Deterministic Replay for Transparent Recovery in Component-Oriented Middleware

Rob Strom, Chitra Dorai
IBM T.J. Watson Research Center
Hawthorne, NY, USA
robstrom@gmail.com, dorai@us.ibm.com

Thomas Huining Feng, Wei Zheng
Dept. of EECS, UC Berkeley
Berkeley, CA, USA
{tfeng, zhengwei}@eecs.berkeley.edu

***Abstract*—We present and evaluate a low-overhead approach for achieving high-availability in distributed event-processing middleware systems consisting of networks of stateful software components that communicate by either one-way (send) or two-way (call) messages. The approach is based on transparently augmenting each component to produce a deterministic component whose state can be recovered by checkpoint and replay. Determinism is achieved by augmenting messages with virtual times, and by scheduling message handling in virtual time order. Scheduling delays are reduced by computing virtual times with *estimators:* deterministic functions that approximate the expected real times of arrival. We describe our algorithms, show how Java components can be transparently augmented with checkpointing code and with good estimators, discuss how our deterministic runtime can be tuned to reduce overhead, and provide experimental results to measure the overhead of determinism relative to non-determinism.**

Keywords- recovery, determinism, replay, component-oriented middleware, high availability, virtual time

I. INTRODUCTION

We present an approach for transparent recovery of component-oriented, distributed event processing middleware, based on automatically augmenting components with *virtual times* to generate an implementation that can be executed with repeatably deterministic behavior which can be made fault-tolerant using checkpoint-replay. We have implemented the system, tested its correctness, measured the performance of our implementation and studied under simulation how the performance is sensitive to various control parameters. We conjecture that our approach simplifies development and provides a low-overhead alternative to other approaches to recoverability, such as transactions plus entity beans and/or object caches.

A. Background

Component-oriented middleware is extensively used in event processing, stream processing, sensor networks, and business logic. Systems such as WebSphere Process Server, ESB, and Message Broker [1] employ component models for mediation components, transformation components, and business logic components. In all these systems, components receive events, respond to service requests, perform computations, and generate new events. Often components keep state in order to correlate events from different sources or to average or aggregate events, or to look for trends or to detect patterns in sequences of events. Components may be written in specialized languages, or in general-purpose languages like Java. Some Java component models are specialized for real-time needs, such as Exotasks [2] and PTides [3].

When machines or communication links fail, any of these component-based applications can break due to lost state or missing messages. This paper is concerned with low-overhead techniques for masking these failures, allowing these applications to continue executing correctly. In particular, we propose to recover state by using checkpointing and replay. We propose to enable replay to work by designing the runtime to force components to execute deterministically. Alternate approaches to recovery are discussed in Section IV.

B. Summary of Approach

We have implemented a new approach for lightweight transparent recovery within a distributed Java-based system called TART (Time-Aware Run-Time). Unlike active replication systems, TART keeps only one active replica of each component, intermittently sending state updates to passive replicas, which perform computations only when the active replica fails. Unlike transactions or optimistic recovery, the only logged messages are those that arrive in TART from external producers. Inter-component communications are not logged.

Networks of components, although they have an intrinsically non-deterministic specification, are forced to execute deterministically by virtue of having all messages augmented by a single timestamp or *virtual time*. Components are transparently augmented with additional deterministic code to compute these timestamps. As in a discrete event simulator, messages arriving at components are processed deterministically, in timestamp order, rather than non-deterministically in order of actual arrival. Because of this determinism, if a component is rolled back to an earlier checkpoint due to a failover, and messages are replayed, it will reach the identical state.

Unlike discrete event simulators, the timestamps are not supplied by users, but are automatically generated by the system by estimator functions that try to produce virtual times that are close to real time. Although this does not affect correctness, it improves performance by increasing the likelihood that messages that are supposed to be executed by a component in a particular order arrive in that order.

Because it is transparent, TART simplifies the development process. Developers write components in Java with few restrictions. State need not be stored in special objects, but instead in ordinary instance variables. The only restrictions are those that must be enforced to guarantee that components don't inadvertently share state. These restrictions can be enforced statically by Java dialects such as Guava [7].

II. How Deterministic Recvery Wor

In this section we discuss the environment for which TART is designed, the correctness criterion, the core features of the approach, and the tuning parameters that control performance.

A. Application Model and Correctness Criterion

A component-based application consists of a network of components that include at least one external producer of input, and at least one external consumer. Figure 1 shows the simplest non-trivial instance of an application in our model. Two components, *Sender1* and *Sender2*, receive asynchronous input from external sources, perform some processing, and then send an event to a service called *Merger*. *Merger* services the message, and based on its state, delivers an external output. In our model, components execute in parallel, and may handle either asynchronous (one-way) messages or service calls (two-way). Each component handles one message or one service call at a time with no internal concurrency. Any component may keep state that persists across calls.

