A Practical Approach for Zero Downtime in an Operational Information System

Ada Gavrilovska, Karsten Schwan, Van Oleson
GIT-CC-02-14
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. This paper 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 and crashes or increased processing loads. This paper describes 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. Combination of pre- and post-buffering replicas is used to attain an optimal solution, which still prevents system-wide failure in the face of deterministic faults, such as 'ill-formed' messages. By minimizing the amounts of synchronization used across replicas, the resulting 'zero downtime' EDE can be scaled to support the large number of subscribers it must service.

1 Introduction

Increasing system's reliability and fault-tolerance by means of replication is a common technique used by researchers and in industry [2, 6, 7, 8, 10, 11, 25]. Performance improvements are attained by use of parallelism and concurrency [22]. 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 [8, 21]. 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 [19].

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 resulting 'business events' being derived available to a potentially large number of remote subscribers, such as airport flight displays. 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 a basis on which the OIS builds its interactions with other system components, like those participating in E-commerce interactions like passenger reservations. Delta's OIS combines three different sets of functionality:

- *Continuous data capture* – as with high performance applications that capture data from remote instruments like radars or satellites, or from physical instruments like the particle accelerators described in [5], an OIS must continuously capture all data relevant to an organization's operations. For airline operations, such information includes crew dispositions, passengers, airplanes and their current locations determined from radar data available from the FAA, etc.
- *Continuous state updates* – as with digital library servers that must both process newly received data (e.g., NASA's satellite data) and then distribute it to clients that request it [17], an OIS must both continuously process all operational information that is being captured (i.e., by applying relevant 'business logic' to it) and then store/distribute the resulting updates of operational state

to all interested parties. In the case of Delta Air Lines, this includes low end devices like airport flight displays, the PCs used by gate agents, and even large-scale databases in which operational state changes are recorded for logging purposes.

- *Rapid response to client requests* – an OIS not only captures data and updates/distributes operational state, but it must also respond to explicit client requests. In the case of Delta's OIS, clients request new initial states when airport or gate displays are brought back online after failures, or when agents initialize their displays (e.g., agent login), and certain clients may generate additional state updates, such as changes in flights, crews, or passengers. To provide timely service to passengers and optimize operational costs, it is important to respond to such requests with predictably bounded delays.

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 relevant 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; as well as service explicit clients' 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 ([14]) that run the specialized TPF (Transaction Processing Facility) operating system [24]. 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 event derivation engine (EDE) introduced to Delta must match its performance while also offering new functionality. Performance requirements of an EDE for operational information systems like Delta's 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. 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-tier replication architecture. The two-tier replicated EDE (REDE) described and evaluated in this paper masks node failures and achieves 'zero downtime', as perceived by the REDE's clients. Its first tier 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 EDE node's failure and recovery from clients. 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 2) 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. 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 avoid 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 either **pre-** or **post-buffers** at each node, a concept that is enabled by the streaming nature of OIS applications. The REDE operates as follows. 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 in the event of non-deterministic software failures on REDE nodes. Pre-buffers guard against deterministic software failures that cause all of 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. The pre- or post-buffers maintained on different REDE nodes need not be entirely consistent. EDE subscribers do not perceive such diminished consistency, because the relaxed synchrony approach implemented in our system weakens consistency only in the time domain, essentially permitting a REDE 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.

Adaptive event mirroring for improved performance and responsiveness. The objective of the second tier is to attain low average delays for REDE event derivation and publication, even when the REDE experiences increased loads in terms of the number of initial state requests received from recovering clients or from other REDE nodes. Another way of stating this objective is that the second tier reduces or eliminates the REDE's timing failures [16, 20]. In Delta's OIS, for example, client recovery can result in substantial and sudden request loads, as exemplified by power failures at airports that result in requests for new initial states when power is restored, for all affected gate agent machines and airport displays. An additional technique used to improve the performance of the second-tier set of nodes – mirror nodes – is to adaptively mirror the events incoming into the REDE onto those nodes. With this technique, we dynamically modify the granularity at which mirroring is performed, thereby also modifying the level of data consistency maintained across different mirror nodes. Such adaptations are made possible by exploiting the streaming nature of the incoming data events and by using certain application-level semantics, such as the knowledge that a position update for a flight will be followed by future position updates, so that it need not be mirrored when loads are high. The benefits of adaptive mirroring are explored in more detail in [15].

