

# Systems Architectures for Transactional Network Interface

Manish Marwah and Shivakant Mishra  
Department of Computer Science,  
University of Colorado,  
Campus Box 0430, Boulder, CO 80309-0430

Christof Fetzer  
Department of Computer Science,  
Dresden University of Technology  
Dresden, Germany D-01062

## Abstract

*Systems such as software transactional memory and some exception handling techniques use transactions. However, a typical limitation of such systems is that they do not allow system calls within transactions. This is particularly true for system calls that interact with file systems, devices, and the network. This paper describes systems architectures that can be used to extend a transactional system to allow network read/write system calls within a transaction. This is done by delaying the sending of network bytes to a peer until a transaction is committed, and implementing a rollback mechanism in case a transaction aborts. Three different architectures, one transport layer and two application layer, are proposed to incorporate this extension. The paper discusses the advantages and limitations of each of these architectures. Prototypes of each of the three architectures have been implemented. The paper describes the design and implementation of these prototypes, and provides an extensive performance evaluation under many different scenarios, including LAN environment, WAN environment (PlanetLab), communication-intensive transactions, and computation-intensive transactions.*

## 1 Introduction

Transactional constructs are being implemented in modern programming languages to provide infrastructure support for systems such as software transactional memory (STM) and some exception handling techniques. STM [8, 5] provides a simple and effective alternative to various lock-based synchronization mechanisms used in concurrent programming. It converts each critical section of the code, marked by the programmer, into a transaction. The transactions proceed without any locks. At the end of a transaction, if more than one concurrently running transactions have made conflicting memory accesses, only one transaction is allowed to proceed. The remaining are aborted and restarted.

One important restriction in such transactional systems is that they do not allow system calls within transactions. The key reason for this restriction is that it is very difficult to roll back certain system calls. This is particularly true for systems call that interact with file systems, devices, and

the network. The fact that state information may escape to external entities such as the disk, devices, or the network makes rollback particularly challenging in these situations. However, an inability to include system calls within transactions severely limits their applicability. With communication networks being increasingly pervasive, more and more everyday computing applications involve some kind of network communication. As a result, current transactional systems that disallow network reads/writes within a transaction do not provide adequate support for most current computing applications. In [2], the authors allow system calls in transactions by ensuring that such transactions never get aborted. This is done by severely limiting concurrency during execution of such transactions. The adverse performance impact of this approach makes it viable only if transactions with system calls are extremely rare.

In this paper, we address this limitation of current transactional systems by incorporating network read/write system calls within a transaction, and providing support for rollback in case the transaction aborts. The key functionality needed to provide this support is to temporarily suppress the sending of the network bytes originating from a transaction until that transaction commits. These bytes are discarded and an appropriate rollback mechanism is incorporated in case the transaction aborts. Providing such a support is difficult because of two reasons: (1) to be effective, no changes should be required at the peer (client), and (2) the performance of the server application should not suffer, particularly during normal operation when most of the transactions commit the first time.

We investigate three different system architectures. All three architectures provide support for including network reads/writes within a transaction, and rollback when a transaction aborts. One of these architectures, the transport layer architecture (TLA), operates at the transport layer, and the other two, the two-connection application layer (2CA) and the single machine applications layer architecture (SMA), operate at the application layer. TLA and 2CA require an intermediate logger to store network bytes for rollback. All three architectures satisfy the following important properties:

1. They are completely transparent to the networking peer (client), i.e. absolutely no changes are required at the peer.
2. No OS changes are needed at the server, and only min-

imal OS changes are needed at the logger.

3. Only minimal changes are needed at the server application.
4. There is minimal overhead under normal scenario, when the number of transaction aborts is very low. The transactional rate achieved in this scenario is comparable to that of non-transactional sends and receives.

We discuss the advantages and limitations of each of these architectures. To evaluate these architectures, we have implemented three prototypes. This paper describes the design and implementation of these prototypes, and provides an extensive performance evaluation under many different computing scenarios, including LAN environment, WAN environment (PlanetLab), communication-intensive transactions, and computation-intensive transactions.

The performance results show that the overhead of using a transaction network interface is relatively insignificant. In fact, for communication-intensive transactions, the overhead becomes negligible. Also, the performance approaches that of the non-transactional transfer rate as the number of aborts goes to zero. In terms of performance, no single architecture is clearly superior to the others. However, there are other design constraints that may make one architecture preferable to another for a particular situation.

