A Practical Approach for 'Zero' Downtime in an Operational Information System

Ada Gavrilovska, Karsten Schwan, Van Oleson
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30329
{ada, schwan, van}@cc.gatech.edu

Abstract

An Operational Information System (OIS) supports a real-time view of an organization's information critical to its logistical business operations. A central component of an OIS is an engine that integrates data events captured from distributed, remote sources in order to derive meaningful real-time views of current operations. This Event Derivation Engine (EDE) continuously updates these views and also publishes them to a potentially large number of remote subscribers. The paper first describes a sample OIS and EDE in the context of an airline's operations. It then defines the performance and availability requirements to be met by this system, specifically focusing on the EDE component. One particular requirement for the EDE is that subscribers to its output events should not experience downtime due to EDE failures, crashes or increased processing loads. Toward this end, we develop and evaluate a practical technique for masking failures and for hiding the costs of recovery from EDE subscribers. This technique utilizes redundant EDEs that coordinate view replicas with a relaxed synchronous fault tolerance protocol. A combination of pre- and post-buffering of replicas is used to attain a solution that offers low response times (i.e., 'zero' downtime) while also preventing system failures in the presence of deterministic faults like 'ill-formed' messages. Parallelism realized via a cluster machine and application-specific techniques for reducing synchronization across replicas are used to scale a 'zero' downtime EDE to support the large number of subscribers it must service.

1 Introduction

Increasing a server system's performance, reliability, and fault-tolerance by means of replication is common practice [2, 5, 6, 7, 9, 10, 22]. Performance improvements are attained by use of parallelism and concurrency [19]. By using additional techniques for fault detection, masking and recovery, replication can deal with hardware failures, and with software failures caused by non-determinism or by certain behavior at isolated node(s). However, deterministic software errors that cause all replicas to fail concurrently cannot be prevented, and their occurrence can result in catastrophic system failures [7, 18]. For instance, in the application considered in this paper, an 'ill-formed' data event processed by the replicated server engine will crash every one of the server's replicas.

This paper describes a scalable server architecture which both meets user-level performance requirements, particularly focusing on reducing delays experienced by end users, and also hides failures (actual or performance), even in the presence of 'ill-formed' input messages. The architecture is evaluated in an emerging application domain for high performance, high availability computing, termed Operational Information Systems (OIS).

Operational Information Systems. An OIS is a large-scale, distributed system that provides continuous support for a company's or organization's daily operations. One example of such a system is the OIS run by Delta Air Lines, which provides the company with up-to-date information about all of its operations, including flights, crews, passenger, and baggage [16].

Delta's OIS is driven by the real-time acquisition of data from many disparate, remote sources, such as FAA data feeds, airports serviced by Delta, etc. One of the principal tasks of the OIS is to integrate such data events, by applying to them relevant 'business logic'. Another important task is to make the 'business events' produced by such logic available to a potentially large number of remote subscribers, such as airport flight displays and gate agents. The model of real-time event subscription, derivation, and publication it thus implements distinguishes an OIS from Data Warehouses and Decision Support Systems. This model is also the basis on which the OIS builds its interactions with other system components, like those participating in E-commerce interactions as with passenger reservations.

Event derivation engines (EDEs) and their performance requirements. An event derivation engine (EDE) is the central processing component of an OIS. Its role is to accept incoming data streams, correlate them, and apply business logic. The EDE therefore, must capture, process, and provide information continuously, with bounded delays and with little or no downtime experienced by both its event sources and subscribers. It must also service clients' explicit requests. Yet, the EDE is susceptible to hardware upgrades and failures, failures and incompatibilities in the software it runs, shared library conflicts, failures due to the receipt of 'ill-formed' data messages, and others.

Traditionally, large enterprise computing has constructed EDEs with clusters of mainframes that run proprietary information systems software. For example, Delta uses a cluster of IBM S/390s that run the specialized TPF (Transaction Processing Facility) operating system [21]. TPF continues to support applications that automate the majority of the airline's operational services. Given Delta's experiences with TPF over the last 30 years, any new EDE introduced to Delta must match its performance while also offering new functionality. Performance requirements include (1) 99.99% average availability; (2) performance of up to 5400 transactions per second, with (3) 1-3 seconds response time attained 95% of the time; (4) at a cost of less than 2 cents per transaction. Additional responsiveness requirements include the ability to service requests from clients within time delays not exceeding 1-2 minutes. Sample requests include downloads of new initial states for end users (e.g., airport displays) after failures and updates to gate agents' operating systems.