Results. The prototype implementation of the two-tier REDE consists of three types of nodes: primary, pre- and post-buffer nodes. Experimental findings with this prototype indicate that the architecture can efficiently 'hide' the failure and the recovery times of REDE nodes, thereby maintaining clients' perception of zero downtime while also offering to clients acceptable delays for event delivery. Specifically, the delays in event delivery experienced by clients during REDE node failures can be maintained within 10 to 20% of the time interval within which a heartbeat message is expected from the server. By selecting this parameter with consideration of the clients' requirements we can attain our goal of perceived zero downtime.

The REDE's combined use of pre- and post-buffers allows it to recover from deterministic software failures due to 'ill-formed' messages. At the same time, high performance of the REDE is attained by use of a lightweight protocol for optimistic replication that takes advantage of the streaming nature of the events continuously processed by the REDE. This protocol outperforms BASE [21], a replication technique that also handles this type of failures, by achieving more than 40% smaller update delays.

The REDE's two-tier design improves performance during recovery, where mirror, rather than primary nodes respond to explicit client requests, such as requests for new initial state after client failure. The design also results in improved performance in terms of clients' perception of downtime when REDE nodes fail.

Remainder of paper. The remainder of this paper first describes our target application, its processing model, and our approach to increasing its availability. We describe the proposed two-tiered replication architecture based on relaxed synchrony in Section 3, and include some implementation detail. Experimental analyses of our implementation are given in Section 4. Related research appears in Section 5. We conclude with a summary of our contributions and future research possibilities.

2 Operational Information Systems

Figure 1 shows how operational information comprised of update events describing airplanes' locations and crew availability, for instance, is captured by a wide area collection infrastructure. This results in a continuous stream 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 including gate readers or radar data collected by the FAA. Legacy systems like the aforementioned

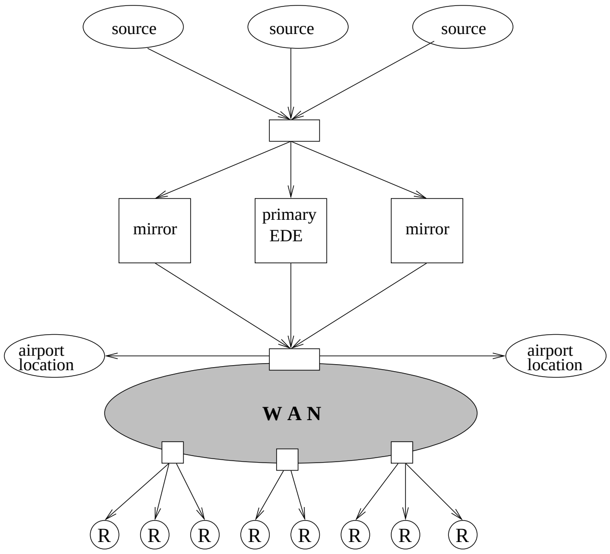

Figure 1: Original Delta architecture. Remote sources stream data to the central EDE, and back-up mirror sites. Processed updates are distributed to a myriad of clients, such as airport displays and airlines agents.

TPF are also a source of capture points, where software agents are strategically injected to capture TPF transactions in real-time. These resulting transaction histories along with other events from capture points are distributed via company-internal or leased network links, and processed by servers located at the company's processing center in Atlanta. A representative processing action performed by the EDE is one that determines from multiple events received from gate readers that all passengers of a flight have boarded. 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.

2.1 Event Derivation Engine

We have already explained that the EDE must support the acquisition, derivation, and publication of events with low latencies and with bounded delays. Furthermore, 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 a power failure and its recovery from it, 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 effects of such failures can be severe due to the large recovery times experienced by the EDE. Recovery overheads are due to the size of the state maintained in the EDE. As detailed in [19], in a large-scale EDE, this Terabyte-size state is the 'working set' from which suitable initial client states and output events are derived. In this paper and in practice, the size of this working set is reduced by focusing on precisely the data needed for near-term operational decisions and actions. Specifically, event throughput and latencies are improved by defining a 'derivation subset', named the Derivation Working Set (DWS), which is a subset of the operational working set (OWS). The lifetime, or window, of the information contained in the DWS is based on business operations for a specific business domain.