## 2 System Architecture

In order to provide a transactional network interface for TCP connections, we investigate three different system architectures: TLA, 2CA, and SMA. The basic idea in all three architectures is that bytes sent out by the server application are not delivered to the peer until the corresponding transaction commits. Furthermore, the bytes destined to the server application are logged, so that those bytes can be resent without peer involvement when a failed transaction restarts. In TLA and 2CA, the transaction bytes (in both directions of the TCP connection) are saved at a separate logger. In SMA, they are saved in the main memory of the application server. Notice that while TLA and 2CA require a separate logger, this logger can be shared with other services, e.g., services that provide support for failure recovery, security related logging, etc.

### 2.1 Transport layer architecture (TLA)

The transport layer architecture requires a logger that provides transport layer support for the management of transactions. All TCP segments, in both directions, are routed through the logger machine, where they are intercepted and logged. A very important property of TLA is that the end-to-end TCP connection semantics are preserved between the peer and the server application. This is analogous to TCP splicing [4, 9, 6, 7] used in building high performance web servers.

Both TLA and 2CA (described in Section 2.2) require a separate control connection between the server and the logger. This TCP connection is used for communicating transaction events, such as transaction *start* (event TX\_START),

*commit* (TX\_COMMIT), *abort* (TX\_ABORT), or *restart* (TX\_RESTART).

#### 2.1.1 Transaction processing

There are three different phases in processing of a transaction: (1) when a transaction is ongoing; (2) when a transaction is committed; and (3) when a transaction is aborted.

**Ongoing transaction:** The server has sent a TX\_START to the logger. The TCP segments that are received from the peer are saved at the logger, and then immediately sent on to the server. The segments that are received from the server are saved at the logger, but *not* sent to the peer. These TCP segments are saved in the kernel, and so no kernel-to-user or user-to-kernel data copying is involved. Since the server to peer bytes are not immediately forwarded to the peer until a transaction commits, the number of bytes that a server can send as a part of a single transaction is at most the size of its TCP send buffer. Similarly, since the acks do not reach the peer during a transaction, the peer is also limited to sending at most a TCP send buffer worth of data as part of a single transaction. This is not a problem in practice, because TCP buffers are quite large, accommodating several megabytes of data per connection, particularly in newer systems equipped with gigabit network cards. Furthermore, the regular non-transactional network interface may be used for transferring bulk data, e.g very large files.

**Transaction is committed:** The server sends a TX\_COMMIT to the logger once it has successfully completed a transaction including all network operations, and is ready to commit. Since the control connection is separate from the end-to-end peer-server TCP connection, it is possible that the server TCP send buffers may still be holding (unsent) data belonging to the transaction that was just committed by the application. On receiving a TX\_COMMIT, the logger sends out all the saved TCP segments destined for the peer. Recall that these segments were stored in the kernel, and so no user-to-kernel data copying is involved in this operation. The logger also purges these TCP segments after forwarding them to the peer, as well as purges the saved peer-to-server TCP segments related to the just committed transaction.

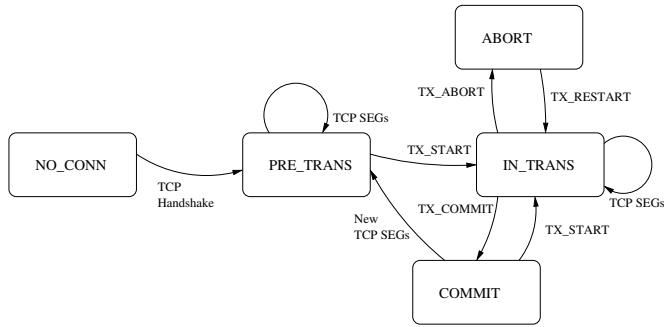
**Transaction is aborted:** When the server application needs to abort a transaction, it sends a TX\_ABORT to the logger, followed by a TX\_RESTART as it prepares to re-execute the aborted transaction. When a transaction aborts, the logger needs to perform a number of tasks related to both the segments received from the server and those from the peer:

- **Server-to-Peer Segments:** The logger discards all the bytes that the server had sent as a part of the aborted transaction. Thus, these are the bytes that the server has sent to the peer, and the peer never receives them, and hence never acknowledges them. So, these bytes must be acknowledged by the logger. The logger acks these bytes by generating one or more *fake* acks and then purges the corresponding saved bytes. Note that acking and discarding these bytes creates a “hole” in the server sequence number space from the viewpoint

of the peer. Thus, in order to keep this transparent from the peer, the sequence numbers on any subsequent segments sent to the peer on this connection are suitably adjusted. A corresponding adjustment is made to the acks going in the opposite direction. Again, this sequence number mapping in the TCP segments going in both directions is done in the kernel, analogous to TCP splicing [9].

