED to its superior it requires notification of the outcome. If it loses the connection with its superior it must

[15] We slightly abuse notation for brevity's sake. We should write an application local to TM X but instead just write X.

[16] Part of this commit processing involves forced disk writes which in some systems constitutes most of the delay

hold locks on resources until it can reconnect. By replying PREPARED it has given up its right to unilaterally abort.

Table 1 summarizes the impact of combining Lazy and Strict Acknowledgement with Early and Late Prepare. The application programmer can choose to use Lazy or Strict Acknowledgement within the current TIP specification. In contrast Early Preparation would require an method to allow the application to request its local TM to carry out the first phase of the two phase-commit protocol. Currently Microsoft's implementation of TIP only supports the most conservative schemes of strict acknowledgement with late preparation.

	LA + PrepASAP	LA+PrepLate	SA+PrepASAP	SA+PrepLate
Isolation Level	READ_COMMITED	SERIALIZABLE	SERIALIZABLE	SERIALIZABLE
Concurrency	Best	Good	Good	Worst
Excluded Participants	Possible	Possible	Not Possible	Not Possible
Ack with results	No	No	Yes	Yes
Blocking Exposure	High	Low	High	Low
Rushed abort	Yes	Yes	No	No

Table 1: The impact of combinations of Lazy Acknowledgement (LA), Strict Acknowledgement (SA), Late prepare (PrepLate) and ASAP prepare on the TIP service.

8 Security and Trust

The two-phase commit protocol relies on the trust of the parties involved.

In two-phase commit, if an application states that it can commit a transaction and then later reneges on this promise atomicity is comprimised. We assume that if an application can be trusted then it will not behave in this way. This assumption is not an invalid one as long as we can be sure of the identities of the applications involved. If they were to act in such a disruptive manner then, because their identities are known, appropriate action can be taken, for example they could excluded from such business transactions in the future. Some protocols deal in trading environment where the level of trust is much lower[7].

We now discuss how to restrict transaction enlistment in TIP to only known trusted parties.

Although the TIP specification[10][9] states that TIP should use TLS[3] to provide encryption and authentication, it gives very few details of how to use the services of TLS to this end. We take this opportunity to outline how public key encryption and authentication services such as those offered by TLS, could be used to make TIP more secure. We extend the TIP protocol to provide secure methods of enlistment. Once TMs are enlisted they should communicate over encrypted channels. If our extensions are adopted we claim the following two security properties will hold.

1. If Application A with local TM X wishes to enlist application B with local TM Y in a transaction then no other application can be mistakenly enlisted. Furthermore A's identity is authenticated to B and vice versa.

2. No outside parties can detect that the messages being sent pertain to a TIP transaction.

We first examine the enlistment of TM Y by X. Let $priv(X)$ and $pub(X)$ denote respectively the private and public keys of a TM or application X. For an excellent account of public key encryption and authentication techniques see[13].

8.1 Secure Pull

The Pull enlistment method was described in Section 2.2. We now describe how this can be made secure. We assume secure communication between an application and its local TM. We refer to Figure 11 in the following.

1. Application A requests a TID from TM X to send to Application B. In this request application A informs TM X that it intends TM Y to initiate the Pull. TM X therefore internally associates the public key, $pub(Y)$, with the TID issued.

