17th ACM Symposium on Principles of Distributed Computing, June 1998

Persistent Messages in Local Transactions

David E. Lowell and Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{dlowell,pmchen}@eecs.umich.edu
http://www.eecs.umich.edu/Rio

Abstract: We present a new model for handling messages and state in a distributed application that we call Messages in Local Transactions (MLT). Under this model, messages and data are not lost after crashes, and all sends and receives are performed in local transactions. The model is unique in that it guarantees consistent recovery without the complexity or overhead of other recovery techniques. Applications using MLT do not need to coordinate checkpoints, track causal dependencies, or perform distributed commits. We show that MLT can be implemented using any reliable protocol. Finally, we describe our implementation of Vistagrams, a system based on the MLT model. We show that Vistagrams are just as fast as traditional messages, despite the recoverability they offer. The efficiency of our model and our Vistagrams implementation is enabled by the availability of fast stable storage, such as the reliable memory provided by the Rio file cache.

1. Introduction

Writing a distributed application that can reliably manage persistent data is difficult. Such applications have to contend with the possibility of system crashes occurring at inopportune moments, possibly leaving the distributed computation in a meaningless state. Furthermore, this problem worsens as distributed systems grow: the probability of some node in a distributed system experiencing a crash increases as more and more nodes are involved in the distributed computation.

Over the years, researchers have developed techniques such as message logging, distributed transactions, and coordinated checkpointing to help ameliorate these difficulties. These methods work by making the state of a distributed computation *recoverable*—that is, they restore or recompute a consistent view of the system's state after a crash. Unfortunately, current recovery techniques tend to send extra messages, use complex recovery, or work only for piecewise-deterministic applications.

In this paper we introduce a new model for handling messages and state in a distributed system that we call *Messages in Local Transactions* (MLT). Applications built around this model receive several important guarantees that help them recover from crashes:

- No data or messages from a committed transaction will ever be lost as a result of a system crash.
- Processes will never have to roll back state as a result of another node's crash.
- Message sends and receives are grouped atomically with local state changes.
- Orphan processes are never created.

The efficient implementation of systems based on our model depends on the availability of fast stable storage, such as the reliable memory provided by the Rio file cache [Chen96]. Such memory can provide the necessary durability for data and messages under our model, without the latency of disk writes.

To show that efficient systems based on the MLT model can be built, we have implemented an MLT-based system called *Vistagrams*. Vistagrams is an extension to our lightweight transaction system Vista [Lowell97]. Vistagrams imposes no overhead for the reliability it offers, thanks to the reliable memory provided by the Rio file cache.

2. Background and Related Work

The trick in restoring the state of a distributed computation after a crash is making sure that the state restored is meaningful. For example, consider the distributed computation depicted in Figure 1. In that computation, process A unlocks a distributed lock, and then sends a lock grant message to process B.

Consider a situation in which process A crashes at point 1, and then recovers to the time in its computation labeled 2 (perhaps because it wrote out a checkpoint at that time). The distributed computation would be in an *inconsistent* state because process B has received a message that process A does not remember it sent [Lamport78]. In this situation process B is said to be an *orphan*. The result of this inconsistency is that both processes could think they hold the lock.

One way to solve this problem is to roll the state of process B back to a point before it received the lock grant message, making its state consistent with process A's state.

This research was supported in part by NSF grant MIP-9521386, Intel, and Digital Equipment Corporation. Peter Chen was also supported by an NSF CAREER and Research Initiation Award (MIP-9624869 and MIP-9409229).

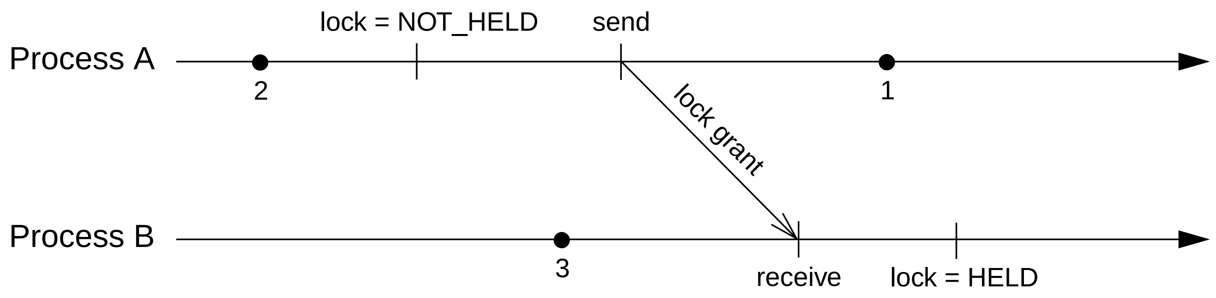

Figure 1: A space-time diagram depicting a distributed computation. Process A releases a distributed lock and grants that lock to process B.