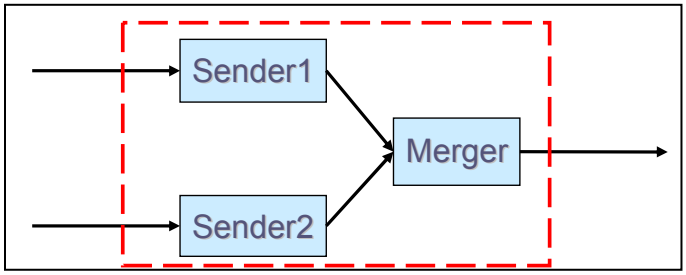

Figure 1. A simple application with two sender components fanning into a single merge component.

Notice that the specification is non-deterministic: for instance, *Sender1* and *Sender2* compete to send event messages to *Merger*, which may dequeue them in any order. In a Java implementation, *Sender1* and *Sender2* would run in separate threads, invoking a synchronized method within *Merger*. These calls are serviced in an undetermined order, based upon factors such as the speed of the threads and the communication delays.

Our correctness criterion for recoverability states that when an application is deployed onto a distributed system of machines, then: despite fail-stop failures (causing one or more machines to stop, losing all state and all messages in transit), and link failures (causing loss, re-ordering, or duplication of messages sent over physical links), the behavior of the application will be the same as the behavior of some correct execution of the application in the absence of failure, except for possible output stutter. Output stutter means that a system may roll back and re-deliver some already delivered external messages. For systems with monotonic output, e.g., output containing a sequence number or other non-repeatable ID, output stutter can be ignored, since external clients can easily compensate for its effects. All communication in our model is, guaranteed to be reliable, FIFO, and fair.

B. Components

We will assume that components are written in Java, since that is how our system was implemented. However, our discussion will apply, *mutatis mutandis*, to other languages or to multi-language component-based systems.

A component is any piece of software that: (a) receives input requests, which can be viewed as messages or as method invocations, (b) performs processing in response to messages, (c) possibly holds state, and (d) possibly sends messages: either by one-way asynchronous delivery, or by bidirectional service calls with response. Components can be arbitrary Java programs, with these restrictions:

- Components do not share memory, but interact only through one-way sends or two-way calls.
- There is no internal concurrency within a component; components may not invoke explicit multi-threading, and all methods are synchronized. There is one logical queue of all messages waiting to enter a component.
- Components may not invoke non-deterministic operations such as a system call to query the amount of free memory. One exception to this principle is timing services: for example, a component may request the current time, because this call is implemented by retrieving the current deterministic virtual time.
- Components do not block for reasons other than waiting for a return from a service call.
- For this paper, we assume that the code and wiring of the components are known prior to deployment (i.e., there is no dynamic rewiring or code generation).

C. Deployment

Components of an application, such as that of Figure 1, originally have no affinity to any particular execution engine; any component can potentially run anywhere. When the application is deployed to a set of execution engines, the following steps occur:

- A placement service assigns individual components to execution engines within the distributed system. An execution engine is either a physical machine or a container such as a JVM within a machine.
- The components are transformed: All message interfaces are augmented to include an additional parameter representing the virtual time that the message will be processed at the receiver; code bodies are augmented with deterministic estimator functions to compute these virtual times; code bodies are additionally augmented with methods to capture and restore checkpoints of component state. The core of TART's recovery mechanism is the transformation of a logically non-deterministic application into a deterministic one that uses checkpoint-replay to recover state. In this paper we discuss how we generate good estimators and we evaluate the effect of estimators and other design decisions on performance.
- Each engine is associated with a backup, which is either a stable storage device for holding checkpoints, or a passive replica residing on a separate execution

engine, which holds checkpoints, ready to immediately become active should the active engine fail.[1]

D. Time, Ticks, Silences

Each scheduler is responsible for keeping track of virtual time, which is discretized into *ticks*. At each tick, some event happens, such as dequeuing a message, computing, sending a message, waiting, etc. Virtual time is intended to be an approximation of real time within close bounds, but for correctness, it is only necessary that (a) causally later events have later virtual times, and (b) the computations of virtual time are deterministic: i.e., given the same input event dequeued at the same virtual time, and the same state, the same computation will be produced, including the virtual times generated for any output messages created.

Each engine tracks virtual time for the components it contains. Each wire between components contains a stream of events. Ticks not containing messages are said to be *silent*.

Unlike Jefferson's Time Warp algorithm for discrete event simulation [8], in which messages are optimistically processed first-come first-served, and then rolled back and re-executed if out-of-order messages arrive, TART's scheduling algorithm is pessimistic: a scheduler processes input messages in strict virtual time order without rollback. Any receiving component receiving messages from more than one sender must know whether a given message has the earliest virtual time. The sending components will need to communicate information about silence ticks as well as message ticks. This communication may travel with the messages, may be sent asynchronously, or may be retrieved lazily, on demand. Silence tracking is adapted from monotonic models [9]; they are the monotonic version of null messages.

E. Receiving and Scheduling Messages

When a message arrives at the system from an external source, it is (a) given a timestamp, and then is (b) logged – either to external stable storage, or to the backup machine. Because the message is logged, it is safe to use the actual real time as the virtual time of this message. Only external messages are logged.