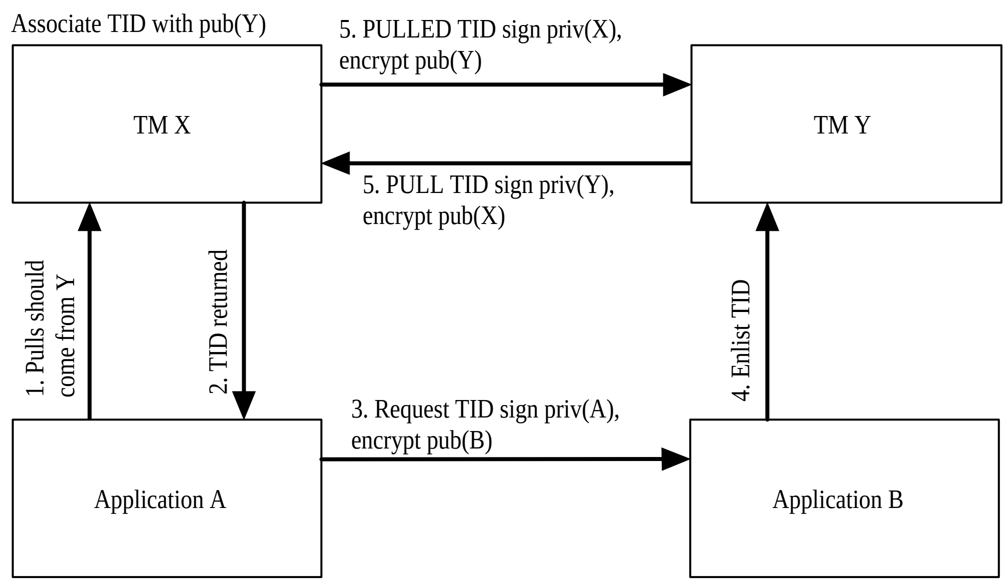

Figure 11: Secure PULL

2. TM Y dispenses the TID.
3. Application A sends the TID with a request for work to application B. The message is encrypted with the public key, $pub(B)$ and signed with $priv(A)$.
4. When application B receives the message it can be authenticated as having originated at A. It is encrypted with $pub(B)$ so no other application could intercept the message. Application B contacts its local TM Y and requests Y to pull into the transaction.
5. In the Pull request Y includes Y's signature generated using $priv(Y)$ and this request is encrypted with $pub(X)$.
6. When X receives the pull request from Y it validates this request by looking up $pub(Y)$ using the TID received and comparing it to Y's signature to ensure that the request was from TM Y.
7. TM X replies PULLED if everything checks out. This message is signed by TM X and encrypted with $pub(Y)$.

8.2 Secure Push

The Push enlistment method was described in Section 2.1. We now describe how this can be made secure. We refer to Figure 12 in the following.

1. Application A makes a request to its local TM X to push the transaction to Y.
2. A Push request is made from X to Y encrypted with $pub(Y)$ and signed with $priv(X)$.
3. On receipt of this request, Y can decide if it wishes to be enlisted in a transaction with X as its superior. If it does enlist then it creates a new TID, associates $pub(X)$ with the TID, and returns it to TM X. This messages is signed using $priv(Y)$ and encrypted using $pub(X)$
4. TM X returns the TID to application A, verifying that it came from Y.
5. Each of A's requests for work to be carried out at B carries the TID received in the previous step. This message can be signed using $priv(A)$ and encrypted using $pub(B)$.

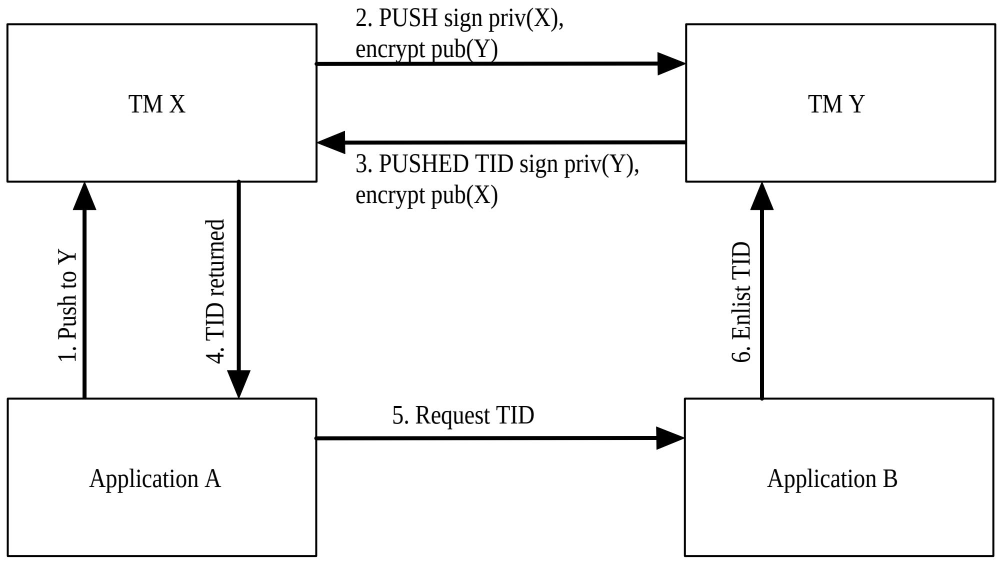

Figure 12: Secure Push

6. When application B registers work with TM Y, the TID is presented. TM Y can check to see if it issued this TID and knows that it was issued to X a valid superior.