Point 3 in the figure would be such a consistent state. The problem with rolling back processes is that doing so can lead to *cascading rollbacks*. If process B rolls back, it might roll back past a message sent to process A, requiring A to roll back further. But rolling back A could force B to roll back further still. This problem could conceivably result in uncontrolled rollback of the distributed computation to some initial state, a scenario called the *domino effect* [Strom85].

2.1. Existing recovery techniques

Researchers have developed several techniques to help distributed computations recover to consistent states while avoiding the domino effect.

In *coordinated checkpointing* [Chandy85, Koo87], processes in a distributed system employ a protocol to synchronize the action of taking local checkpoints. The synchronization ensures that the point in the global computation preserved by the local checkpoints represents a consistent state in the computation. If a crash occurs in the system after such a checkpoint, all nodes simply roll back to their most recent checkpoint without further regard for consistency. However, taking a coordinated checkpoint involves sending many messages at checkpoint time, and can thus delay normal processing.

Message logging schemes, such as sender-based logging [Johnson87], optimistic logging [Strom85], and the logging used in Targon/32 [Borg89], all record messages so they can be used to recompute the state of a distributed computation after a crash. All message logging systems assume the processes to be recovered are deterministic (i.e. given the same sequence of messages as input, they will compute the same result).

In sender-based logging, outgoing messages are logged in volatile memory so they can be resent to a failed process. A crashed process recovers by restarting its computation from a recent checkpoint, and then requesting that all messages sent to it since that checkpoint be resent. The downsides of sender-based logging are that it involves working nodes in the recovery of a failed node, it adds messages to the message transfer protocol, and it can only recover from a single failed node at a time. Manetho extends sender-based logging to be able to survive the failure of arbitrary numbers of processes [Elnozahy92].

In optimistic logging, incoming messages and checkpoints are recorded asynchronously on stable storage. Because checkpointing and message logging proceed asynchronously, the system may have to roll back working nodes when some other node fails. To determine which processes might need to roll back in a crash, the system has to keep track of causal relationships between processes. Optimistic logging can recover from the failure of an arbitrary number of processes.

In Targon/32, all messages sent in the system are pessimistically mirrored at relevant backup processes. If a process fails, its backup process is activated and brought into a consistent state by letting it play through its input queue of messages. Targon/32 relies on specialized interconnect hardware to provide efficient, 3-way, atomic message delivery.

Distributed transactions [Gray78] are also used to provide recovery for distributed systems. In distributed transactions, the decision to commit a transaction involves coordinating the commit of related transactions on other nodes. Several rounds of messages are needed to ensure that all the processes commit or abort together. Distributed commit protocols are similar to coordinated checkpointing, and like coordinated checkpointing, they can involve a significant amount of overhead at commit time.

2.2. Rio and Vista

As we will see in Section 3, an efficient implementation of our MLT model requires fast stable storage and fast transactions, such as that provided by the Rio file cache and the Vista transaction library respectively.

Like most file caches, Rio caches recently used file data in main memory to speed up future accesses [Chen96]. Rio seeks to protect this area of memory from its two common modes of failure: power loss and system crashes. While systems can protect against power loss in a straightforward manner (by using a $100 uninterruptible power supply, for example), protecting against software errors is trickier. Rio uses virtual memory protection to prevent operating system errors (such as wild stores) from corrupting the file cache during a system crash. This protection scheme does not significantly affect performance. After a crash, Rio writes the file cache data in memory to disk, a process called *warm reboot*. In essence Rio makes the memory in the file cache persistent. Chen et al. verified experimentally that the Rio file cache was as safe as a disk from operating system crashes.