Responsiveness and availability via replication. This paper presents a two-tiered software/hardware architecture for an EDE. The architecture aims to attain high levels of availability and reliability, while also offering a high degree of responsiveness (i.e., bounded response times) to explicit client requests. The effects of failures are mitigated by use of replicated software and hardware components, as well as replicated data. High availability is attained by runtime detection and elimination of certain deterministic software errors that cause all replicas to fail, thereby reducing the downtimes experienced in systems where all replicas have to be rebooted (up to 45 minutes in the current system) to 1-2 minutes. Redundancy attained through replication is also the way bounded request response times are attained, effectively achieving 'zero' perceived downtimes for typical end users. Finally, by using standard cluster computing hardware and COTS operating systems, the architecture is easily expanded to provide new functionality, such as support for passenger paging upon flight arrivals, gate changes, etc.

A two-tiered replication architecture. The first tier of the two-tiered replicated EDE (REDE) is a replicated engine that applies business logic to incoming events, and publishes update events to the OIS' clients. The main purpose of this tier is to effectively detect and mask from clients an EDE node's failure and recovery. The first tier of the REDE is enhanced by a second tier of 'mirror sites' to which data is mirrored from the REDE. Its purpose is (1) to handle the requests for new initial states (explained in more detail in Section 3) made by clients and REDE nodes during recovery, so as to improve the REDE's responsiveness to such requests, and (2) to also reduce the negative performance implications of recovery actions for REDE nodes. An additional technique used to improve the performance of the second-tier set of nodes (i.e., mirror nodes) is to adaptively mirror the events incoming into the REDE onto those nodes. This adaptive mirroring technique is explored in more detail in [13] and will not be further described in this paper. Both tiers use hardware that is arranged as a cluster of machines, with cluster nodes tightly coupled via a switch, but independently powered, and using replicated software and data, so that the failure of one node does not cause failures of other cluster nodes. Finally, event replication and mirroring within the REDE may be costly, especially if REDE nodes are required to operate synchronously. We reduce the potentially high overheads of strong synchronization across REDE nodes by relaxing the consistency requirements of nodes' internal states, using domain knowledge and exploiting the streaming nature of the event data processed and generated by the REDE.

Replication with pre- and post-buffers. The key to our implementation of reliable and responsive REDE nodes is the use of **pre-** or **post-buffers** at each node, a concept that is enabled by the streaming nature of OIS applications. At any one time, one of the REDE nodes acts as a primary node, while the others maintain either pre-buffers of received, but still unprocessed data events, or post-buffers of already successfully processed data. The role of post-buffers is to implement fast replay of already processed data in the event of a primary's hardware failure or a non-deterministic software failure. Pre-buffers guard against deterministic software failures that cause all of the REDE nodes' executions of business logic to fail. A pre-buffer permits, for instance, the extraction of 'ill-formed' messages from the incoming event stream, if such messages cause node failures.

Consistency across REDE nodes is weakened in the time-domain only, essentially permitting a node's pre- or post-buffer to be 'behind' or 'ahead' of other nodes' buffers, as long as sufficient state exists to recover from failures. More importantly, in the event of a primary's failure, failover to a pre- or post-buffer node hides the delay experienced during recovery from the REDE's clients. For limited levels of replication, this effectively results in 'zero' perceived downtime by clients. To attain this goal, however, we make the assumption that clients participate in this process, by detecting and eliminating duplicate update events sent to them based on unique event timestamps.

2 Operational Information Systems

Figure 1 shows how operational information is captured in a wide area setting. This information is comprised of update events describing airplanes' locations and crew availability, for instance, and its continuous capture results in streams of update events received by the OIS server (i.e., the EDE). The data events being captured from, processed at, and distributed to large numbers of company sites concern the current state of company assets (e.g., airplanes or crews) and that of their customers' needs (e.g., passenger or luggage volumes and destinations). The ability to track and manipulate such information in a timely fashion is critical to the company's operations and profits. In the case of Delta Air Lines, events originate at system capture points, such as gate readers or radar data collected by the FAA. Legacy systems like the aforementioned TPF are an additional source of capture points, where software agents are strategically injected to capture TPF transactions in real-time. The outputs generated by the center's EDE server are used by a myriad of clients, ranging from simple airport flight displays to complex web-based reservation systems.