- **Peer-to-Server Segments:** These are the bytes that the peer has sent to the server that were part of the aborted transaction. Recall that these bytes were saved in the logger before being forwarded to the server. These bytes have to be re-sent to the server once the (aborted) transaction is re-started. The logger sends these bytes with *new* sequence numbers. Note that this entire process of restarting a transaction is completely transparent to the peer. As with server-to-peer TCP segments, the sequence number of subsequent peer-to-server TCP segments are suitably adjusted due to the insertion of these (re-sent) peer-to-server bytes. Also, a corresponding adjustment is made in the acks going from the server to the peer. Once again, this sequence number mapping in the TCP segments going in both directions is done in the kernel, analogous to TCP splicing [9].

## 2.1.2 Logger State



**Figure 1.** States assigned to a transactional TCP connection at the logger.

Figure 1 shows the states a TCP connection is assigned at a logger in order to support transactional data on that connection. A server application can run any number of transactions on a TCP connection, as long as all these transactions are serialized, i.e., only one transaction can run on a connection at any instant.

After a TCP connection is established, the state of that connection is marked as PRE\_TRANS and data passing on that connection is logged. This is done even though a TX\_START has not yet been received from the server. The reason for this is as follows. When a server starts a transaction, it may read bytes, part of that transaction, that

may already be in the TCP receive buffers of the server. Thus, it is possible that the initial transaction bytes from the peer may pass the logger before the server can indicate a TX\_START. To address this scenario, the logger saves all segments – even those that are not a part of an ongoing transaction, since they may become part of a future transaction. Saved bytes that do not get associated with a later transaction are aged out and discarded. This aging out is done both on the basis of the amount of such data and the time duration for which it has existed on the logger. While in the PRE\_TRANS state, the logger continues to save TCP segments that arrive. It transits to the IN\_TRANS state when a TX\_START message is received from the server. This message also includes the starting TCP sequence number offsets, in both directions, for that transaction.

From the IN\_TRANS state, the server can signal to commit or abort a transaction. The connection for a committed transaction stays in the COMMIT state until all the segments related to the transaction are reliably received. The commit message from the server also includes TCP sequence number offsets marking the end of the transactions. If a segment with a sequence number beyond the end of the committed transaction is received, the logger transitions to the PRE\_TRANS state.

If a transaction aborts, the logger acks and discards server-to-peer bytes. As mentioned earlier, from the peer’s perspective, this creates a hole in the server’s sequence number space that needs to be plugged by suitably modifying the sequence and ack numbers of future segments. A similar adjustment is also required due to resending of bytes to the server. Figures 2(a) and 2(b) show the sequence number space of a server-to-peer TCP connection. Assume that the number of peer-to-server bytes resent is  $d_1$ , and the number of server-to-peer bytes discarded is  $d_2$ . Then for the peer-to-server ( $P \rightarrow S$ ) direction, the new sequence ( $S'_{p \rightarrow s}$ ) and ack ( $A'_{p \rightarrow s}$ ) numbers can be computed as:

$$S'_{p \rightarrow s} = S_{p \rightarrow s} + d_1; \quad A'_{p \rightarrow s} = A_{p \rightarrow s} + d_2$$

Similarly, for the server-to-peer ( $S \rightarrow P$ ) direction, the new sequence ( $S'_{s \rightarrow p}$ ) and ack ( $A'_{s \rightarrow p}$ ) numbers can be computed as:

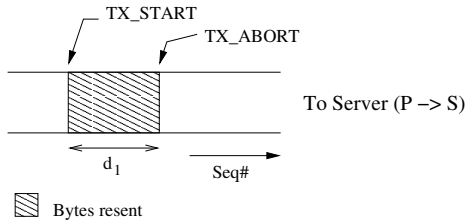
$$S'_{s \rightarrow p} = S_{s \rightarrow p} - d_2; \quad A'_{s \rightarrow p} = A_{s \rightarrow p} - d_1$$

Once again, note that this is similar in concept to how sequence and ack numbers are modified in a TCP splice [9].