Vista builds on the persistent memory Rio provides. Vista lets applications allocate chunks of persistent memory and perform atomic and durable transactions on that memory [Lowell97]. Vista provides atomicity by logging to persistent memory (provided by Rio) rather than to disk, and as a result Vista's transactions are extremely fast: small transactions can complete in under two microseconds.

As we will show, Rio and Vista combine to provide the perfect substrate for an MLT implementation: Rio makes writing to stable storage very fast; Vista makes local transactions almost free.

2.3. Failure model

Our MLT model and Vistagrams implementation of the model are designed to handle a specific class of failures.

Like most distributed systems, we assume processes follow the fail-stop model [Schneider84]. Specifically, we assume that processes do not commit transactions containing faulty memory stores or message sends.

As the reader will see in Section 3, our model writes to stable storage. Correspondingly, its reliability is as good as the stable storage employed. Users of our model who want their data to survive faults ranging from bit flips to nuclear bombs need merely employ a suitable storage subsystem.

Our Vistagrams implementation of MLT uses the reliable memory provided by the Rio file cache for its stable storage. Rio is geared towards providing safety from software failures (operating system and process crashes), which account for 90% of all failures [Gray91]. Furthermore, Rio can be extended to handle hardware failures by replicating data across multiple memories [Abbott94, Muller96, Gillett96].

Our work further assumes that failures in a distributed system will be transient. System crashes, packet loss, and network failures are all assumed to be temporary problems. In essence, we assume that if we send a message to a node enough times, it will eventually be delivered (even if it means continuing retransmission after a crash).

With this background for our work in place, we next describe our model for reliable, distributed computation that overcomes the drawbacks of existing consistent recovery techniques.

3. Messages in Local Transactions Model

A traditional atomic durable transaction groups a series of updates to persistent data into an indivisible unit. The transaction mechanism guarantees that either all the updates in a transaction will take place, or none will, and maintains this invariant even if a crash occurs in the middle of processing the transaction. In the Messages in Local Transactions model, we add message sends and receives to the class of operations that can be performed atomically and durably within a local transaction.

The Messages in Local Transactions model has two principle components. First, committed messages and data are not lost across system crashes. Second, sending and receiving messages is done atomically with the other operations in the transaction.[1]

An application using this model would group a series of persistent updates, sends, and receives into an atomic unit using a local transaction.

Once the local transaction commits, the application receives several assurances. First, the updates performed in the transaction are permanent. The messages sent in the transaction are guaranteed to be sent and not lost, even in the presence of crashes by either the sender or receiver. Furthermore, it is impossible for the sender to "forget" that a message has been sent: since the send and local state changes are done together atomically, the process' local persistent state will always accurately reflect whether the message was sent.

Second, if the transaction aborts, the messages "sent" within the transaction will not go out, and the state of the process' persistent data will be rolled back to its state at the beginning of the transaction. Any receives of messages performed within the transaction will be undone so that those messages will be delivered again during a subsequent receive.

Making message sends atomic requires deferring message sends until commit. An alternative is to send the message before commit, then abort the receiver if the sender aborts. However, this alternative requires senders and receivers to coordinate during commit. One immediate implication of deferring sends until commit is that an application cannot send a request and receive the response to that request in a single transaction. The application would instead have to commit the transaction in which the send of the request appears, and then begin another transaction in which to receive the response. Hence the application may need to recover to an increased number of points in the program.

4. Implications of the model

The Messages in Local Transactions model has several important ramifications for distributed systems.

4.1. Guarantees to the application

Applications built around the MLT model will never recover to an inconsistent state and will never create orphan processes. This is because local state changes are durable and done atomically with any related message sends. In the example in Figure 2, the marking of the lock 'NOT_HELD' by process A and the act of sending the lock grant message to process B would be done atomically in a local transaction. Once that transaction commits, all the operations in

1. Aspects of this model were proposed by Soparkar et al. as a construct for databases [Soparkar90]. They were not able to explore the implications fully because they lacked fast stable storage.