The EDE must also maintain for its many clients meaningful 'initial states' that they can acquire to recover from failures. A typical example is an airport flight display that upon recovery from a power failure, cannot interpret new flight events published by the EDE until it has been sent an initial state that is then incrementally updated by these events. The role of the EDE, then, is twofold: (1) it serves as an event processing engine, and (2) it maintains initial states on behalf of clients that request such states as needed. Currently, at Delta, the EDE processes up to 12 million source messages per day, derives at least that many business events to an initial deployment of $10,000$ remote subscribers, whose number is expected to grow to over $100,000$ in the foreseeable future. An explosion of initial state queries places high levels of additional load on the system. These initial states can be on the order of several MB; for flight information display applications these states are 5 MB of XML-represented data.

The goals of the EDE are (1) to minimize the time between the receipt of an event and the publication of the resulting business event and (2) to provide a high availability model to its clients that does not add additional latency beyond a client's perception of a real-time event. To meet (1) and (2), a challenge beyond dealing with large, bursty client requests is to address failures experienced by the EDE. The

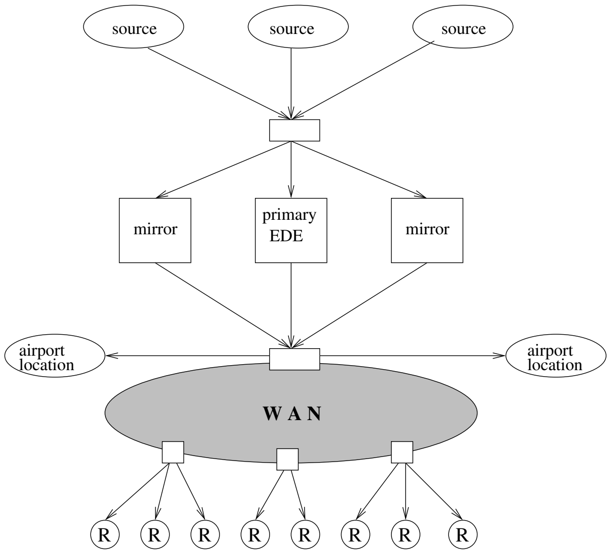

Figure 1. Original Delta architecture.

effects of such failures can be severe due to the large recovery times experienced by the EDE, the latter caused by the size of the state maintained in the EDE. However, even if the state is re-loaded quickly, total recovery time of the EDE much exceeds this re-load time. First, upon recovery, the EDE must deal with a potentially large backlog of incoming events and the corresponding outgoing events. Second, there may be outstanding initial state requests. In fact, measurements performed on Delta's EDE indicate recovery times that often exceed 45 minutes. This is not an acceptable situation, particularly since failures in this system are not infrequent. Instead, recovery time should be masked and clients should not perceive any system downtime. This is the problem addressed by the replicated server architecture described in the next section.

3 Replication Architecture Design and Implementation

Basic architecture. The prototype design of our two-tiered replication architecture is presented in Figure 2. Event streams originate from a number of data sources, and are replicated to a set of first-tier nodes which constitute the REDE. Each REDE node maintains its own application state and executes 'business logic' rules on incoming events. A relaxed synchrony replication protocol is used to maintain clients' perceptions of shared state consistent across REDEs.

Upon receipt of an event, the primary REDE node timestamps and mirrors the event to a set of second-tier mirror nodes, based upon a number of application-specific rules that determine the mirroring frequency and granularity, and using a degree of event filtering that complies with the consistency requirements of the application [13]. While this paper focuses on the attainment of high availability and 'zero' downtime via EDE replication and event mirroring, a second purpose of event mirroring is to distribute the loads generated by initial state requests issued upon clients' failure, thus decreasing the average update delay experienced by clients, even under unexpected bursts of request load (see the experimental results presented in Section 4). Thus, the two-tiered design results in an architecture where, under varying operating conditions, update delays can be maintained within bounds that result in what clients perceive as 'zero' downtime. Our implementation, both at the REDE replicas and the mirror sites, separates the application-specific event processing functionality (expressed through a set of 'business logic' rules) from the execution of the control mechanisms necessary for consistent replication and fault detection and recovery [13].