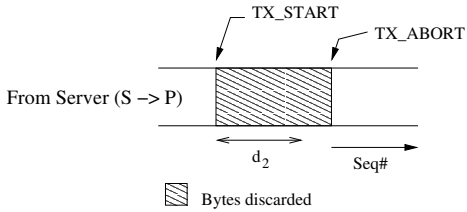
## 2.1.3 TLA: Discussion

The main advantage of TLA is that it maintains the end-to-end TCP connection semantics between the peer and the server. It employs techniques similar to TCP splicing by performing all buffering and sequence number mapping in the kernel. As a result, based on the experience from TCP splicing [6], the performance overhead due the introduction of the logger is expected to be insignificant.

There are some concerns in using TLA. One is its impact on the calculation of RTT. Since the logger holds server



(a)  $P \rightarrow S$  sequence space



(b)  $S \rightarrow P$  sequence space

**Figure 2.** Sequence and ack number adjustments are required due to transaction aborts.

segments until a transaction commits, the round trip time (RTT) computed by the server will be skewed, which could adversely impact the throughput between the server and its peer. However, the actual data transfer during a transaction is not very high (a few megabytes at the most). So, the impact of a somewhat skewed RTT is expected to be minimal. Also, as mentioned earlier, non-transactional network interface is used for large data transfers.

Another concern is related to the duration of a transaction. Since the data sent out during a transaction is not acknowledged until the transaction is committed, the TCP connection may fail if the transaction lasts too long. The server and peer TCP may generate re-transmissions, which although harmless from the point of view of correct functioning, can lead to some overhead. However, transactions are not expected to last very long in a practical setting, perhaps a few times the RTT at the most.

Finally, the amount of data sent/received during a transaction is bound by the size of the TCP send buffer on the server and the peer. As mentioned earlier, this is not a problem in practice, because TCP buffers are typically quite large, accommodating several megabytes of data per connection.

## 2.2 Two Connection Application layer architecture (2CA)

In 2CA the TCP connection between the peer and the server is broken up into two distinct TCP connections: (1)

a TCP connection between server and the logger (logger—server TCP connection); and (2) a TCP connection between the logger and the peer (logger—peer TCP connection). Logically, the logger in this architecture operates in a very similar fashion as that in TLA. In fact, the control connection between the server and the logger and the messages exchanged on this connection (see Section 2.1) are identical.

The logger state machine is also almost identical to that of TLA shown in Figure 1. The key difference is that when a transaction is ongoing, the logger application saves the server and peer bytes in main memory. When the logger application receives bytes from the server, it saves them in main memory. Note that these bytes are acked immediately by the logger—server TCP on the logger side. When the logger application receives bytes from the peer, it saves them in main memory, and sends them to the server via the logger—server TCP connection. When a transaction commits, the logger application sends out the server bytes to the peer via the logger—peer TCP connection. Similarly, when a transaction aborts, the logger application discards server bytes, and resends peer bytes to the server via the logger—server TCP connection. Conceptually, this architecture is very similar to that of a proxy application, reading data from one socket and writing it to the other.

### 2.2.1 2CA: Discussion

The main advantage of 2CA is that no OS changes are required at the logger. Furthermore, the amount of data sent during a transaction is not limited by the size of the TCP send buffers. There are two major drawbacks of 2CA as compared to TLA: (1) The end-to-end TCP semantics of the connection between the server and the peer are no longer preserved. The TCP bytes now logically go from the peer (server) to logger, and then from the logger to the server (peer). This is in contrast to TLA, where the TCP bytes are logically exchanged between the server and the peer. (2) 2CA is expected to be slightly more inefficient because it requires copying data between kernel space and user space twice: once when the logger application reads data from a socket, and then again when it writes that data.

### 2.3 Single machine application layer architecture

The single machine architecture (SMA) does not use a separate logger. Instead, the function of the logger is performed by local library functions. Bytes written to a socket are stored in memory. Only when a transaction commits are the saved bytes sent to the peer. Similarly, bytes received from a socket are stored in main memory in addition to passing them on to the server application. Only when a transaction commits are the saved bytes purged from memory. On a transaction abort, bytes written to a socket are discarded without sending them to the peer, and bytes received from a socket are again passed to the application after a restart. Transaction events such as start, commit and abort are communicated using an appropriate IPC mechanism. This architecture is similar in spirit to the one described in [3], which aims to support external I/O in atomic blocks.