There are two deleterious effects of failures at the EDE. Namely, even if its DWS 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 they enter the system at a single point, termed routing engine (RE). This RE propagates events to some arbitrary number of nodes. In order to reduce its likelihood of failure and enable its implementation as hardware or within the network

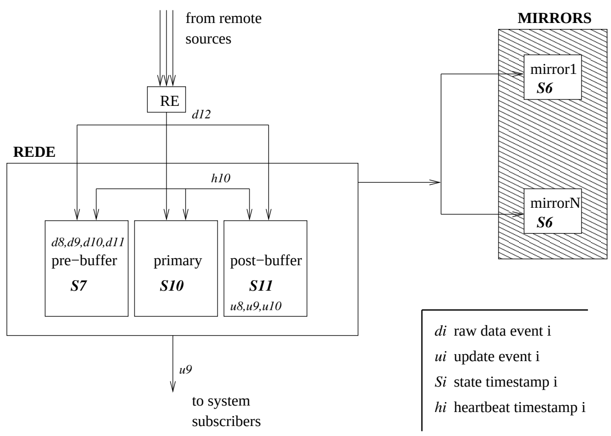

Figure 2: Two-tiered Replication Architecture. REDE nodes' states are after heartbeat event *h7* has been exchanged. Primary node already published update event *u9* and has just processed *d10* and updated its state to *S10*. Post-buffer node has advanced ahead to data event *d11*, but maintains in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

infrastructure, the RE is kept intentionally simple, in that it only replicates events to multiple nodes, whereas more complex actions requiring DWS state are implemented by the replicated EDEs (REDEs). Each REDE node maintains its own DWS working set, and a relaxed synchronous replication protocol is used to maintain clients' perceptions concerning DWS consistency across REDEs.

Upon receipt of an event, the primary REDE node timestamps and mirrors the event to a set of mirror nodes, based upon a set of application-specific rules determining mirroring frequency and granularity, and using a degree of event filtering that complies with the consistency requirements of the application [15]. 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 update delays can be maintained within bounds that result in what clients perceive as zero downtime, under varying operating conditions. 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 [15].

Replicated Event Derivation Engine. The REDE (see Figure 2) consists of three different derivation engines that are 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 from the RE, locally performs the necessary timestamping of the events, and passes the events to the local business logic. We use vector timestamps, whose components correspond to a separate input stream, within which events' order is captured through their identifiers. Timestamps are uniquely computed based on event order within the original stream, the stream identifier, and a set of application-specific rules, in a monotonously increasing manner. Application semantics are captured though a set of *semantic rules* which describe the actions associated with different event types, or even their content, in terms of both the event mirroring and timestamp computations. At each node, the most current timestamp reflects the node's current view of the application's state. The mirror sites also maintain their views of the application's state through the vector timestamps of the events they receive from the REDE.

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 such a rate so as to maintain 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. The synchronization required to guarantee that EDE subscribers do not experience diminished consistency, 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 then 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, whose expiration implies failure occurrence. The frequency of the heartbeat traffic is expressed in terms of number of processed events, and its value should be set with consideration of both the average processing time of events in the system t_{proc}, and the timeout value t.

Detection. Within the duration of a single timeout, the pre- and post-buffer nodes can detect a primary site failure (timing or performance [16, 20]), and prepare to take over. 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, which implies that the EDE subscribers need to be capable of detecting and removing duplicate updates. In the event of 'ill-formed' messages, the update delays are longer, and in addition to the timeout value and the buffer sizes, depend also on the average processing time, and on the mechanism which detects and removes these erroneous messages. In either case, the REDE continues to stream newly derived data, and the failure of the primary node is hidden from its subscribers.

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 may occur with respect to global timestamp synchronization.

When a node is brought back up, it immediately starts buffering messages. It then sends a heartbeat message that contains the event's source identifier, and the event timestamp within its stream. Another node within the REDE responds with the vector timestamp computed for this event, or for the earliest consecutive event within the same stream. The responding site is either the pre-buffer, or if only one site is remaining, it is the primary buffer.

The recovering site issues a state request which contains this vector timestamp, either to a non-primary REDE node, or to a mirror site. The responding site awaits for its internal computation to advance to this level, if necessary[1], and then responds with the initial state set. The recovering node installs the received initial application state, and the associated vector timestamp, and starts processing the buffered stream of events. Whether the recovering node will maintain a pre- or a post-buffer is decided based on the currently active nodes within the REDE.