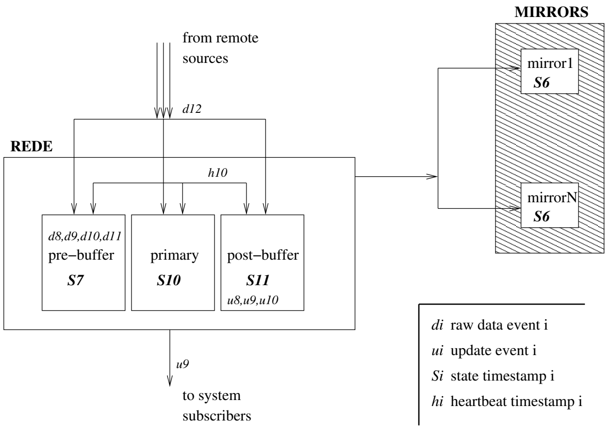

Figure 2. Two-tiered Replication Architecture. REDE nodes' states are after heartbeat event *h7* has been exchanged. Primary node already published update event *u9* and has just processed *d10* and updated its state to *S10*. Post-buffer node has advanced ahead to data event *d11*, but maintains in buffer updates following timestamp *h7*. Pre-buffer node cannot advance beyond *S7*, and holds in pre-buffer subsequent data events.

Replicated Event Derivation Engine. The REDE (see Figure 2) consists of three different derivation engines, weakly consistent in the time domain [1], of which only one at a time acts as a primary event publisher. Each of the replicas receives the same event stream, locally performs the necessary timestamping of the events, and passes the events to the local business logic. At each node, the most current event timestamp reflects the node's current view of the application's state. Application semantics, required for the application-specific mirroring, are captured though a set of *semantic rules* that describe the actions associated with different event types, or even their content. The mirror sites maintain their view of application state based on the timestamps of events mirrored onto them.

The pre-buffer node postpones the 'business logic' processing of newly arrived data, at least until it has been successfully processed by the primary node. The post-buffer node executes the business logic at a rate that maintains a predefined number of most recently processed events in its buffer. Newly derived business events are published solely by the primary REDE node. The primary node also controls the way in which mirroring is performed.

Relaxed Synchrony Protocol. Synchronization is required to guarantee that EDE subscribers do not experience diminished consistency. This is implemented through the exchange of heartbeat/control events. These events contain the timestamp tms_p of the last successfully processed data event by the primary node. The post-buffer node always maintains in its buffer all processed events with timestamps greater than the most recent tms_p. The pre-buffer node can remove from its buffer, and process, all events with timestamps preceding tms_p. Timeouts are associated with these control events; their expiration implies failure occurrence.

Detection. Within the duration of a single timeout, the pre- and post-buffer nodes can detect a primary site failure (timing or performance [15, 17]). The failover mechanism is as follows. First, the node maintaining the post-buffer sends a heartbeat control event. Second, it resubmits all events in its post-buffer. Third, it assumes the primary site's functions. Existence of 'ill-formed' messages in the data stream will also cause the post-buffer node to fail. In these cases, the pre-buffer node executes an error detection and removal mechanism, extracts these messages, and then becomes the primary site. Our prototype implementation trivializes this process, by explicitly inserting marked 'ill-formed' messages in the event stream, whose detection is done by simple comparison. In the event of hardware failure of the primary node, replay of the post-buffer node is required. This implies that the EDE subscribers need to be capable of detecting and removing duplicate updates. In the event of 'ill-formed' messages, after they are extracted from the pre-buffer, the remaining messages are first processed and then the derived updates are published to clients. This results in increased update delays, but there are additional factors contributing to such delays. Such factors include timeout values used with heartbeat messages, queuing delays experienced due to buffering, the average processing time of events, and the mechanism used to detect and remove erroneous messages. Independent of these, the REDE continues to stream newly derived data events to its subscribers, thereby hiding the failure of the primary node.