Suppose that messages arrive at *Sender1* and *Sender2* at times 50000 and 80000, respectively. Since *Sender1* was idle, it immediately dequeues the message and begins the computation. Suppose each *Sender*[i]'s specification is to receive inputs containing sentences, remember a count of how many times each word has been encountered, update this count, and output a message containing the total count for this sentence of the number of times the words in this sentence have been encountered. Code Body 1 shows the processing method of *Sender*[i], prior to transformation by the system:

At deployment time this code is automatically transformed: the signature of `port1.send` is augmented to contain a second parameter representing the virtual time at which the message will arrive at the receiving component's queue, and the code is augmented to compute this virtual time using an estimator function. The estimator computes the output message's virtual time by incrementing the virtual dequeue time of the input message by an estimate of the sum of the computation time and the transmission delay time. Any estimator that yields a virtual time in the future will be correct, but TART's performance is if the estimated arrival time is often a close approximation to the real arrival time. (We will demonstrate this in our experimental study.) The estimator function code might be the following (assuming that *Sender* and *Merger* are in the same JVM and hence that communication delay is negligible):

```
outVT = inVT + 61000*sent.length;
```

Since the code is a loop with a variable number of iterations, a good estimate is the iteration count times the expected number of ticks per iteration. (In our implementation, a tick is a nanosecond.) Static analysis can give an approximate estimate of the correct coefficient; we shall see later that after several hundreds of messages have been processed, the coefficient can be refined based upon empirical measurement.

Code Body 1. Message processing code body for *Sender*[i].

```
public void processSentence(String[] sent) {
   int count = 0;
       for (int i=0; i < sent.length; i++) {
           String word = sent[i];
           Integer wordCount = map.get(word);
           if (wordCount == null) {
             wordCount = 0;
           }
           map.put(word, wordCount+1);
           count += wordCount;
       }
       port1.send(count);
 }
```

Suppose that the sentence lengths are 3 for the message sent to *Sender1*, and 2 for the message sent to *Sender2*; then the messages sent to *Merger* will have respective virtual times of 50000+3*61000 = 233000, and 80000+2*61000 = 202000.

In our example, *Sender1* reaches a virtual time of 233000, sends its message communicating silence for ticks 0 through 232999, and a message at 233000. It then is ready to receive a new message. The dequeued virtual time of that new message will be the maximum of its virtual time and 233000. *Sender2* will behave similarly.

In a conventional (non-deterministic) Java implementation, these two messages will usually arrive in virtual time order, but might occasionally arrive in the other order (due to variations in relative speeds of the threads due to cache misses, page faults, and other factors). In a conventional JVM, Merger will process the messages in their actual arrival order, leading to non-determinism. But in TART, we force *Merger* to process the messages in virtual time order – i.e. *Sender2*'s first.

Now let us look at the *Merger* component. It must process the messages in virtual time order regardless of their actual arrival order. Whichever message arrives first, it must examine the other competing sender wires to make sure that there is no earlier message. Because TART uses pessimistic scheduling, the *Merger* component's scheduler must delay an earliest received message with time *t* until it knows that all other

[1] In the remainder of the paper, we will assume the special case in which recovery from a single failure is adequate, and that therefore each execution engine is associated with a single passive replica residing on a different machine.

senders with connections to this component have promised silence through time *t*.[2] In this example, *Merger*'s scheduler will process the message from *Sender2* at time 202000 once it learns that *Sender1* is silent through that time. As we shall see, different variants of our algorithm will provide different mechanisms for speeding up the propagation of silence. The delay in holding a message waiting for other senders to either send messages with earlier virtual times or to promise silence is called pessimism delay. Pessimism delay is the principal contributor to the overhead of deterministic scheduling. TART is practical provided that this overhead can be minimized to the point where it is cheaper to use determinism plus checkpoint-replay to recover state rather than some more expensive mechanism such as transactions. The measurement section assesses various tuning factors and their effect on pessimism delay.

F. Fault-tolerance

The design point of TART is to provide low overhead and good performance during failure-free operation, and correct behavior after failures. We discuss here the work TART performs to achieve fault-tolerance both in normal operation, and after failure:

1) Tick tracking:

Time is discretized into *ticks*. Each tick on a communications channel between components is accounted for either as a data tick, or as a *silence* [9].

2) Logging and Checkpointing:

Messages received from the external world are logged, in case the first execution engine loses state and requires replay. Messages sent between executions do not need to be logged.

During normal operation, an execution engine intermittently takes a "soft checkpoint" of the state of its components. Soft checkpoints are asynchronously sent to the designated passive replicas. A passive replica only holds the state; it need not do any processing. The checkpoint frequency is a tuning parameter: more frequent checkpointing reduces recovery time but increases overhead.

The code is augmented as follows: (1) For large structures like hash tables needing incremental checkpointing, updates since the last checkpoint are stored in an auxiliary structure; (2) a method is provided to gather all full checkpoint state and all incremental changes and to return them to the scheduler, which then serializes them and sends them to the partner.