8.3 Implications

In our scheme we require the application to application channels to be encrypted and secure. Although the TIP specification states that the TIP protocol channel should be secure it does not specify that the applcation channel should also be. A simple attack can be constructed if the application to application channel is comprimised, for both push and pull methods of enlistment. Currently Microsoft's implementation does not fully address this problem.

The scheme we suggest provides authentication and security. Any application or TM can authenticate a request and decide whether or not to join the transaction. A problem arises when the transaction tree has a depth of more than one.

If application A trusts application B and vice versa A can enlist B. B can now enlist another application C who it trusts but who is not trusted by A. A has no direct control over who joins the transaction. It may only authenticate subordinates. In this scheme A must be sure that it not only trusts its subordinates but also that it trusts them not to enlist any TM that it does not trust, and so on. We call this criteria *trust transitivity.*

9 Problems with TIP

In traditional transaction processing environments the participants in a transaction tend to cooperate. In an Internet environment there is an assumption that the participants might be competitors and may therefore not cooperate all of the time. It is therefore important to design an Internet protocol so that if a participant deliberately acts in a way to unfairly disadvantage other participants this can be detected. In the last section we discussed how enlistment of only trusted participants could be achieved. In this section we show how, once enlisted, a participant can act to deliberately disrupt a transaction without detection. The fact that it is trusted is not enough to ensure a fair playing field if it could act in such a way that blame cannot be apportioned.

9.1 Deliberate blocking

TIP uses 2PC to ensure atomicity. It is well known that 2PC suffers from *blocking*. Blocking occurs when a TM is prevented from terminating a transaction due to inopportune failures in other parts of the system[2]. In TIP a TM X becomes blocked if the connection to its superior fails in the **p** state. This will block every descendent of X.

In Figure 13 the connections have been lost between R and the most superior TMs in the subtrees labeled A and D. Until these connections are repaired no TM can terminate in either subtree D or A. They are blocked. Is it very important that the connection to a superior is not lost when it is in the **p** state because termination will require reconnection.

In TIP a TM could fake failure in order to block other TMs. It would be difficult for other blocked participants to tell the difference between genuine and fake failures. The other participants might have to hold locks on resources while blocked, and thus be at a disadvantage.

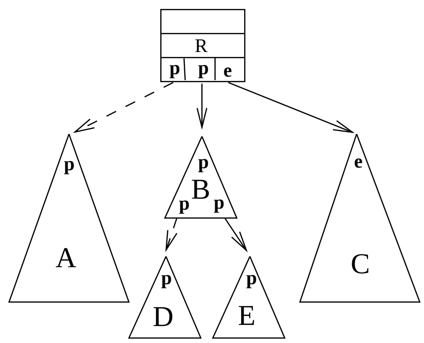

Figure 13: Blocking in TIP

One method which might reduce blocking (deliberate or otherwise) is to pass the TIP URL of R down the tree when it is grown. If connections are lost some blocking (deliberate or otherwise) can be avoided by reconnecting to R to determine the outcome[17]. This optimization is not currently supported by TIP.

9.2 Jamming

Participants in an Internet transaction might deliberately delay responses, faking failure, and then abort a transaction in order to gain a competitive advantage. We call this *jamming*. Two examples of jamming are given below.

Example 1 *Suppose a PC manufacturer places an order for DRAM and CPUs from two separate vendors. After the quantities and prices have been agreed upon the PC manufacturer starts to commit the transaction. Once the CPU vendor replies PREPARED to the PREPARE message it must not sell the allocated CPUs to anyone else. The DRAM producer is slow at responding to the PREPARE message and when it does so it replies ABORTED. The DRAM vendor (who also makes CPUs) used the transaction to ensure that the promised order of CPUs could not be sold elsewhere thereby restricting supply in the market place enabling it to sell its own CPUs at a higher price.*

Example 2 *Suppose a pension fund wishes to buy some government bonds and some gold futures in order to adjust the risk profile of the fund. The fund contacts two different investment banks one willing to sell it gold futures the other government bonds. The fund decides to use TIP to carry out the transaction*[18]*. The pension fund then tries to commit the transaction. A PREPARE*

[17] If a decision has been made it will be known at R.