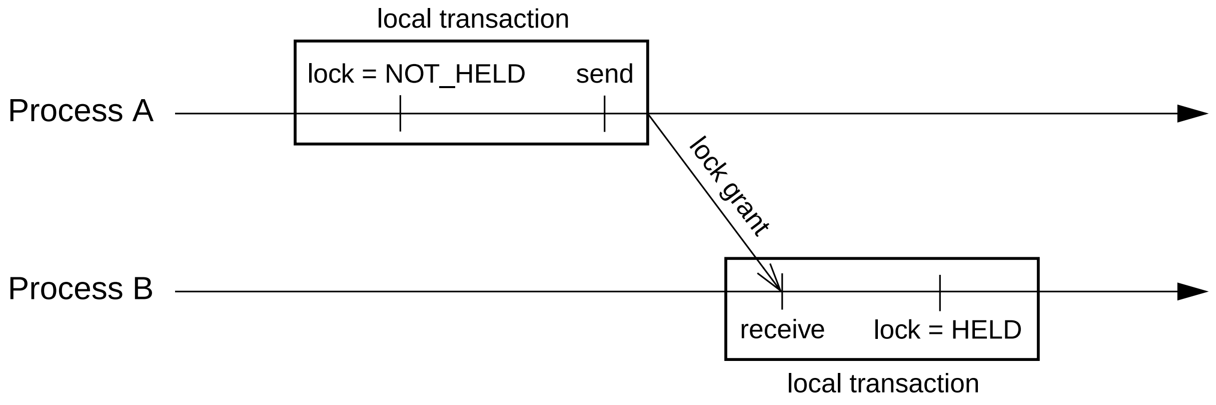

Figure 2: The same distributed computation as shown in Figure 1, except that this system uses the MLT model. Persistent state changes are grouped in local transactions with relevant message sends and receives.

that transaction are permanent and will not be lost if the system crashes. If that transaction aborts, the lock grant message will not be sent to process 2 and the lock variable will not be marked 'NOT_HELD' when the process recovers.

Similarly, the receiving of the lock grant message and marking the lock 'HELD' by process B would also be done atomically in a local transaction.

4.2. Comparison to existing systems

The MLT model provides applications the same guarantees of recoverability and consistency as other protocols. However, MLT lacks much of the complexity and overhead of other recovery systems.

First, since MLT does not require the state of the system to be recomputed from past input, MLT works perfectly well with processes that are not deterministic.

Second, when a process crashes under MLT, surviving processes are unaware of the fault (other than a delay while the crashed node recovers) and are unaffected. Surviving nodes are not involved in the recovery of failed nodes: they will never have to roll back to a prior state or be involved in any coordination of checkpoints to support recovery.

Third, all decisions by a process to commit transactions, send or deliver messages, or modify local state are strictly local under MLT. No distributed commit is needed. Nor is there any need to track causal dependencies between processes to maintain consistent recovery.

Fourth, recoverable distributed systems can be built in which only a subset of the nodes are designed around the MLT model. It is sufficient for processes to pass messages using the same reliable protocol, but to handle their own recovery using any appropriate method. On a related note, since every message that goes out on the wire represents committed data, there is no need to further delay messages to nodes outside the system that may or may not be recoverable (a printer for example).

Finally, systems built around the MLT model can tolerate the simultaneous failure of every node in the system because each node handles its own recovery. Furthermore, nodes can safely fail while recovering.

An interesting feature of MLT is its ability to group multiple message sends into atomic units, providing a relaxed variant of atomic multicast. In traditional atomic multicast, the sender is guaranteed that either all recipients will deliver a multicast message, or none will. When grouping multiple sends within a local transaction under MLT, the sender receives a related guarantee: if the sender's transaction commits, all destination processes will eventually deliver the message. If the sender's transaction aborts, none will deliver.

4.3. Protocol independence

MLT can be implemented as a part of any reliable protocol, without adding extra copies or messages. To see why this is so, consider that a reliable protocol must already buffer messages and retransmit them in case of packet loss. To provide an MLT service, a reliable protocol must simply perform its message buffering on stable storage, and retransmit messages until received, even over crashes. Hence, providing MLT service merely constrains *where* a protocol buffers messages and *when* it retransmits.

MLT implementations must also defer message operations in order to commit them atomically with local state changes. Deferring message operations again does not constrain the specifics of the reliable protocol employed. It merely constrains when the protocol is initiated.

5. Design and implementation of Vistagrams

We have implemented a system based on the Messages in Local Transactions model. Our system, which we call Vistagrams, is an extension to our lightweight transaction library Vista.