3) Failover:

If an engine fails, its passive backup becomes active. The checkpoint is restored, and connections are made to sending engines. The checkpoint is likely to be in the past, but then the sending engine will be asked to replay messages.

4) Replay:

If either due to failover, or due to message loss on links, a gap is detected in the sequence of ticks, the sender or senders will be prompted to resend the range of ticks for which there is a gap. If the "sender" is an external component rather than another TART component, then the messages are re-sent from the log. Messages that were only delayed can appear to be lost; therefore replaying them can occasionally cause duplicate messages to arrive, but the duplicate messages will have duplicate timestamps and will be discarded.

G. Controls Affecting Performance

Failure-free performance is affected by checkpoint frequency, and by pessimism delay. Checkpoint frequency is usually tuned to provide the desired level of time to recover. But pessimism delay is an intrinsic overhead needed to assure determinism. There are numerous factors that can affect pessimism delay. They can be divided into those tunable within TART's runtime, and those external to the runtime.

The external factors are all the things that increase the variability of actual execution time of a program having a known virtual execution time. They include: hardware factors such as memory contention, CPU contention, and cache misses; software factors such as operating system interrupts, page faults, thread scheduling (if threads compete for processors); and language runtime factors such as garbage collection, memory allocation, and data structure reorganization. Some of these sources of variability are addressed in algorithms such as Java real-time garbage collectors [10]. In these environments, there are independent reasons for providing low-variability real-time response (also, and sometimes confusingly, called "determinism").

There are factors within the TART execution environment that can affect pessimism delay, and these factors are the most important to control to get good performance, since the external factors may not be under our control. They are the following:

1) Quality of Estimators:

Estimators can range from very crude to very smart. The crudest estimator just estimates a fixed average computation time per message. Smarter estimators, as in our example, derive an estimate based upon one or more other parameters, such as number of times different blocks within the code are executed. Of course, we don't want an estimator to have so many parameters that the cost of computing the estimate significantly adds to the overhead. We have just begun to explore how sensitive performance is to the quality of the estimator.

Estimators are also required for communication delay between components in remote machines. Once again, a crude estimate can be just a constant based upon expected communication delay. Alternatively, it can be a function based upon expected queuing delay. To be deterministic, it cannot depend upon non-deterministic state such as the current queue size. It must instead use deterministic factors that correlate with queue size, such as the number of messages sent within a recent number of virtual ticks of time. Refining such estimators is a matter for future work.

Finally, it is known that when several senders are sending messages at different rates, then in the absence of aggressive silence propagation protocols, it is actually better for the virtual time estimates not to exactly match real-time, but rather for the process that is slower on the average to eagerly promise more silence ticks and delay the next data tick to be after that range of silence ticks, to improve the chance that messages from the faster process will not be delayed. This so-called "bias algorithm" [11] was designed to reduce pessimism delay in a simpler system that just performed deterministic merges without any intervening computation.

[2] In the rare event that messages from two different schedulers arrive at the identical time, there must be a deterministic tie-breaking rule, e.g. using ID numbers of the wires to break ties.

2) Sensitivity of Schedulers:

If some threads are consistently generating estimated virtual times ahead of or behind real time, it is possible that they will increase the overhead. Dynamically changing the priority of these threads to slow down the fast threads or speed up the slow ones may improve overhead. An alternative, discussed below, is to take a determinism fault and dynamically adjust the estimator.

3) Silence Propagation Techniques:

The most naïve treatment of silence is "lazy silence propagation". In this approach, if a component sends a message at time t_1, no silences are sent until the next message at time t_2, at which time silence are sent for the ticks from t_1+1 through t_2-1. This can often cause considerable pessimism delays. Other approaches involve "curiosity-driven silence", in which a receiver that is engaged in a pessimism delay explicitly requests the sender to compute a new silence interval, and "aggressive silence", in which senders that have not sent silence for some time explicitly send it without asking. The approach described above of a component known to have a slow data rate promising extra silences even at the cost of forcing future messages to have later virtual times is called "hyper-aggressive silence propagation", since it eagerly marks certain ticks as silent before knowing whether they normally would be silent or not.

4) Determinism Faults and Dynamic Re-tuning:

If the system consistently has virtual time out-of-sync with real time and is suffering excessive pessimism delays that cannot be fixed by changing the priority of threads, it may be necessary to re-calibrate the estimators. Since detecting and reacting to such a condition non-deterministically affects virtual times, we must treat such a situation as an exception to the determinism principle – a *determinism fault.* In order for replay to work correctly in the presence of determinism faults, we must log these events synchronously. Therefore, determinism faults are an extra overhead whose frequency we expect to minimize. For example, suppose in our earlier example, we decide at some point that the coefficient should have been 62000 ticks per iteration instead of 61000. We recalibrate the estimator only after logging the virtual time of the change, e.g. time 100,000,000. During replay, the component must be careful to use the old estimator until reaching time 100,000,000, and only then using the new estimator.