Flexible synchronization. Our implementation of relaxed synchrony replication within the REDE intuitively allows for consistency to be maintained with smaller overheads compared to other virtual synchrony protocols. At one end of the performance spectrum, by reducing the heartbeat rate to one per every processed event, and maintaining buffer sizes of one, we achieve strong consistency, at the cost

[1] The computation of both the mirror sites and the pre-buffer lags behind the primary site, so the vector timestamp may be ahead of these sites' internal views of application state.

of higher overheads. To address these overheads, we provide mechanisms through which the timeout values, the rate of the heartbeat traffic, as well as the lengths of the pre- and post-buffer, can be set by the application programmer, so as to best capture the fault tolerance semantics of the concrete application. Reducing these values increases the consistency level that exists among the REDE nodes, but also increases update delays and can push the system's performance level further away from the clients' expectations. Detailed studies need to be performed in order to retrieve the correlations existing between these parameters, network bandwidth and the experienced delays by the clients. We present some initial results of our evaluations in Section 4.

Implementation. The two-tiered REDE is implemented 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 [12] is used to efficiently move events across nodes. Thus, communication is carried out via multiple logical event channels among which the 'data' channels carry application-specific information, captured in data events, and bi-directional 'control' channels carry control and heartbeat events, necessary for guaranteeing consistent state across REDE nodes and mirrors. Control channels are also used to make changes to the ways in which mirroring is performed. Our implementation assumes a reliable communications infrastructure, which guarantees lossless ordered event delivery. This, combined with our application-specific timestamping, alleviates the need for complex ordering mechanisms.

Two types of incoming event streams exist in our application: one that carries FAA flight position information, and another that carries flight information alike to what is carried by Delta's internal system, such as current flight status (landed, taxiing), passenger and baggage information, etc. If an incoming event to be mirrored already has the appropriate format, then no data processing is necessary in the 'auxiliary unit', otherwise format translation is performed within every REDE node. Some additional mirroring-related processing is necessary in the case for the 'application-specific' and 'adaptive' mirroring, where incoming data is first extracted from the event stream, then filtered, and then converted into the appropriate outgoing event format.

4 Evaluation

Experiments are performed with the replicated server (REDE and mirror sites) running on the aforementioned cluster machine. 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 [18]. 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 our two-tier architecture, as a function of varying data sizes, by comparing the total processing time for a fixed event sequences of our replicated implementation, to one which 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 additional increase due to heartbeat traffic when a replicated EDE is used, adds another 5% to 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 behavior of our infrastructure in the event of failures, and show 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. The 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. We run each test 20 times, and present the average delays. The same tests were repeated for 'hardware' failures, for which the post-buffer replay is sufficient (lighter bars in Figure 4), and for 'ill-formed' message failure, for which the pre-buffer replica becomes the primary node (darker bars).

We have also implemented the BASE replication technique, which extends the Byzantine fault tolerance solution [8, 21]. Our implementation of BASE differs from the one described in [8, 21] by not using an abstraction function, thereby assuming that the concrete state is equivalent to the abstract state. The control messages carrying the timestamps of the nodes' most recent updates represent the

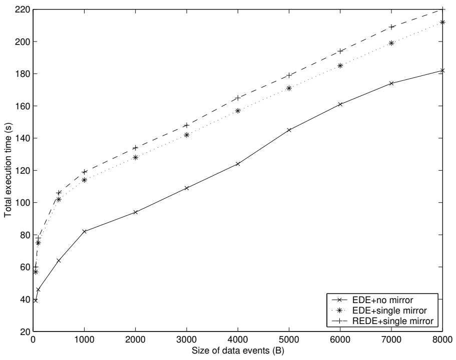

Figure 3: Overhead of implementation.

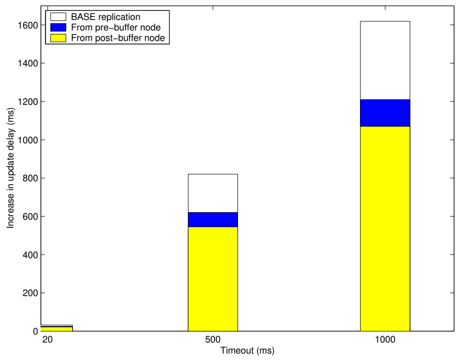

Figure 4: Average update latency increase as a result of failure.

checkpoints and incremental state transfers used by BASE. Furthermore, we do not implement proactive recovery and restart, nor the propose_value-check_value mechanism which inspects the correctness of the primary. We simulate the distinction in the implementation of the replicas, which allows for software errors to be captured in BASE by removing an 'ill-formed' message from the pre-buffer of the non-primary node. As a result, the implementation of BASE used in this paper is simpler than the one described in [21], so that its measured performance provides a lower bound for BASE. Nonetheless, the replication technique used by the REDE outperforms BASE in all cases. Measurements demonstrating this fact appear in Figure 4, in which the additional overheads of BASE compared to our technique are represented by the white bars.