As shown in Figure 3, Vistagrams provides a request/response-based interface. Clients send requests to servers and get back responses, and can do work in between. Each request by the client and corresponding response from the server are correlated by a globally unique request ID.

```
vista_begin_transaction
vista_end_transaction
vista_abort_transaction

vista_send_request
vista_receive_request
vista_send_response
vista_receive_response
vista_select
```

Figure 3: Some of the routines in the Vista and Vistagrams API.

Although we refer to processes as clients and servers, any process can be a client or server in our system. In a request/response cycle the process initiating the request is called the client and the process that receives the request and sends the response is called the server. In a subsequent cycle, these roles may be reversed.

Our message delivery protocol is optimized for request/response interactions. The server's response confirms that it received the client's request, and the acknowledgment for the server's response is then piggybacked on a subsequent request. This protocol preserves the common case of a total of two messages used in a reliable request/response cycle between client and server.

At a high level, Vistagrams has several key tasks to attend to.

First, Vistagrams has to buffer outgoing messages persistently so it can retransmit them after packet loss or system crashes.

Second, Vistagrams has to defer many operations until commit, such as sending messages, freeing buffers, removing requests and responses from tables, and enqueueing acknowledgments to be sent to servers. These operations are deferred by adding them to various operation logs that are played back at commit.

Third, Vistagrams must commit outgoing message sends atomically with updates in the surrounding transaction. Committing atomically involves setting a single flag that marks the transaction committed, and then performing all the deferred operations. Those operations are carefully designed to be idempotent so that they can be redone if a crash occurs during log playback.

Finally, Vistagrams has to keep retransmitting committed messages until they are received, possibly after a crash by the sender or receiver.

Because Vistagrams messages are retransmitted over crashes, and because the information needed for the recipient to filter messages is persistent, Vistagrams provides efficient, *exactly-once* message delivery. Exactly-once delivery guarantees that a sent message will be delivered exactly one time by the receiving process. Providing such a semantic efficiently has been acknowledged widely to be difficult, because system crashes can cause processes to forget which messages have already been delivered [Lampson93].

5.1. Vistagrams implementation

To show the structure of our Vistagrams implementation we will follow a request message from the client to the server and back.

Client sends request: When the client makes a call to `vista_send_request`, the outgoing message is assembled in a persistent buffer and added to the request table. The request table is the main client-side data structure. It is a hash table of vistagrams (hashed on request ID). It is used to correlate requests and responses, as well as to buffer messages for retransmission, or responses that are received out of order. Once the new request is in the request table, it is added to a log of messages to send when the enclosing transaction commits. At commit, the request is sent to the server using UDP.

Server receives request: The server waits for a new request in `vista_receive_request`. When it receives a request, it adds an entry to the response table. The response table is the central server-side data structure. Like the request table, it is a hash table hashed on request IDs. It is used to record the return address for a response, as well as to buffer outgoing response messages. After adding an entry to the response table, the incoming request message is returned to the user.

Server sends response: The server processes the request and then sends a response by calling `vista_send_response`. `vista_send_response` builds the outgoing message using the return address stored in the response table. It then persistently buffers the response in the appropriate response table entry (in case the response message is lost and needs to be retransmitted), and adds the response message to the log of messages to be sent at commit. When the server's transaction commits, the response message is sent from the server to the client.

Client receives response: The client calls `vista_receive_response` to receive the response message. The `vista_receive_response` function waits up to a fixed interval for the response to arrive. If the response does not arrive in that time, `vista_receive_response` sends the request to a special retransmit port on the server, and the routine waits again. Clients can retransmit indefinitely.

Once the response arrives, the response message is copied into a user-supplied buffer. An ack for the response is added to a special "ack queue" so that if the surrounding transaction commits, the response will be acknowledged during a subsequent request to the server. In addition, entries are made in appropriate operation logs so that the request table entry and its buffered request message are both deleted on commit.