The main advantage of this architecture is that it does not require any changes to the kernel, and since it does not split the server–peer TCP connection into two connections as is done in 2CA, it retains the the end-to-end semantics of a TCP connection.

The main drawback is that this architecture conceptually co-locates the functionality of the logger with the application server. This may be undesirable in many instances where server resources, such as memory and CPU are scarce. Furthermore, a failure of the logger code can potentially cause the application server to fail. Also, many users may already have a logger for other purposes that is shared by the application servers and may want to use that instead of installing additional software on every application server.

### 3 Experiments and performance evaluation

We conducted a series of experiments over networks with diverse characteristics to evaluate the performance of the three proposed transactional network interface architectures, namely, TLA, 2CA and SMA. We measured five important performance characteristics in our experiments: (1) Measure the overhead of a transactional network interface as compared to a non-transactional one; (2) Compare the relative performance of the three architectures; (3) Measure the performance of the three architectures over WAN links with varied round trip times; (4) Measure the impact of percentage of transaction aborts in computational-intensive as well as communication-intensive transactions; and (5) Measure the effect of transaction aborts on the performance of the system as the percent of transactions that abort is varied.

We performed the experiments in both LAN and WAN settings. The server and logger machines that we used are attached to the same LAN on the campus network of the University of Colorado at Boulder. To test under a LAN environment, the peer application was installed on a machine connected to the same LAN as the logger. In order to run our experiments over diverse WAN links, we used PlanetLab [1] nodes as our testbed. We placed the peer on two distinct sites: (1) a PlanetLab node at MIT (planetlab7.csail.mit.edu); and (2) a PlanetLab node in Singapore (planetlab3.singaren.net.sg). These locations were chosen, since they provide a wide variety of round trip times (RTT) between the server in Colorado and the peer. The RTT is about 69 ms for the machine at MIT and about 254 ms for the one in Singapore. Compared to these the RTT for the LAN scenario is about 0.25 ms.

#### 3.1 Experimental Setup

Like most organizations, the University of Colorado (CU) has a firewall which restricts outside machines from connecting to TCP ports on machines inside the firewall. Thus, a network peer on a PlanetLab node cannot initiate a TCP connection to a server located inside the campus network. The only network port that is open to TCP connections originating from outside the campus network is the ssh port. We use ssh tunneling, supported in OpenSSH (a

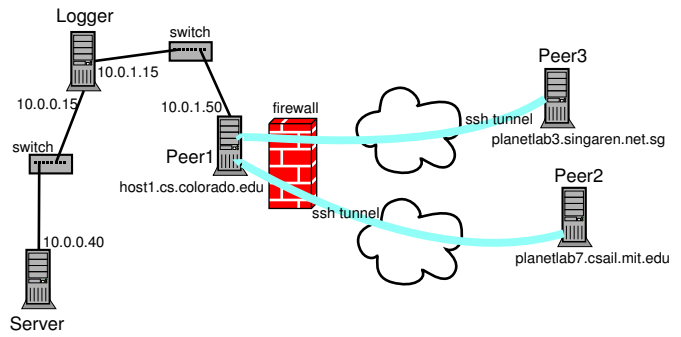


Figure 3. Experimental setup.

free version of ssh available on Linux platforms), to circumvent this problem. The peer-to-server TCP connection is tunneled through an ssh session between a machine inside the campus firewall and a PlanetLab node. The experimental setup is shown in Figure 3. The three locations where the server’s peer is located during the course of the experiments are marked in the figure as Peer1, Peer2 and Peer3.

#### 3.2 Experiments

A single run in our experiments consists of sequentially executing 100 transactions. A transaction is comprised of a request (a few tens of bytes long) sent to the server on a TCP connection, followed by a response of size 100 KB. The peer performs an active connect to establish a TCP connection and all transactions are sent on the same connection. From the server’s perspective, a transaction starts when a request arrives and commits when it has generated and sent the entire response. We simulate an STM system on the server by performing aborts on a certain percentage of the transactions. We abort a transaction at most once, i.e., all transactions that abort on the first attempt succeed on the second. Since the point where a transaction aborts influences the performance of the system, all transactions that abort do so exactly mid-way in the transaction, i.e., when the server application has sent out half the output response. Furthermore, the actual transactions that abort in a run are chosen at random.

We assume that the computational cost of a completed transaction is 30 ms and that of an abort is half of that, i.e., 15 ms. This cost is implemented by using the `nanosleep()` system call (which on most Linux systems has a resolution of about 10 ms) on the server. Note that this system call only guarantees that the sleep time will be *at least* the amount passed as an argument. In practice, we found that for the values we used, the actual sleep time is routinely a few ms over.