[18] Atomicity is important as the gold futures are of little use to the fund without the government bonds and vice versa.

message is sent to each supplier. Ideally both suppliers will respond PREPARED together and the transaction will then commit. Suppose however the seller of the bonds responds PREPARED as soon as it is asked to PREPARE but the seller of the gold futures delays its response, because it is waiting for some imminent news on interest rates. If the news is favourable (the market price falls lower than the agreed delivery price) it replies PREPARED and the transaction commits. If the news is not favourable the seller of gold futures will reply ABORT and the transaction will globally abort.

Bond prices and gold futures generally move in the opposite directions when interest rates change. This means by delaying a response to a PREPARE message the bond seller always profits at the expense of the seller of the gold futures.

9.3 Superior's Advantage

A third problem in Internet commerce based on TIP is that of the superior's built in advantage.

Example 3 *Suppose Company R wishes to buy a product from a company A but does not know how much it should bid. In order to determine the price it might start by offering a high price for the product from the vendor A . Company R then starts to PREPARE the transaction. If the subordinate replies PREPARED, company R may think it has paid too much for the product. R can now abort the transaction and starts afresh with a lower offer. This asymmetry puts A at a disadvantage since once it has agreed to the price, R still has the option to abort.*

9.4 Excessive Aborts

Consider the example in Figure 14. A connection is lost with one subordinate of R before it enters the **p** state. All other connections from R have entered **p**. Once this failure has been detected by R it moves the connection to the **a** state. The global outcome must now be abort. If instead the TM tried to reconnect to its subordinate the connection may be able to be re-established in the **e** state and global abortion might be avoided. To do this we could introduce a new rule **ERec**.

$$\textbf{ERec}\ \frac{X(\mathbf{e}) - \rightarrow Y(\mathbf{e})}{X(\mathbf{e}) \longrightarrow Y(\mathbf{e})}\ \mathbf{a} \notin OUT(X) \cup IN(X)$$

A modification that might prevent excessive aborts would oblige a TM to try to reconnect before aborting (perhaps providing proof that the connection could not be established) otherwise it could always claim that a network fault caused an abort decision. In this way by enlisting and then faking failure and aborting a TM could deliberately abort transactions.

10 Trusted Root and Flattened Tree

Although TIP is just a protocol, its facilities engender a particular way of porocessing transactions that may not be optimal for all potential applications. In the previous section we saw a variety of problems that designers might experience if they were to use TIP in a very general way to carry out Internet transaction processing. We now suggest a very simple strategy to reduce these problems. This strategy can be achieved using TIP, but doing so in a restricted way. Part of our purpose in proposing this strategy is to force potential implementors to justify their design decisions if they opt for the more general use of TIP.

10.1 The Transaction Server

We propose the implementation of a transaction server (TS) which acts as the root for *all* TMs in a transaction. This server would be implemented by a third party who was impartial to, and trusted by, the businesses using the server. The server would be located at a secure site with an emphasis on very high reliability and scalability. Requirements of the server would be

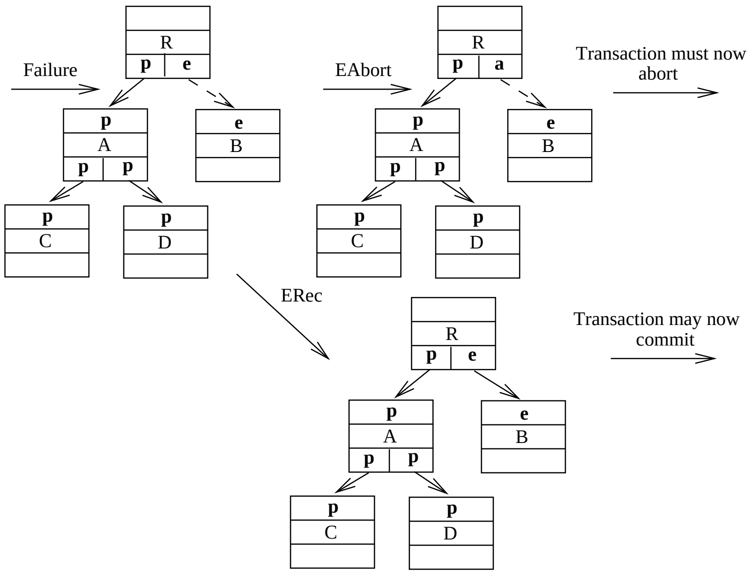

Figure 14: Excessive aborts in TIP

- High availability achieved using a combination of fault tolerance and robust hardware and software components.
- High Internet connectivity. The server should have multiple independent connections to the Internet at different geographical locations. The purchase of lease lines would allow the server to continue providing service between transaction managers which would otherwise become disconnected due to failures in the Internet.
- The server should support all current versions of transaction protocols. This allows transaction managers to communicate with one another through this central server even if they were running incompatible versions of transaction protocols.
- Fast non-volatile storage. This could be implemented using non-volatile RAM and fast disk arrays.