Retransmission of messages: The server maintains retransmit ports that receive any retransmitted requests so that they can be handled specially. The server is notified asynchronously via a signal when a retransmitted message arrives. When a request arrives on one of those ports, the signal handler checks to see if the server already knows about the request by looking up the request ID in the response table. If the table contains an entry with a committed response message, that response is resent to the client

Client

```
            .
            .
            .
vista_begin_transaction();
   lock->status = PENDING;
   vista_send_request(lock_req);
vista_end_transaction();

vista_begin_transaction();
   lock->status = LOCKED;
   vista_receive_response(lock_grant);
vista_end_transaction
            .
            .
            .
```

Server

```
main()
{
   /*
    * Vista setup and allocation of
    * persistent data...
    */

   for (;;) {
      vista_begin_transaction();
         vista_receive_request(req);
         process_request_in_transaction();
         vista_send_response(lk_grant_msg);
      vista_end_transaction();
   }

}
```

Figure 4: A sample recoverable distributed lock manager using Vistagrams. Pseudocode is used in the parameter lists for brevity. The client is shown at some point in its execution requesting a lock from the server.

(presumably the original response message was lost). If the entry in the response table does not contain a committed response, the retransmitted request is dropped. This situation could arise when the server has received the request, but not yet sent or committed the response. Finally, if the response table does not already contain an entry for the retransmitted request (which could occur if the original request message was lost), the request message is forwarded to the waiting server port on the same host.

5.2. Sample application

Figure 4 shows a simple recoverable distributed application built with Vista and Vistagrams.

The client is shown at some point in its execution requesting a lock from the server. The client is forced to break up its lock request into two transactions because the sending of the lock request is deferred until commit. If the client should crash, it will have to check during recovery if the lock is in a “PENDING” state, and if so redo the second transaction. The server is completely recoverable as is, and needs to do no extra work to recover after a crash.

6. Vistagrams performance

Vistagrams are designed to be very fast. Our Vistagrams implementation does not add any messages to the standard reliable request/response protocol used for message delivery. Nor does our Vistagrams implementation add any copies to the process of sending messages through the protocol. All the persistent buffering and logging that Vista and Vistagrams do to ensure atomicity and durability of messages and data is done using persistent VM provided by the Rio file cache. No disk I/Os are done during the course of performing a transaction or sending a message. We would expect that avoiding extra messages, copies, and disk I/Os would translate into a very fast system.

6.1. Microbenchmark

To evaluate our claims about Vistagrams performance, we use a simple distributed application that sends requests of varying sizes to a server and receives responses from that server. The server simply sends the request data it receives back to the client. We use two versions of this distributed application.

Our baseline system is a volatile and non-recoverable implementation of this system. Like Vistagrams, it retransmits messages to handle packet loss, and it correlates requests and responses. It does not use local transactions to send and receive messages atomically with local state, and messages and data are lost after crashes. We compare this baseline system with a fully persistent and recoverable implementation, built using Vista and Vistagrams.

Both versions of the application use identical reliable request/response protocols for message delivery. Hence by comparing the round-trip request/response times for each of these two systems, we get an accurate picture of the cost of making the sample application recoverable using Vistagrams and Vista transactions.

6.2. Environment

We run our benchmark on two Intel-based PCs, each with a 266 MHz Pentium II CPU and 128 Megabytes of memory. The systems are connected via a switched 100Base-T network, using an Intel Express 10/100 Fast Ethernet Switch. The server and client processes run on separate machines.

6.3. Results

Figure 5 shows the results of our benchmark runs. The plot shows the time for a request/response cycle for each of our two test systems. For reference, we also show the round-trip time to send a UDP message of the given size from client to server and back.

As expected, Vistagrams adds almost no overhead to the baseline system (the Vistagrams and baseline curves are coincident). A Vistagrams request/response cycle takes 2-6 microseconds longer than a non-recoverable request/response cycle. This is a negligible fraction of the 240-2300 microsecond round-trip time. Vistagrams achieves high performance by avoiding extra copies and messages, and by using the reliable memory and fast transactions provided by Rio and Vista.

7. Future Work

The MLT model and Vistagrams have two principle drawbacks. First, applications have to be written using transactions. Transactions can be difficult to retrofit into existing applications. Second, as mentioned in Sections 3 and 5.2, applications might have to commit transactions earlier than desired because sends are deferred until commit. These early commits can in some instances complicate recovery because it may force an application to end a transaction in the middle of a logically atomic series of steps. Hence the application must be able to recover to an increased number of points in the program.

We are working on a lightweight checkpointing system that addresses both of these issues. By transparently performing local-process checkpoints synchronously with every message send, this system provides consistent recovery for distributed systems without the burden of using transactions or handling any aspects of recovery.

Since our MLT model is protocol independent, it might be interesting to use MLT with protocols that provide richer guarantees to the application, such as causal delivery, totally ordered delivery, or atomic multicast.