It should be noted that lazy, curiosity-based, and aggressive silence propagation techniques can be arbitrarily mixed and/or dynamically changed without requiring a determinism fault. This is because while the identities of the silent ticks depend only on the estimators, changing the way in which this silence is communicated has no effect on the actual behavior. On the other hand, sending hyper-aggressive silence affects which future ticks are allowed to contain data; changing the rules for hyper-aggressive silence affects the estimator and hence requires a determinism fault.

H. Strategies for Silence Propagation and for Deriving Estimators

We derived estimators for computation time based upon the assumption that computation time is a linear function of how many times each basic block executes. For example, in Code Body 1, let ξ_1 be the number of times the loop executes, and ξ_2 be the number of times the conditional executes. The calls to `get` and `put` are assumed to take unit cost, so there are no additional parameters. Then the computation time Δt is given by

$$\Delta\tau = \beta_0 + \beta_1\xi_1 + \beta_2\xi_2 \ . \qquad (1)$$

Before execution, a rough estimate of the β_i's is made based upon known costs per instruction. Later, after some execution samples are taken, measuring ξ_1, ξ_2, and Δt, a linear regression is taken to fit the coefficients.

We executed Code Body 1[3] 10,000 times with random numbers of iterations between 1 and 19 on a machine not particularly designed for low-variability real-time execution: a ThinkPad T42 running Windows XP Version 2002, and JDK 5.0. Figure 2 shows the distribution of execution times. Because the conditional and send statement contributed so little to the total execution time, we fitted only the single coefficient β_1, which yielded 61.827 microseconds, or 61827 ticks, as shown in Equation (2), with a reasonably good R^2 of 0.9154. The distribution of the residuals is highly right-skewed. There is close to zero correlation between the number of iterations and the residuals (hence a good linear fit).

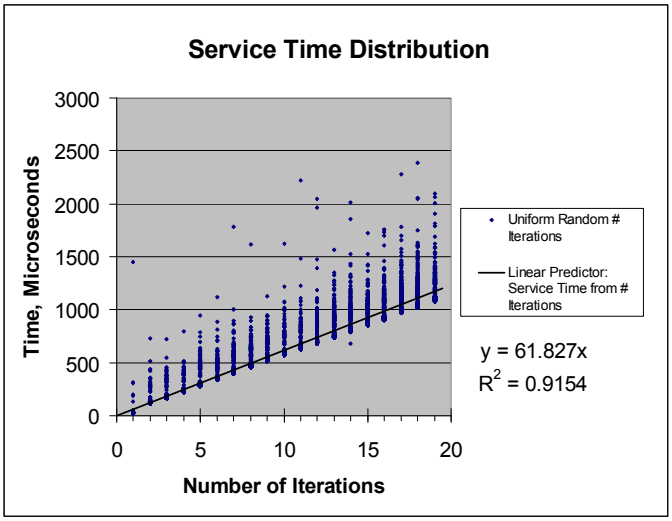

Figure 2. Computation time as a function of number of iterations.

$$\Delta\tau = 61827\xi_1 \ . \qquad (2)$$

In the subsequent section we show how we validated this estimator and measured how sensitive total system performance is to inaccuracies in this estimator.

For silence propagation, we used a curiosity-based technique. The receiver tracks the silence ranges for all senders. When it is time to dequeue a message and the earliest message has time t, and the other senders are not silent through time t (i.e., we are in a pessimism delay situation) a curiosity probe is sent to the other senders. They then compute the silence range. If the sender is idle, it is silent through the virtual time it be-

[3] To get more useful execution times given the accuracy of our real-time clock, each iteration of the loop was executed 300 times.

came idle plus the computation time of the shortest possible processing (i.e., one tick earlier than the earliest possible time it could deliver a message were it to become busy one tick from now). If the sender is busy, it computes the earliest possible time it could compute a message based upon the known state of the process. Notice that for programs such as Code Body 1, the number of iterations of the loop is known before the first iteration is executed; in other programs containing loops, this might not be the case.

III. Experimental Results

We performed a number of experimental studies, both with simulated and with real systems, to measure the overhead of deterministic recovery and the factors influencing this overhead.

A. *Overhead as a Function of Variability, Prescience, Crudeness of Estimator*

In the first study, we simulated the system of Figure 1 for the case of a multiprocessor system. In this configuration, the three components each had a dedicated thread, with each thread executing in its own processor. The *Sender*[i] components each executed code similar to Code Body 1: a loop that took 60μs of virtual time per iteration, and took an average of 10 iterations per message. The fluctuation of real time was modeled by having the program progress each virtual tick by an amount of real time governed by a normal distribution with mean of one tick and a standard deviation of 0.1 ticks. (This was an unrealistic approximation; in reality the distribution of the difference between real time and predicted virtual time is much skewed.) Curiosity probes were assumed to take 20μs (probably an over-estimate, hence unfavorable to TART's algorithm). External clients fed messages into the *Sender*[i] components via a Poisson process with average inter-arrival time of 1 msg/1000μs at each Sender. The Merger component had a fixed processing time of 400μs per event received. Thus, each Sender[i] processor was on the average 60% utilized (600μs/1000μs); the Merger processor was on the average 80% utilized (400μs/500μs).