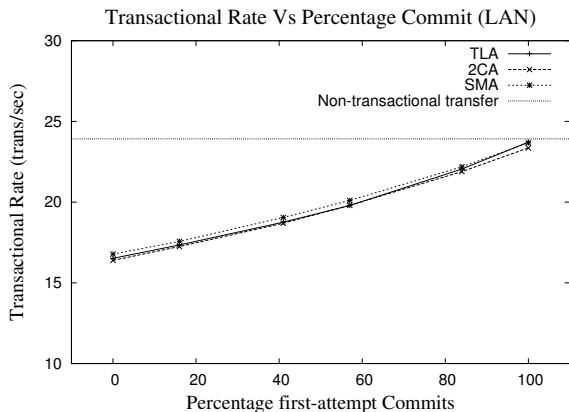
We run experiments for a range of first-attempt commit percentages, i.e., commits that occur without any aborts. In short, we also refer to this simply as commit percentage. We start with 0%, which corresponds to all transactions aborting in the first attempt before committing. The experiments are repeated at every interval of about 20%, with the final run at

100% commit, which corresponds to all transactions committing in the first try. Furthermore, we took at least three measurements for each run, excluding the first run which was discarded to minimize the effect of cache misses.

We run experiments for three different scenarios: (1) LAN, with the peer on the local LAN; (2) WAN1-MIT, with the peer located at MIT; and (3) WAN2-SG with the peer located in Singapore. For each of these scenarios, we conducted experiments for all three architectures, and for each architecture, six different values of commit percentage (ranging from 0% to 100%) are considered.

Regular, non-transactional network interface is also used in each scenario to transfer exactly the same request/response data as is done in the transactional architectures. The measurements obtained are used to determine the overhead of the three transactional network interface architectures.

### 3.3 Discussion of results: LAN Scenario

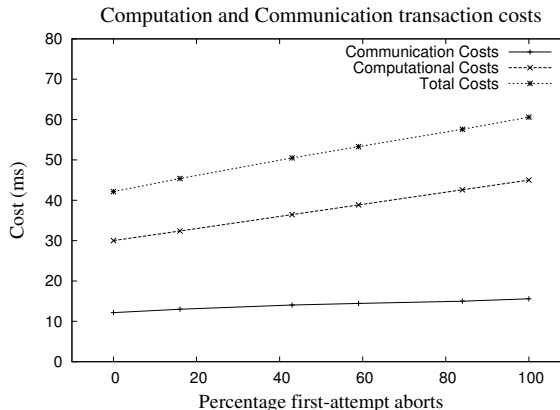


**Figure 4.** Transactional rate versus percentage of first-attempt commit transactions for the LAN scenario.

The first set of experiments consisted of running the transactions for the LAN scenario, with the peer on the same LAN as the logger. The logger is multihomed and as can be seen in Figure 3, all packets between the peer and the server must pass through the logger.

The transactional rates for the three architectures under different commit percentages are plotted in Figure 4. The graph shows that the transactional rate increases with increase in the percentage commits. As the number of aborts decline, the resulting saving in computational and communication costs are reflected in the increasing transaction rate. From Table 1, which summarizes the results, we see that TLA at 0% commit has an average transaction time of 60.6 ms. Out of this the computational cost is at least 45 ms (15 ms for the failed attempt plus 30 ms for the committed transaction) and the rest, 15.6 ms, is the communication cost. We calculate the computational and communication costs for all abort percentages for TLA and plot it in Figure 5. This graph shows that in addition to the computa-

tion costs (which are linear by design), the communications costs also grow only linearly with increase in the number of aborted transactions. Although, the data plotted is for TLA, this result is true for the other two architectures as well.



**Figure 5.** Communication and computational costs of a transaction for TLA in the LAN scenario.

When implemented without transaction semantics, the transactional rate is 23.92. This is the best rate that the transactional architectures can hope to achieve. From Figure 4, we see that as the percentage of first-attempt commits reaches 100%, the transaction rate of the three architectures approaches that of the non-transactional case. In fact, in the region close to 100%, the difference is quite low. This is especially important, because good transactional systems are engineered such that the abort percentage is low and that during normal operation the first-attempt commits are close to 100%. In such a scenario, the overhead of using one of the proposed transactional architecture is quite low.