Internet transaction processing would now proceed as follows. The initiator application, A, requests a TID from the TS[19]. When application A wishes to enlist another application B, it provides B with the TID returned by the TS. B now pulls into the transaction by requesting its TM to contact the TS. Notice that all TMs pull into the transaction by contacting the TS. The high extra connectivity of the TS to different geographical locations around the Internet protects against connections failing and therefore reduces blocking and transaction aborts. The transaction tree structure will be a single routed flattened tree, of depth one. See Figure 15. When it is ready to terminate the transaction the initiator application contacts the TS. A two-phase commit is then carried out between the TS and all its subordinates.

In this scheme there are no problems with transitive trust. The initiator application can specify exactly who it is willing to trust when it requests the TID from the TS and the TS will restrict enlistment to these TMs. Rings of trusted sites can be built up and the initiator can specify one of these rings.

Acts of Deliberate Blocking and Jamming can be better controlled by the TS implementing a time out after which, if no response is received from a site, it will abort the transaction. Realtime commit protocols have been proposed[5] which could be supported by the TS. There is no problem with Superior's advantage since the TS is impartial.

[19] Security features can be addressed using the usual methods of authentication and encryption.

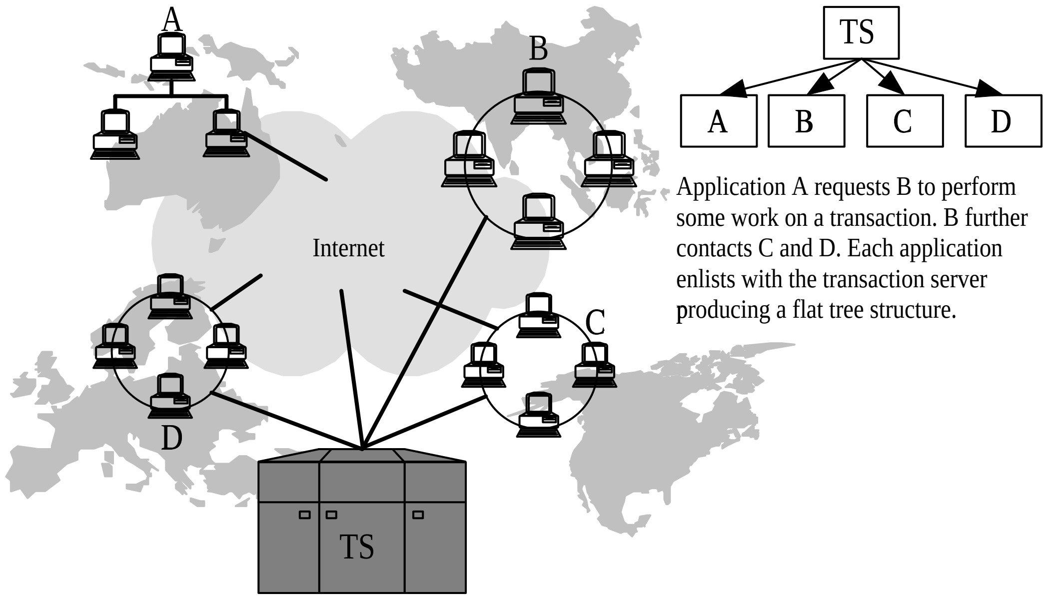

Figure 15: An Internet Transaction Server

Blocking will be reduced in two ways. Firstly because the TS is connected to the internet at many points failure is less likley to isolate a subordinate in its prepared state. Secondly, because the transaction tree is very flat the time a subordinate is in its prepared state is reduced.