We varied the variability of the *Sender*[i] processors by stages from constant (every invocation called for 10 iterations) to variable with uniform random distribution of from 1 to 19 iterations. We simulated three modes of execution:

Non-deterministic: The *Merger* processes messages in real-time arrival order.

Deterministic: The *Merger* processes messages in the order of the virtual times, sending curiosity probes to elicit information about silent ticks whenever it detects a pessimism delay. On receiving a curiosity probe, a sender delivers information about silences, but if it is executing a loop, it is assumed not to know how many more iterations will follow.

Prescient: Same as above, except that on receiving a curiosity probe, a busy sender knows (because the code computes the iteration count prior to entering the loop) how many remaining iterations it must execute before it can send a message, and therefore generates a more accurate estimate of which future ticks are guaranteed to be silent.

We tracked each message as it entered and executed each queue. In the deterministic mode we also tracked the time message were held due to pessimism delay. We counted the number of out-of-order messages, the number of curiosity probes, and the average end-to-end latency.

Figure 3 compares the three execution modes. Notice that variability in the service time of the Sender processes affects latency (greater variability leading to greater latency), but does not significantly affect the percentage overhead for determinism, which ranged from 2.8% to 4.1% for non-determinism with curiosity without prescience, and only slightly better for non-determinism with prescience.

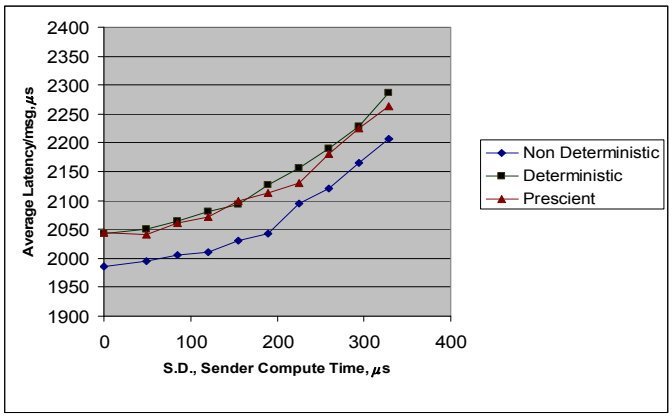

Figure 3. Latency as a function of variability of sender computation.

Although there is about a 3.5% latency degradation due to determinism, in this example we were unable to detect any throughput degradation due to determinism at all! We estimated throughput by increasing the message rates of the external clients from the initial 1000 messages/second gradually until the system became unstable due to inability to keep up with message rates. In both deterministic and non-deterministic execution modes, the system saturated at 1235 messages/second.

We conjecture the following explanation: when the system is close to being saturated, queues are rarely empty. In these circumstances, by the time a message percolates to the top of a queue, it has been sitting on the queue long enough for many messages from other senders to arrive. Therefore the chance is low that these senders have not sent information about silences for times earlier than the time of the message ready to be processed. The frequency of curiosity probes and pessimism delay, the main sources of overhead, becomes negligible.

Although increasing the variability of the service time of *Sender* does not affect percentage overhead of determinism for a suitably smart estimator (one that actually takes number of iterations into account when estimating virtual time), this is not true if the estimator is particularly crude.

We re-ran the experiment, this time substituting a "dumb" estimator that always predicted a computation time of 600μs – the average computation time per message over all executions. In this version of the experiment, the overhead of determinism varied considerably as a function of the standard deviation. In the non-variable case (where each message always takes 10 iterations) the dumb estimator, as expected, slightly outperforms the smart estimator with non-prescient silence estimates (because such estimates think that the loop might take fewer than 10 iterations, even though actually it never

will). But in the more variable cases (each message taking a random number of iterations with mean of 10), the variation in number of iterations behaves just like operating system jitter, and does affect the overhead: it steadily increases, reaching a high of 13% for the case where the number of iterations is in the range from 1 to 19.

B. *Overhead with Realistic Jitter*

In the preceding subsection, our simulation made an unrealistic assumption about the distribution of execution times for a given amount of virtual time, assuming a normal distribution, centered symmetrically on the value of one real tick per virtual tick.

In a second study, we took measurements of an actual run of a *Sender* component in a real computer environment, subject to the effects of operating system interrupts, faults, variability of memory allocation time, etc. We imported 10000 of these execution time measurements into our simulation, representing a random number of iterations uniformly distributed from 1 to 19. This time, in the simulation, after randomly choosing a value for the number of iterations, we used the estimator of equation (2) to compute the predicted virtual time, and a random measurement from our imported set having the same iteration count, to compute the real time. Thus the data from a real execution on a single processor was used to simulate a three-processor environment with a more realistic distribution of real-time jitter.[4]

Using these simulations of realistic jitter, we sought to measure the sensitivity of performance to "inaccurate" estimators – that is, estimators that differed from the equation (2) obtained by linear regression. The results are displayed in Figure 4.

The estimator coefficient computed from the linear regression was 61.827; the best performance actually occurred at 60 microseconds per iteration, although there is considerable variation from run to run, as the wobbles in the curve indicate. The 60 microsecond per iteration estimate is also the point at which other indicators of overhead reach their minima, namely number of out-of-order messages (under 10% of all messages), and curiosity probes (average of 1.5 per message). Between 60 and 62 microseconds per iteration, actual latency is nearly flat.