The differences between the three architectures is not significant. Although SMA seems to perform marginally better, in percentage terms, the difference is negligible. Also, towards the 100% commit level, TLA and SMA seem to perform identical, while 2CA is marginally behind. It is possible that 2CA is slightly behind due to the fact that it involves kernel-to-user and user-to-kernel space copying at the logger, however, the difference is small enough that this cannot be conclusively stated and requires further investigation.

The average transaction rates, together with the average, minimum and maximum time taken per transaction are summarized in Table 1.

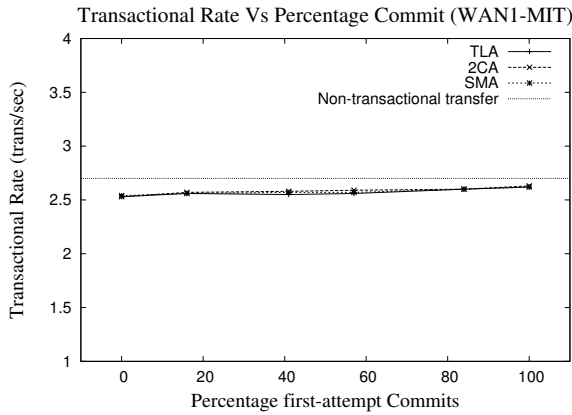
### 3.4 Discussion of Results: WAN Scenarios

In order to measure the performance of our architectures with the peer separated from the server over a WAN link, we used a PlanetLab node at MIT and another in Singapore. Since the RTT's of the links to these machines are large, we expect the communication part of the total transaction time to be dominant. Taking one set of measurements

% Commit	TLA				2CA				SMA			
	tx/sec	Time per tx (ms)			tx/sec	Time per tx (ms)			tx/sec	Time per tx (ms)		
		ave	min	max		ave	min	max		ave	min	max
0	16.5	60.6	59.7	61.5	16.4	61.0	60.0	69.9	16.8	59.6	58.4	60.6
16	17.4	57.6	41.9	62.4	17.3	58.0	41.6	62.3	17.6	56.9	42.0	60.8
41	18.8	53.3	41.9	61.5	18.7	53.5	41.8	63.1	19.1	52.5	41.9	60.5
57	19.8	50.5	41.9	78.1	19.9	50.5	41.8	62.3	20.1	49.8	41.9	60.6
84	22.1	45.4	41.7	61.3	21.9	45.7	41.8	61.9	22.2	45.1	41.8	60.5
100	23.7	42.2	41.7	44.3	23.4	42.8	41.8	44.7	23.7	42.3	41.8	43.1

**Table 1.** Average transaction rate for the three architectures in the LAN scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

for a WAN scenario for the three architectures with varying commit percentages takes at least a few hours (Note that we repeat each run three or more times and take the average.). For the WAN links, it took us several attempts of running the experiments in order to collect meaningful results due to significant temporal variation in the network characteristics of the links.

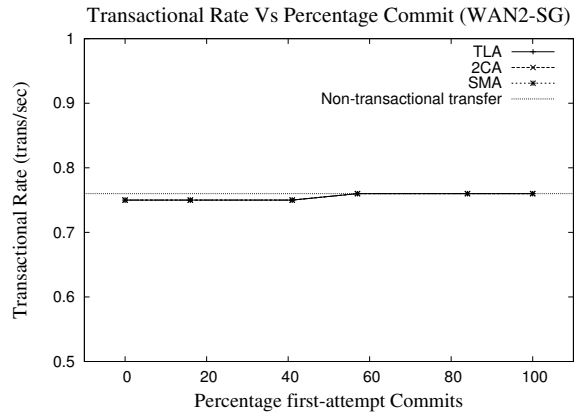


**Figure 6.** Transactional rate versus percentage of first-attempt commit transactions for the **WAN1-MIT scenario**.

Figure 6 shows the average transaction rates for the MIT node (WAN1-MIT scenario). The increase in the transaction rate with the commit percentage is *not* very visible in this case unlike in the LAN scenario. Also notice that this is despite the fact that the y-axis in the WAN1-MIT case has a much smaller range.

This is because the data transfer time taken between the server and the peer dominates all other factors. Here, the computational cost of a transaction is only about 10% of the total cost as compared to the LAN scenario where it is about 75%. Furthermore, the performance variations due to the different architectures become insignificant. The slight differences between the three architectures seen in the figure are likely due to changes in the network link characteristics during the course of running the experiments. The non-transactional transfer rate in this case corresponds to

2.7 transactions/sec which is within 7% of the lowest transactional rate of any of the architectures at any commit percentage.