Experimental results indicate that on average, the increase in the event update latency experienced by the REDE is less than 9% greater 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%. In comparison, when using BASE, update latency is increased by more than 60%. This increase is mostly due to the higher communication overheads resulting from keeping BASE replicas closely synchronized. 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, or (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. Measurements are performed for different sizes of initial states requested by clients. The results presented in Figure 5

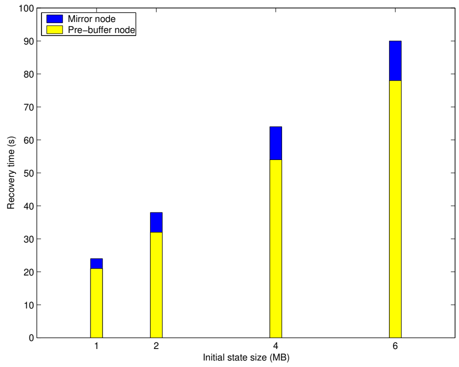

Figure 5: Recovery times when pre-buffer or mirror node is used to provide initial states.

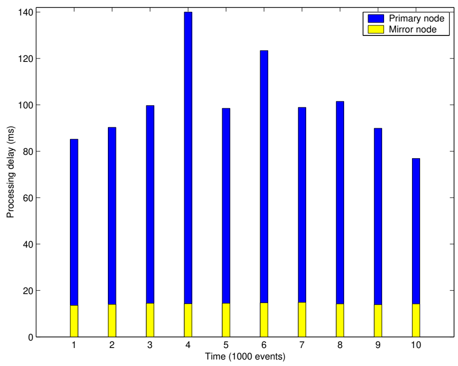

Figure 6: Comparison of processing delays when a mirror node provides the initial states vs. the primary REDE node.

shown 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. This difference is 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 increase in some cases more than 10 times. Such an order of magnitude increase will probably result in perceivable down time 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 motivation of our two-tiered design by analyzing the impact of increases in the clients' request load on the update delay. The darker bars in Figure 7 show the increases in average update delays, which result from increased rate of incoming clients' requests. Separating the request servicing functionality, reduces the direct impact of the request rate on the delays. 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, and by reducing the overall consistency between the mirrors, and the REDE

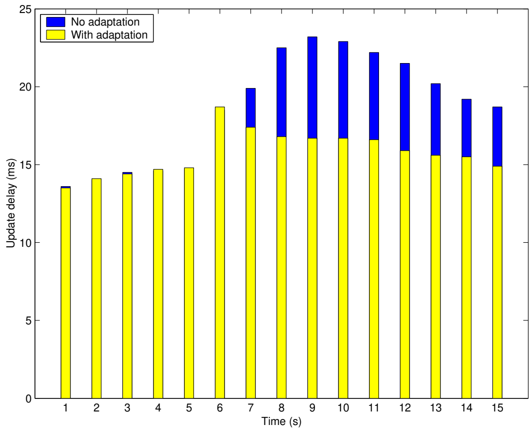

Figure 7: Controlling increases in update latencies through adaptations in the mirroring process. A burst of client requests occurs during the time interval 5-10 sec.

nodes, to continue publishing the state updates, with minimal oscillations in their values. 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 which fall within the regular service levels, and perceive no down-time, as a result of the system overload during the unusual operating conditions.

Discussion of results. The experimental analyses of our approach indicate that the two-tier REDE architecture masks node failures from clients with relatively low overheads, 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 few minutes after the node is brought back up. Finally, we justify the two-tiered REDE design first 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, and then by showing 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, 6, 7, 8, 10, 11, 25]). As with our solution, most approaches rely on a checkpointing mechanism, through which they track each other's progress, and a log of events that need to be either undone or redone when a failure occurs [3, 13]. 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 face or failures [4, 6, 7, 8] or erroneous assumptions in optimistic synchronization solutions [23].

Well-known systems like Horus [25] and Transis [11] 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. In addition, our principal goal is to achieve what appears to be zero downtime to the EDE event subscribers. One outcome is the two-tier nature of the REDE described in this paper.