It is conceivable that because of the skewed distribution of residuals, it might be marginally better to use a slightly lower value for the estimator computed by linear regression (which assumes residuals are normally distributed). Further study is needed; nevertheless these curves do suggest that best performance is obtained with an estimator coefficient close to the value obtained by linear regression.

C. *Distributed Implementation*

We ran an actual multi-engine implementation, not a simulation, of the TART protocols, using a variation of the application of Figure 1, but with constant-time services and ad-hoc estimators. The *Sender* components were on one engine, the *Merger* on a second. We compared non-deterministic execution to deterministic execution with both lazy and curiosity-based silence propagation. The results, shown in Figure 5, suggest that curiosity-based silence propagation, even without the more aggressive approaches suggested, still had less than a 20% overhead relative to non-determinism, which we conjecture is still less than the overhead of transaction processing.

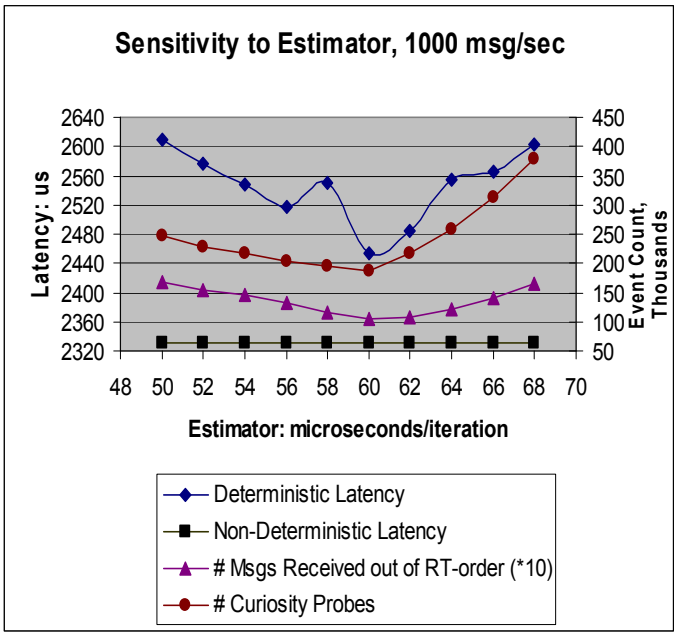

Figure 4. Sensitivity of performance to estimator, measured over a simulated run of one minute at 1000 messages/second/sender (120,000 total messages).

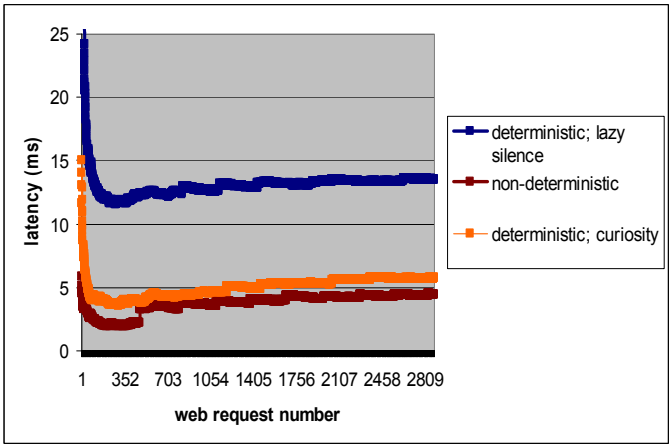

Figure 5. Performance of a real two-machine distributed implementation.

IV. Discussion, Related Work, and Future Work

We have demonstrated a technique for achieving recoverability by modifying a multi-threaded component-oriented application to yield an augmented application which (a) uses virtual timestamps to execute deterministically, and (b) uses checkpoint-replay to restore state. Because this is a transparent technique, developers can use a simple programming model in which state is stored in ordinary instance variables.

We have shown that low execution overheads can be obtained by generating timestamps with estimators for which virtual time approximates real time. We have illustrated a simple

[4] Certain sources of jitter unique to a multi-processor environment, such as cache interference, could not be captured by this study.

technique based on linear regression to calibrate these estimators. Using such estimators, the overhead relative to non-deterministic execution remains small even when components take variable amounts of processing time per message.

Alternative approaches to recovery in this environment are: replication, transactions, and checkpoint-replay mechanisms.

Replication (e.g., [4]) involves having active backups and keeping them synchronized; in effect processing is performed redundantly in multiple places. Basile et al. [14] propose combining active replication with determinism using a loose synchronization algorithm, in which one replica is a *leader* and the others *followers*. Non-deterministic mutex acquisition information from leaders is gathered and transmitted to followers. In this and in similar approaches, replicas are simultaneously active, unlike in our approach, where the active replica is the only one executing, and the passive replica only has information about a checkpoint of an earlier state. Instead of gathering non-deterministic information from the leader as it occurs and transmitting it to followers, our timestamp mechanism computes extra information ahead of time, namely the virtual time at which all relevant events will happen, forcing otherwise non-deterministic events to become deterministic.