**Figure 7.** Transactional rate versus percentage of first-attempt commit transactions for the **WAN2-SG scenario**.

The results for the second WAN link scenario (WAN2-SG), where the peer is located on a PlanetLab node in Singapore, are shown in Figure 7. Here the variation in the transactional rate due to the architectures or the commit rate is even less. In fact even with a y-axis with a very narrow range, hardly any variation is seen. The corresponding non-transactional rate in this case is 0.76 transactions per second, which is the same as that for the three architectures when the commit percentage is greater than 57%. The results are summarized in Table 3.

## 4 Conclusions

In this paper, we have proposed and compared the design and performance of three system architectures to support a transactional network interface. A transactional network interface allows an STM system to support network read/write operations within an atomic block. In addition to STM, there are several other potential uses of a transactional network interface. For example, the proposed archi-

% Commit	TLA				2CA				SMA			
	tx/sec	Time per tx (ms)			tx/sec	Time per tx (ms)			tx/sec	Time per tx (ms)		
		ave	min	max		ave	min	max		ave	min	max
0	2.53	395	389	622	2.53	395	358	633	2.54	394	389	639
16	2.56	391	373	611	2.57	389	341	577	2.56	390	373	593
41	2.55	392	348	648	2.58	387	373	620	2.57	389	352	603
57	2.56	390	374	600	2.59	390	342	623	2.57	388	348	640
84	2.60	385	375	621	2.60	385	343	609	2.60	384	359	622
100	2.62	382	374	584	2.63	382	374	634	2.62	381	374	602

**Table 2.** Average transaction rate for the three architectures in the WAN1-MIT scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

% Commit	TLA				2CA				SMA			
	tx/sec	Time per tx (s)			tx/sec	Time per tx (s)			tx/sec	Time per tx (s)		
		ave	min	max		ave	min	max		ave	min	max
0	0.75	1.34	1.14	2.33	0.75	1.33	1.32	2.34	0.75	1.33	1.32	2.35
16	0.75	1.34	1.30	2.32	0.75	1.33	1.30	2.19	0.75	1.34	1.30	2.34
41	0.75	1.33	1.30	2.33	0.75	1.33	1.30	2.32	0.75	1.33	1.30	2.33
57	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.19	0.76	1.32	1.30	2.33
84	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.32	0.76	1.32	1.30	2.34
100	0.76	1.32	1.30	2.20	0.76	1.31	1.30	2.32	0.76	1.31	1.30	2.32

**Table 3.** Average transaction rate for the three architectures in the WAN1-SG scenario. Also listed are the average, minimum, and maximum times taken for a transaction.

tectures can be used to extend exception handling systems that rollback state and re-execute code to support network read/write operations. The logger implemented in TLA and 2CA can be used for building a fault-tolerant server where server applications may be non-deterministic in nature.

The performance results show that the overhead of using a transaction network interface is relatively insignificant. In fact, for connections with long RTT values, the overhead becomes negligible. Also, the performance approaches that of the non-transactional transfer rate as the number of aborts go to zero. In a well engineered system, it is highly likely that the abort rate is low during normal operation.

In terms of performance, no single architecture is clearly superior to the others. However, there are other design constraints that may make one architecture preferable to another in a particular situation. For example, unlike 2CA, TLA provides an end-to-end TCP connection and is efficient since it avoids kernel-to-user and user-to-kernel space copying. However, 2CA has the advantage that it does not require any OS changes on the logger. Similarly, TLA or 2CA is preferred over SMA if the user does not want a logger on each application server and would like to reuse an existing logger machine for that purpose.

## References

[1] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawroniak. Operating systems support for planetary-scale network services. In *NSDI*, pages 253–266. USENIX, 2004.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, April 2006.

[3] T. Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[4] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of INFOCOMM'98*, March 1998.

[5] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[6] M. Marwah, S. Mishra, and C. Fetzer. Fault-tolerant and scalable tcp splice and web server architecture. In *SRDS*, pages 301–310. IEEE Computer Society, 2006.

[7] M.-C. Rosu and D. Rosu. Kernel support for faster web proxies. In *USENIX Annual Technical Conference, General Track*, pages 225–238, 2003.

[8] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[9] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.