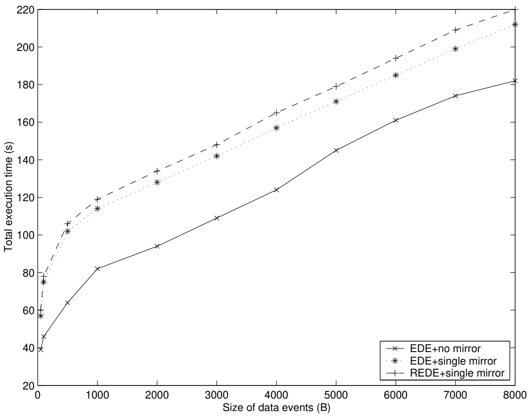

Figure 3. Overhead of implementation

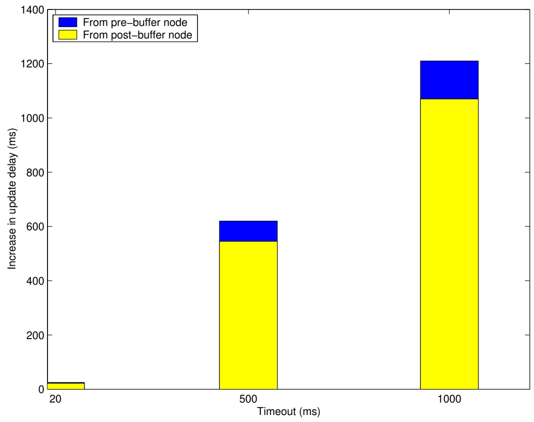

Figure 4. Average update latency increase as a result of failure

Recovery. Upon failure of the REDE's primary node, the failover mechanism activates the pre- or post-buffer node, and the primary node's recovery process is masked from the system's clients. During the recovery of the REDE's failed nodes, we avoid retrieving the initial state from the currently active primary node, since such additional load would result in increased update latencies. Instead, we rely either on the pre-buffer, or on the mirror sites if only one REDE node is currently active. The only interaction between the recovering node and the current primary node is with respect to global timestamp synchronization. The recovering node starts buffering incoming messages, which will later be applied to the received state.

A more detailed explanation of the implementation of our prototype system and its synchronization mechanisms appears in [14].

4 Evaluation

Experiments are performed with the replicated server (REDE and mirror sites) running on up to eight nodes of a cluster of 300MHz Pentium III dual-processor servers running Solaris 5.5.1. The ECho event communication infrastructure [11] is used to efficiently move data events across nodes. The 'flight positions' data stream used in these experiments originates from a demo replay of original FAA streams, and it contains flight position entries for 50 different flights. The evaluation metric is the total execution time of the simulation. To simulate client requests that add load to the server's sites, we use httperf version 0.8, a standard tool for generating HTTP traffic and measuring server performance. Httperf clients run on 550MHz Pentium III Xeon nodes connected to the server cluster via 100Mbps Ethernet.

Event replication overheads are acceptable. A set of microbenchmarks measures the basic replication overheads in the two-tiered REDE architecture, as a function of varying data sizes. We compare the total processing time for a fixed event sequence in the replicated implementation, with an implementation that contains a single EDE without additional mirrors. No failures nor additional load due to incoming client requests are assumed for this experiment. The heartbeat rate is set at one per 20 processed messages, and the timeout value at 1sec. The mirroring overhead alone is captured through the dotted line in Figure 3, which represents the case when a one-node EDE is used in combination with a single mirror site. The increase due to heartbeat traffic when a replicated EDE is used, is an additional 5% of the total execution time, which results in a total increase in the execution time of approximately 20%.

Pre- and post-buffers effectively mask the down-time experienced by clients. Next, we evaluate the REDE's behavior in the event of failures, demonstrating the increase in update delay as a result of our failover mechanism and the potential duplication of published events. We simulate failures by instrumenting the original event stream with events that indicate a primary site failure, or that are explicitly marked as 'ill-formed' messages. The measurements in Figure 4 are gathered with a single mirror-replicated REDE setup, and represent the increase in update latency as a result of a single failure. The delay encountered is computed by measuring the time difference of when the event would have been processed, had it not indicated a failure, and when the following event is sent. We use a data size of 500B, for which the average processing time is 8.5ms. Measurements are taken for three different timeout values: 20, 500, and 1000ms, and the corresponding heartbeat rates are every 2, 25, and 50 events, respectively. The tests were repeated for 'hardware' failures, for which the post-buffer replay is sufficient (lighter bars in Figure 4), and for 'ill-formed' message failures, for which the pre-buffer replica becomes the primary node (darker bars).