Transaction systems provide more than mere fault-tolerance (e.g., an option to roll back a partially completed action, and concurrency control to give parallel interleaved units of work the appearance of serializability), but incur a commit-time overhead. Although transactions are invaluable for applications that require them, they are less than fully transparent and can be expensive if only recovery from failures is required. Developers must explicitly indicate transaction boundaries and commit points. State must be stored in special objects such as entity beans or an object cache [5]. Jimenez-Peris et al. [12] propose an approach based on active replication and reliable totally-ordered group multicast[13] within a transactional model. They modify the scheduler to assure that non-deterministic scheduling decisions are executed in the same way on all replicas. Totally-ordered multicast protocols are expensive relative to one-way mechanisms using timestamps and silence propagation[11]. Furthermore, our model and implementation, while not fully general (since unconstrained shared memory multithreading is disallowed), requires neither the special structuring nor the overhead of transactions.

Optimistic recovery [6] can recover an arbitrary network of components (recovery units) communicating by messages. It assumes that components are piecewise deterministic, meaning that between message receipts (or other non-deterministic events that can be modeled as message receipts) components are deterministic, but that message arrivals are not deterministic, that messages sent from multiple sources can arrive in arbitrary orders, which must be logged. The implementation of optimistic recovery therefore requires that arrival orders of messages must be logged, and that vector-timestamp dependency vectors be maintained on all messages.

We conjecture that the overheads of logging external messages and intermittently sending asynchronous soft checkpoints in our approach will be lower than the overheads of performing distributed transaction commits per processed event, and that these lower overheads are worth the price of the pessimism delays introduced by determinism. In future work, we intend to explicitly compare an industrial-strength stream processing application, involving stateful transformations, using the approach described here, versus using an approach where state is stored in a transactional object cache.

If fan-in is high, or if sending components are remote, we conjecture that curiosity-based silence propagation will have to be augmented with other approaches including aggressive and hyper-aggressive silence propagation. We have not yet explored dynamic approaches to adjusting the silence propagation techniques, or approaches to developing smart estimators for communication delay.

Because TART uses timestamps, it suggests the possibility of combining components with automatically-generated estimators with time-aware components with user-generated timestamps, in which timestamps represent arrival deadlines.

REFERENCES

[1] IBM, http://www-ibm.com/software/websphere/products/businessint/

[2] Auerbach, J., Bacon, D. F., Iercan, D. T., Kirsch, C. M., Rajan, V. T., Roeck, H., and Trummer, R.: Java takes flight: time-portable real-time programming with Exotasks. In: *LCTES '07:* Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools, 51–62. ACM Press, New York, NY, USA.

[3] Feng, T. H., Lee, E. A.: Real-Time Distributed Discrete-Event Execution with Fault Tolerance. In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08), St. Louis, MO, USA, April, 2008.

[4] Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-Tolerance in the Borealis Distributed Stream Processing System. In: Proceedings ACM SIGMOD, June 2005, Baltimore MD.

[5] Blackburn, S.M., Stanton, R.B.: The transactional object cache: A foundation for high performance persistent system construction. In: Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, August 30-September 1, 1998, Tiburon, CA, U.S.A., Ronald Morrison, Mick Jordan, and Malcolm Atkinson, http://citeseer.ist.psu.edu/blackburn98transactional.html

[6] Strom, R., and Yemini, S.: Optimistic Recovery in Distributed Systems. ACM Transactions on Computer Systems, August 1985.

[7] Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races**.** In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications, 2000.

[8] Jefferson, D.: Virtual time. ACM Transactions on Programming Languages and Systems, 7(3):404–425, July 1985.

[9] Strom, R. Recovery in the SMILE Stateful Publish-Subscribe System. In: Proceedings of the International Workshop on Distributed Event-Based Systems, May 2004.

[10] Bacon, D. F., Cheng, P., Rajan, V. T.: A real-time garbage collector with low overhead and consistent utilization. In: Proceedings of ACM Conference on Principles of Programming Languages. New Orleans, Louisiana, 285–298, 2003.

[11] Aguilera, M., Strom, R.: Efficient Atomic Broadcast Using Deterministic Merge. In: Proceedings, Principles of Distributed Computing, Portland, OR, July, 2000.

[12] Jimenez-Peris, R., Patino-Martínez, M., Arevalo, S.. Deterministic Scheduling for Transactional Multithreaded Replicas. In Proceedings of IEEE Symposium on Distributed Reliable Systems (SRDS), pp. 164-173, Nuremberg, Germany, October 2000.

[13] Hadzilacos, V., Toueg, S.. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, ed., *Distributed Systems*, pp. 97-145, Addison, Wesley, Reading MA, 1993.

[14] Basile, C., Kalbarczyk, Z., Iyer, R., Active Replication of Multithreaded Applications". IEEE Transactions on Parallel and Distributed Systems (TPDS). Vol. 17, No. 5. pp. 448-465. May 2006.