Adaptable Mirroring in Cluster Servers

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

Abstract

This paper presents a software architecture for continuously mirroring streaming data received by one node of a cluster-based server to other cluster nodes. The intent is to distribute the load on the server generated by the data's processing and distribution to many clients. This is particularly important when the server not only processes streaming data, but also performs additional processing tasks that heavily depend on current application state. One such task is the preparation of suitable initialization state for thin clients, so that such clients can understand future data events being streamed to them. In particular, when large numbers of thin clients must be initialized at the same time, initialization must be performed without jeopardizing the quality of service offered to regular clients continuing to receive data streams.

The mirroring framework presented and evaluated here has several novel aspects. First, by performing mirroring at the middleware level, application semantics may be used to reduce mirroring traffic, including filtering events based on their content, by coalescing certain events, or by simply varying mirroring rates according to current application needs concerning the consistencies of mirrored vs. original data. The intent of dynamically varied mirroring is to improve server scalability, both with respect to its ability to stream data events to a large number of clients and to deal with large and highly variable request volumes from clients that require other services, such as new initial states computed from incoming data events. Second, we present an adaptive algorithm that varies mirror consistency and thereby, mirroring overheads in response to changes in clients' request behavior. Third, our framework not only mirrors events, but it can also mirror the new states computed from incoming events, thus enabling dynamic tradeoffs in the communication vs. computation loads imposed on the server node receiving events and on its mirror nodes. This capability is used for adaptive event coalescing in response to increases or decreases in client request loads.

*Van Oleson, a Ph.D. student at Georgia Tech, is also employed by Delta Airlines IT Division

1 Operational Data Services and Servers

Server systems structured as cluster machines have become increasingly common, to drive search engines [1], operate mail servers [2], or provide scientific data or computational services [3, 4]. Our research concerns one important property of such cluster servers, which is their ability to continue to provide high levels of service even under increasing loads or when clients' request behaviors vary dynamically.

Operational Information Systems. In contrast to the interactive high performance applications considered in our previous work [5, 6], this paper addresses an emerging application domain for high performance computing and for the cluster servers on which these applications run, which is that of operational information systems [7] (OIS). 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 we have been studying is the OIS run by Delta Air Lines, which provides the company with uptodate information about all of their flight operations, including crews, passenger, and baggage. 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 [8], 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 FAA radar data, 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 [9], 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 databases in which operational state changes are recorded for logging purposes.

- *Responses 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.

Problem: predictable performance despite bursty requests. The ability to capture and manipulate operational information in a timely fashion is critical to a company's operations and profits. The problem addressed by this paper is how to offer predictable performance in an operational information system, even in the presence of bursty client requests. Stated more explicitly, for an OIS this means that (1) the central server must continue to process captured data and update operational state, (2) the server must continue to distribute state updates to clients, and (3) it must respond to explicit client requests in a timely fashion. In this example, predictability can be quantified precisely: Delta Air Lines currently requires that no essential operational state is lost, that the distribution of state updates be performed within less than 30 seconds after they have been received, and that client requests like those for state initialization must be satisfied within less than a minute.

Predictability requirements vary across applications. In multimedia systems, clients may be sent lossy data as long as real-time data rates are maintained. In Delta's OIS, acceptable delays for lossless data are determined by the business processes being applied to operational data and by the need to offer what appears to be 'zero' downtime and consistent flight information to gate agents' and passengers' views of such data. For applications like IBM's information services for the Atlanta Olympic Games, even small delays were devastating: both television viewers and journalists were disappointed when IBM's servers could not keep up with bursty requests for updates while also steadily collecting and collating the results of recent sports events in a timely fashion.

Solution approach: dynamic mirroring in cluster servers. We offer the following solution to achieving predictable performance for large-scale data services like Delta's OIS: (1) cluster machines serving as data servers internally use high bandwidth, low latency network connections and can be connected to the remainder of the OIS via multiple network links, and (2) server-internal mechanisms and policies dynamically 'spread' the operational state being computed across the cluster machine, as well as distribute client requests across the cluster. The intent is to continuously provide timely service for both operational data processing and client requests.

The technical approach implemented and evaluated in this paper is that of *data mirroring*, where captured data structured as application-level *update events* is mirrored to all other cluster machines interested in such events. For such update events, all mirror machines execute the same business logic, thus resulting in the OIS's operational state being naturally replicated across all cluster machines participating in event mirroring. The result of such mirroring is that clients' requests for IS state may be satisfied not just by one, but by any one of the mirror machines. The resulting parallelization of request processing for clients coupled with simple load balancing strategies enables us to offer timely services to clients even when request loads become high.

Mirroring of update events and the resulting load balancing of client requests are made feasible in part by the architecture of the cluster computer used for implementing the OIS. This is because intra-cluster communication bandwidth and latency are far superior to those experienced by data providers and by clients. Furthermore, we share the assumption with other cluster-based server systems [2] that OIS software is structured such that all cluster machines can perform all OIS tasks. This assumption is reasonable since it is not the complexity of OIS 'business logic' that causes performance problems but instead, problems stem from the potentially high rates of incoming operational data and the large number of clients for server output (numbering in the 10's of thousands for Delta's OIS, for example). Finally, we do not assume that all cluster machines act as mirrors and that all OIS state is replicated. Instead, since OIS state can grow to many gigabytes, in general, update events must be mirrored both to sites that replicate local state and to sites that need such events for functionally different tasks. This also implies that in general, client requests must be distributed both according to the functional distribution of OIS software on the cluster and the mirroring of state that is being performed. This paper ignores the functional distribution of OIS state and instead, focuses on event mirroring, its performance implications, and its utility.