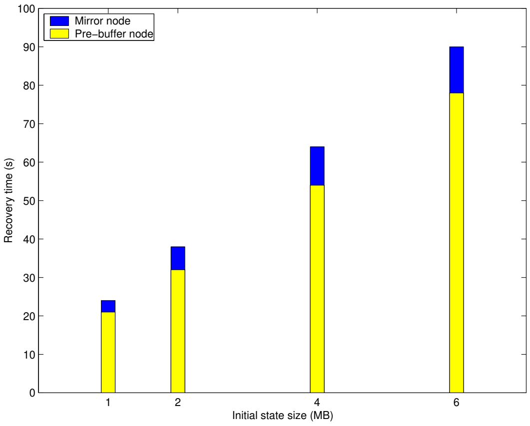

Figure 5. Recovery times when pre-buffer or mirror node is used to provide initial states

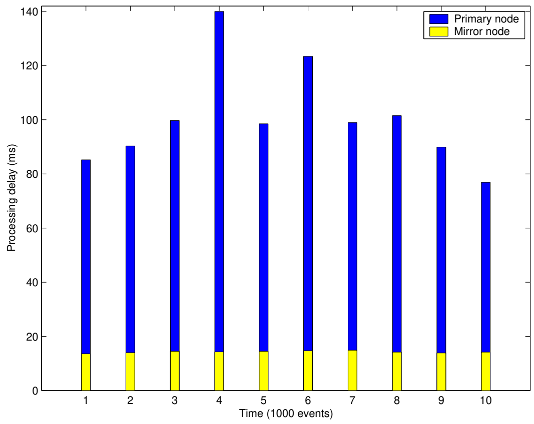

Figure 6. Comparison of processing delays when a mirror node provides the initial states vs. the primary REDE node.

Experimental results indicate that on average, the increase in the event update latency experienced by the REDE is less than 9% larger than the timeout value, if the post-buffer node can assume primary node status. If the post-buffer node also fails and if the 'ill-formed' message must be extracted from the pre-buffer, then the event update latency increases by 22%. The measured increases in the update delays show that with the REDE design, perceived downtime is reduced to values that fall within the response time requirements of our application. In effect, therefore, the failure of a node within the REDE is masked from clients, resulting in what they will perceive as 'zero' downtime.

The recovery mechanism for a REDE node is evaluated for two situations: (1) when a pre-buffer node is available, and (2) when a mirror site services the state request. In both cases, a heartbeat rate of 1 control message for each 20 processed events is used, with an event size of 500B. The results presented in Figure 5 show that the failed node recovers within 1.5 minutes in both situations. We further observe that when a mirror node needs to be polled for initial state, then the recovery process is on average only 10-15% longer, compared to the case of a pre-buffer being available, mostly due to the fact that mirror nodes typically lag behind the primary node with respect to their pre-buffers.

Mirroring is critical for efficient client request processing. The next experiment demonstrates the performance implications of using the primary node to service client requests (i.e., requests for initial states), in place of using a mirror node for this task. We measure and compare the processing delays experienced by data events if a recovering node must poll a mirror site for the initial states vs. when it polls the primary node. The results in Figure 6 show the processing delays for every 1000th event in a stream of $10,000$ events for both cases. We observe that the processing times can increase by a factor greater than 10. Such an order of magnitude increase will probably result in perceivable downtime by the clients. This experiment supports our motivation not to rely on the primary node for initial state during recovery, but to poll the pre-buffer or the mirror nodes instead.

Mirror node replication addresses variations in client request loads. We further support the REDE's two-tiered design by analyzing the impact of increases in clients' request load on the update delays experienced by 'regular' events. The darker bars in Figure 7 show the increases in average event update delays, which result from an increased rate of incoming client requests. This increase can be avoided by moving request servicing functionality onto mirror nodes. Second, with our adaptive mirroring support, we can modify the mirroring function, reduce its frequency, allow event coalescence, or some filtering based on the events type or even content. Such modifications enable the system to better adapt to unusual operating conditions. Reducing the overall consistency between the mirrors, and the REDE nodes, allows the REDE primary node to continue publishing state updates in a timely manner, even under increased request loads issued to the mirror sites. The lighter bars in Figure 7 correspond to delays when such adaptation of the mirroring function takes place. We observe that the latency increases are reduced by more then 35%. As a result of this, 'regular' clients are more likely to experience delays that fall within the regular service levels, and perceive 'zero' downtime, even during system overload under unusual operating conditions.