	Deep Tree	Flat Tree
Transitivity of trust required	No	Yes
Deliberate blocking	Yes	Reduced
Blocking	Yes	Reduced
Jamming	Yes	No
Superior's advantage	Yes	No
Logging latency	High	Low
Excessive aborts	Yes	Reduced
Commit message latency	High	Low
Serializable with read only and lazy ack	No	Yes
Version incompatibility	Yes	No
Exchange of keys	Easy	Hard

Table 2: Comparison of deep and flat transaction trees in TIP

Logging latency is reduced in two ways. Firstly, the TS can have hardware support for faster disk writes and secondly because the transaction tree has been flattened logging at intermediate levels is not required. Furthermore, the outcome of a transaction will be available at a TM more quickly as it will not have to travel down a deep tree. We call this reduction of *commit message latency*.

If the TS disallows further enlistment from any TM once it has sent a PREPARE message to any of its subordinates serializable isolation, even when lazy acknowledgement is combined with the read only optimization, is guaranteed.

If the TS is equipped with multiple versions and types of transaction protocols it will mean that TMs that could not previously communicate with one another due to version incompatibility can now do so through the TS. Another service the TS could provide is a trusted exchange of public keys for the participating TMs. Table 2 summarises the advantages of using a TS.

There is cost associated with this scheme. Firstly the implementation support and maintenance of a TS would be costly. In an environment where many TM were doing business this cost could be amortized over the many users of the TS. Secondly, the TS must be scalable and highly reliable. This is because the centralised approach introduces a single point of failure. Lastly, if most of the TMs involved in the transaction are geographically distant from the TS then more messages will have to be sent over this larger distance compared with the deep tree scheme. However these messages can be sent in parallel so the response time will be similar in both schemes.

11 Conclusions and Future Work

We have seen that there are choices to be made both by the implementers of TIP and the application programmers using the TIP services. The particular choices made will depend on the level and type of transactional services required.

Our model of TIP provides a good framework in which to reason about possible TIP executions. A possible future direction might extend TIP to use a quorum based three-phase commit protocol to further alleviate the problems of blocking. Our model is amenable to model checking. Using a modal logic such as CTL liveness and safety properties can be formally verified using game based model checking algorithms.

An implementation of the transaction server would provide some insight into the usefulness of such a strategy and also perhaps provide some measure of the extra robustness gained using this scheme over a more conventional approach. In either case if past growth is an indication markets will be very quick to leverage the potential of Internet transaction processing services.

12 Acknowledgements

I would like to thank Jon Cargille and his team at Microsoft Corporation for their cooperation, suggestions and support when writing this paper.

References

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[3] T. Dierks and C. Allen. The TLS protocol. Technical Report Internet Draft IETF 2246, IETF, January 1999. On-line, http://www.ietf.cnri.reston.va.us/rfc/rfc2246.txt.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[5] R. Gupta, J. Haritsa, K. Ramamritham, and S. Seshadri. Commit processing in distributed real-time database systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, December 1996.

[6] T. Kempster, C. Stirling, and P. Thanisch. Diluting ACID. Technical Report ICSA-47-99, Dept. of Computer Science, University of Edinburgh, Febuary 1999. On-line, http://www.dcs.ed.ac.uk/home/tdk/.

[7] Steven Ketchpel and Hector Garcia-Molina. A sound and complete algorithm for distributed commerce transactions. Technical report, Distributed Computing, 1999.

[8] B. Lampson and D. Lomet. A new presumed commit optimisation for two phase commit. Research Note CRL 93/1, Digital Equipment Corporation, Cambridge Research Laboratory, 1 Kendall Square, Cambridge, MA 02139, February 1993.

[9] J. Lyon, K. Evans, and J. Klein. Tip supplemental information. Technical Report Internet Draft IETF 2372, IETF, July 1998. On-line, http://www.ietf.cnri.reston.va.us/rfc/rfc2372.txt.

[10] J. Lyon, K. Evans, and J. Klein. Transaction internet protocol. Technical Report Internet Draft IETF 2371, IETF, July 1998. On-line, http://www.ietf.cnri.reston.va.us/rfc/rfc2371.txt.

[11] Moses Ma. Agents in e-commerce. *Communications of the ACM*, 42(3):79–91, March 1999.

[12] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimisations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, October 1995.

[13] Andrew S. Tanenbaum. *Computer Networks (Third Edition Section 7)*. Prentice–Hall, Upper Saddle River, NJ 07458, 1996.