8. Conclusion

Writing distributed applications that reliably manage persistent data involves navigating a host of thorny issues, such as ensuring consistent recovery while avoiding the domino effect. Furthermore, many of the traditional techniques for recovery of distributed applications have not been widely adopted in the field because of performance issues, or because of the limited scope of applications those techniques support [Birman96].

The advent of reliable memory and fast transactions have created new options for recovery of distributed systems. Our Messages in Local Transactions model guarantees that applications will not lose persistent data or messages as a result of a system crash, surviving processes will never have to roll back, and systems will always recover to a consistent point in the computation. Furthermore, systems designed around our MLT model can be high performance, and can be nondeterministic.

We have shown that it is possible in practice to build an efficient system based on MLT. Our Vistagrams system[2] provides MLT service while achieving the same performance as an ordinary reliable-messaging protocol.

2. Vista, Vistagrams, and the Rio version of the FreeBSD kernel are available at http://www.eecs.umich.edu/Rio.

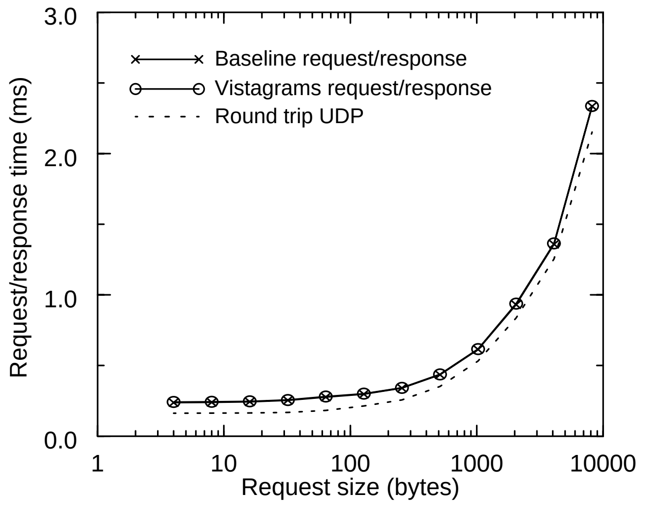

Figure 5: Vistagrams performance. This graph shows the time for a reliable request/response cycle between client and server with requests and responses of varying size. The baseline system is a non-recoverable system with volatile messages and data. The Vistagrams system is fully persistent and recoverable. Note that the Vistagrams and baseline curves are coincident. For reference, we show the round-trip time to send a UDP message of the given size from client to server and back.

9. Acknowledgments

We would like to thank Farnam Jahanian, M. Satyanarayanan, Nandit Soparkar, and Willy Zwaenepoel for their timely insights that helped with the development of our thoughts in this paper.

10. References

[Abbott94] M. Abbott, D. Har, L. Herger, M. Kauffmann, K. Mak, J. Murdock, C. Schulz, T. B. Smith, B. Tremaine, D. Yeh, and L. Wong. Durable Memory RS/6000 System Design. In *Proceedings of the 1994 International Symposium on Fault-Tolerant Computing*, pages 414–423, 1994.

[Birman96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications, 1996.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrman, and Wolfgang Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Chandy85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States in Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Chen96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, October 1996.

[Elnozahy92] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.

[Gillett96] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[Gray78] J. N. Gray. *Operating Systems: An Advanced Course*. Springer-Verlag, 1978. Notes on Database Operating Systems.

[Gray91] Jim Gray and Daniel P. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[Koo87] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lamport78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson93] Butler W. Lampson. *Reliable Messages and Connection Establishment*. Addison-Wesley, 1993. in Distributed Systems, edited by Sape Mullender.

[Lowell97] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, pages 92–101, October 1997.

[Muller96] Gilles Muller, Michel Banatre, Nadine Peyrouze, and Bruno Rochat. Lessons from FTM: An Experiment in Design and Implementation of a Low-Cost Fault-Tolerant System. *IEEE Transactions on Reliability*, 45(2):332–340, June 1996.

[Schneider84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Soparkar90] N. Soparkar and A. Silberschatz. Data-Value Partitioning and Virtual Messages. In *Proceedings of the 1990 ACM Symposium on Principles of Database Systems*, pages 357–364, April 1990.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.