Discussion of results. The experimental analyses of our approach indicate that the two-tiered REDE architecture masks node failures from clients with relatively low over-

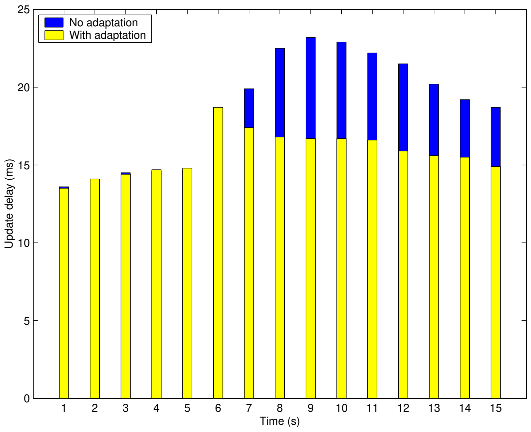

Figure 7. Controlling increases in update latencies through adaptations in the mirroring process. A burst of client requests occurs during the time interval 5-10 sec.

heads, at the cost of increased update latencies on the order of 1 second. To clients like gate displays or gate agents, such overheads will appear as 'zero' system downtime. Furthermore, in the case of client failure, the REDE supports client recovery within a few minutes after the node is brought back up. We further justify the two-tiered REDE design by demonstrating drastic increases in the update delays experienced by clients if the primary node must be used for initial state recovery, instead of a mirror site. Finally, we show that with dynamic adaptation of the mirroring function, the mirrors still provide the requested initial states, however the system's response time is maintained at more constant levels. Hence, the second tier contributes towards our goal of perceived zero downtime and masked failures.

5 Related Work

Using replication as a technique to increase system reliability and fault-tolerance has been widely accepted among both researchers and in industry (e.g., [2, 5, 6, 7, 9, 10, 22]). As with our solution, most approaches rely on (1) a checkpointing mechanism through which they track each other's progress, and (2) a log of events that must be either undone or redone when a failure occurs [3, 12]. The exchange of heartbeat traffic, either explicit or embedded in other messages, and timeouts associated with it, is a standard mechanism for detecting that a failure has occurred. Limited buffering of uncheckpointed state, state updates, and/or messages is used to maintain the system in a consistent state in the presence of failures [4, 5, 6, 7] or erroneous assumptions in optimistic synchronization solutions [20]. Well-known systems like Horus [22] and Transis [10] have already demonstrated the utility of replicating the processing performed on input events, including the use of group synchronization to maintain certain levels of consistency across such active replicas. We build on the approach demonstrated by these systems, by replicating both data and processing, but we differ in our use of application-level knowledge to further relax the requirements of synchrony across replicated processes. A further difference is our principal goal to achieve what appears to be zero downtime to the EDE event subscribers. An outcome of our work is the two-tiered nature of the REDE described in this paper.

There are many techniques for reducing the levels of consistency to be maintained across replicas, including the system-level provision of mechanisms through which this can be dynamically specified and/or controlled by the application [9, 15]. We provide an API through which the programmer can express consistency requirements in terms of the types and values of the data exchanged in the application. Furthermore, our replication mechanism is optimized by using a relaxed synchrony approach, which is enabled by the streaming nature of our application domain. Other systems propose similar optimizations ([9, 15, 23]), but the generalized nature of their solutions does not allow them to fully benefit from this fact.

BASE, an extension of the Byzantine fault-tolerance system, BFT [7, 18], is a technique which handles deterministic software failures by replicating the affected service. They rely on non-deterministic behavior across multiple, potentially different system implementations, similarly to N-versioning [8], to increase the probability that not all implementations will fail. They strictly maintain pre-buffers, and they do not permit individual replicas to advance beyond the primary. In comparison, our approach combines the use of pre- and post- buffers, which allows us to prevent system-wide failures caused by deterministic software faults, while still benefiting from lightweight, optimistic synchronization under other circumstances.

Finally, the use of micro-protocols [15] for constructing highly configurable fault-tolerant distributed services is well-known. Our research would benefit from such work to construct modular and perhaps, dynamically composable higher-level protocols that could more flexibly leverage application semantics when dynamically adapting event mirroring and the levels of consistency and synchrony maintained across replicated EDEs.