Concrete applications often motivate decisions regarding the levels of consistency to be maintained across replicas. Many existing systems provide a mechanism through which this can be dynamically specified and controlled by the application [10, 16]. 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, the replication mechanism we use is optimized by using a relaxed synchrony approach, which is enabled by the streaming nature of our application domain. Hiltunen et al. [16], TACT [26], AAS [10] and other systems, propose similar optimizations, 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 [8, 21], is a technique for handling the problem of deterministic software failures by replicating the affected service. They rely on non-deterministic behavior across multiple, potentially different system implementations, similarly to N-versioning [9], to increase the probability that not all implementations will fail. They strictly maintain pre-buffers, and block the advancement of individual replicas ahead of the primary. In addition, the primary cannot modify system state without consulting the other replicas, via a voting procedure. In comparison, our approach is the combined use of pre- and post- buffers. This allows us to prevent a system-wide failure caused by deterministic software faults, while still benefiting from lightweight, optimistic synchronization under other circumstances.

AAS [10] is a fault-tolerance approach that also targets the particular requirements of the airline industry, recognizing the different failure semantics and fault-tolerance requirements that exist in this domain. The approach proposes different levels of replication and different degrees of synchronization, for different types of services. This approach is more general than ours. In comparison, we exploit the streaming nature of our data, to make more specific decisions about the replication of the publishing node, and to define the replicas' degree of synchronization through pre- and post-buffers.

The use of micro-protocols [16] 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 publish data with bounded update delays, and in addition, to process their explicit request. To meet these service requirements, this paper describes and evaluates a two-tier 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 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, often bursty requests from clients for information, and (2) to dynamically vary consistency between nodes in order to deal with changes in operating conditions, such as sudden bursts of requests, thereby offering to client a 'zero downtime' view of the system.

Our ongoing research is continuing to focus on the reliable diffusion of events with bounded delays, also 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 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] Mustaque Ahamad and Rammohan Kordale. Scalable Consistency Protocols for Distributed Services. *IEEE Transaction on Parallel and Distributed Systems*, 1999.

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

[3] Lorenzo Alvisi, Karan Bhatia, and Keith Marzullo. Tracking Causality in Causal Message-Logging Protocols. *to appear in Distributed Computing*, 2001.

[4] Gerco Ballintijn, Maarten van Steen, and Andrew 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, August 1999.

[5] J. Becla and A. Hanushevsky. Creating Large Scale Database Servers. In *Proc. of Ninth Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.

[6] Kenneth Birman, Andre' Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[7] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budio, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[8] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proc. of the 4th OSDI*, San Diego, CA, October 2000.

[9] 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.

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

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

[12] Greg Eisenhauer, Fabian Bustamante, and Karsten Schwan. Event Services for High Performance Computing. In *Proc. of Ninth High Performance Distributed Computing (HPDC-9)*, Pittsburg, PA, August 2000.

[13] Mootaz Elnozahy and Willy 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.

[14] Enterprise System Architecture/390 - Principles of Operation. IBM Corporation. http://www-3.ibm.com/software/ad/hlasm/ library/pdf/39xsm.pdf.

[15] Ada Gavrilovska, Karsten Schwan, and Van Oleson. Adaptible Mirroring in Cluster Servers. In *Proc. of 8th High Performance Distributed Computing (HPDC)*, San Francisco, CA, August 2001.

[16] Matti A. Hiltunen, Vijaykumar Immanuel, and Richard D. Schlichting. Supporting Customized Failure Models for Distributed Services. *Distributed System Engineering*, 6:103–111, December 1999.

[17] Robert E. McGrath, Joe Futrelle, Ray Plante, and Damien Guillaume. Digital Library Technology for Locating and Accessing Scientific Data. *ACM Digital Libraries '99*, pages 188–194, August 1999.

[18] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. In *Workshop on Internet Server Performance'98*, Madison, WS, June 1998.

[19] Van Oleson, Karsten Schwan, Greg Eisenhauer, Beth Plale, Calton Pu, and Dick Amin. Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, October 2000.

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

[21] Roberto Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using Abstractions to Improve Fault Tolerance. In *Proc. of 18th ACM SOSP*, Banff, Canada, October 2001.

[22] Yasushi Saito, Brian Bershad, and Henry 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, December 1999.

[23] Jeremy Sussman, Idit Keidar, and Keith Marzullo. Optimistic Virtual Synchorny. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Nurnberg, Germany, October 2000.

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

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

[26] Haifeng Yu and Amin Vadhat. The Costs and Limits of Availability for Replicated Services. In *Proc of 18th ACM SOSP*, Banff, Canada, October 2001.