6 Conclusion and Future Work

This paper deals with reliable Operational Information Systems, where reliability is expressed in terms of service degradation experienced by clients as a result of failures or overloaded operating conditions. Specifically, the OIS' clients rely on its capability to continually publish data and in addition, to process their explicit requests, both with bounded delays. To meet these service requirements, we develop and evaluate a two-tiered replication architecture. The first tier is a replicated event derivation engine, whose nodes rely on a relaxed synchronization protocol and on message pre- and post-buffering, to provide clients with a consistent view of system state in the event of a failure. Pre- and post-buffers provide a solution that (1) hides a node's failure and its recovery process, with minimal increases in the processing delays experienced by clients, and (2) prevents system-wide crashes in the face of deterministic software faults such as 'ill-formed' messages. The purpose of the second tier is (1) to maintain low update delays when also having to service explicit requests from clients for information, and (2) to deal with changes in operating conditions, such as sudden bursts of requests. This is achieved by dynamically varying consistency among nodes, thereby offering to clients a 'zero' downtime view of the system.

Our ongoing research addresses the reliable diffusion of events with bounded delays in wide-area systems. Specific work includes the consideration of application-level multicast for event distribution, and the efficient encoding of events containing XML data with Just In Time XML adapters, thereby reducing overall event latency. Finally, we are extending our focus on reliability to collectively evaluate total system reliability, including the applications, middleware, systems, and network infrastructure. Specifically, by applying content routing techniques to the OIS architecture, control of the flow of events can be based on application semantics. This will allow for adaptable overlay networks that can restrict the flow of information when network reliability is reduced.

References

[1] M. Ahamad and R. Kordale. Scalable Consistency Protocols for Distributed Services. *IEEE Transaction on Parallel and Distributed Systems*, 1999.

[2] Akamai Technologies. http://www.akamai.com.

[3] L. Alvisi, K. Bhatia, and K. Marzullo. Causality Tracking in Causal Message-Logging Protocols. *Distributed Computing*, 15(1), 2002.

[4] G. Ballintijn, M. van Steen, and A. S. Tanenbaum. Simple Crash Recovery in a Wide-area Location Service. In *Proc. 12th International Conference on Parallel and Distributed Computing Systems*, pages 87–93, Fort Lauderdale, FL, Aug. 1999.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), Aug. 1991.

[6] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budio, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[7] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proc. of the 4th OSDI*, San Diego, CA, Oct. 2000.

[8] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Spproach to Reliability of Software Operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, Toulouse, France, 1978.

[9] F. Cristian, B. Dancey, and J. Dehn. High Availability in the Advanced Automation System. In *Proc. of 20th Int. Symp. on Fault-Tolerant Computing*, Newcastle upon Tyne, UK, 1990.

[10] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr. 1996.

[11] G. Eisenhauer, F. Bustamante, and K. Schwan. Event Services for High Performance Computing. In *Proc. of Ninth High Performance Distributed Computing (HPDC-9)*, Pittsburg, PA, Aug. 2000.

[12] M. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, pages 526–531, May 1992.

[13] A. Gavrilovska, K. Schwan, and V. Oleson. Adaptible Mirroring in Cluster Servers. In *Proc. of 8th High Performance Distributed Computing (HPDC)*, San Francisco, CA, Aug. 2001.

[14] A. Gavrilovska, K. Schwan, and V. Oleson. A Practical Approach for Zero Downtime in an Operational Information System. Technical Report GIT-CC-02-14, College of Computing, Georgia Tech, Mar. 2002.

[15] M. A. Hiltunen, V. Immanuel, and R. D. Schlichting. Supporting Customized Failure Models for Distributed Services. *Distributed System Engineering*, 6:103–111, Dec. 1999.

[16] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[17] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proc. of 22nd Symposium of Fault-Tolerant Computing*, pages 386–395, 1992.

[18] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstractions to Improve Fault Tolerance. In *Proc. of 18th ACM SOSP*, Banff, Canada, Oct. 2001.

[19] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability, and Performance in Porcupine: A Highly Scalable Cluster-based Mail Service. In *Proc of 17th ACM SOSP, OS Review*, Kiawah Island Resort, SC, Dec. 1999.

[20] J. Sussman, I. Keidar, and K. Marzullo. Optimistic Virtual Synchorny. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Nurnberg, Germany, Oct. 2000.

[21] IBM Transaction Processing Facility. IBM Corporation. http://www.s390.ibm.com/products/tpf.

[22] R. van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[23] H. Yu and A. Vadhat. The Costs and Limits of Availability for Replicated Services. In *Proc of 18th ACM SOSP*, Banff, Canada, Oct. 2001.