Advantages of mirroring. We seek two advantages from data mirroring. First, by distributing client requests across mirrors, request loads may be distributed across multiple cluster machines, thus responding to client requests in a timely fashion even under high load conditions. Second, by reducing the perturbation imposed on the OIS from bursty client requests, the predictability with which the OIS server can continuously capture, process, and deliver operational information is improved. Results presented in Section 4 demonstrate that a cluster machine can mirror update events to a moderate number of cluster machines without imposing undue loads on the cluster interconnect, and that the perturbation caused by client requests can be controlled in this fashion. In addition, prior work has established that simple request load balancing algorithms offer good performance on cluster server machines [1, 10]. Another, well-known reason for data mirroring is the increased reliability gained from the availability of critical data on multiple cluster nodes [11], a topic not explored in detail herein.

Performance issues and solutions. The basic technical problem we address for cluster servers used in operational information systems is their scalability in face of dynamic variations in client request patterns and/or of the captured information being processed. The scalability metric used captures the predictability of the services implemented by the server: how does a server react to additional loads it experiences, with respect to deviations in the levels of service offered to its 'regular' clients? With this metric, high server scalability implies small deviations for large additional loads. The intent of this metric is simple. It measures how an operational information system reacts to unusual or irregular operating conditions, with respect to its continued ability to provide information to its clients at the rates and with the latencies dictated by ongoing business processes. In the case of an airline, unusual operating conditions include (1) 'bringing up' an airport terminal after a power failure and (2) dealing with inclement weather conditions. Case (1) requires the airport terminal's many 'thin clients' (e.g., airport displays) to be re-supplied quickly with suitable initial states, thereby once again enabling them to interpret the regular flow of data events issued by the server. Case (2) results in changes in the distributions of captured events and therefore, changes in the loads imposed on certain service tasks. For instance, in inclement weather conditions, it would be appropriate to track planes at increased levels of precision, thus resulting in increased loads on servers caused by the additional tracking processing and in increased communication loads due to the distribution of tracking data.

Both Cases (1) and (2) have similar effects on the data server: when faced with these additional loads, it cannot continue to issue output events at the rates required by its clients. Experiments shown in Section 4 demonstrate this fact. However, these experiments also demonstrate a problem caused by event mirroring, which is that the overheads of event mirroring itself can be significant. For instance, mirroring can result in a 30% slowdown on our cluster machine when there are 4 mirror machines. The remainder of this paper investigates technical solutions to address this problem and thereby establish the practicality of event mirroring:

- *Mirroring overheads* – the intent of mirroring is to improve server scalability, in part by offloading tasks from certain server nodes. Yet, at the same time, mirroring overheads can slow down server nodes and thereby reduce scalability. The performance effects of mirroring experienced on a cluster machine are explored in detail in Section 4.1.

- *Application-specific mirroring* – mirroring differs from message broadcast or multicast in that it is performed at the application level. This enables us to substantially reduce mirroring traffic compared to implementations described previously [11], by filtering events based on their data types [12] and/or their data contents [13], by coalescing certain events particularly when the effects of a later event overwrites those of previous events, or by simply varying mirroring rates according to current application needs concerning the consistency of mirrored vs. original data. In Section 4.2, we experimentally evaluate the performance implications of some of these choices.

- *Adaptive mirroring* – overheads in mirroring are due to the rates at which data events are mirrored and the degrees of data consistency across multiple cluster nodes enforced by the mirroring mechanism. Thus, by reducing rates or consistency when servers experience increased request loads due to unusual operating conditions, server scalability can be improved substantially. We also present a dynamic algorithm for mirroring adaptation and evaluate its impacts on server scalability in Section 4.3.

Remainder of paper. In the remainder of this paper, we first explain in more detail the OIS application that drives our research. Next, we briefly outline a software architecture for event mirroring, describe the mechanisms behind application-specific and adaptive mirroring, and present some implementation detail. Section 4 presents experimental results attained on cluster machines at Georgia Tech. At the end, we compare our work to related research, followed by concluding remarks and future directions.

2 The OIS Application

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. The data

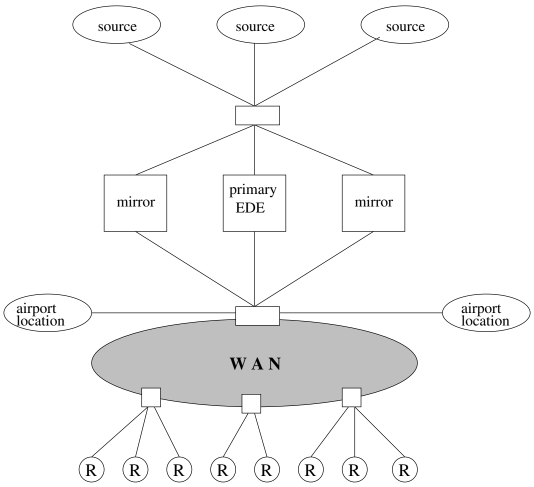

Figure 1. Delta's Operational Information System

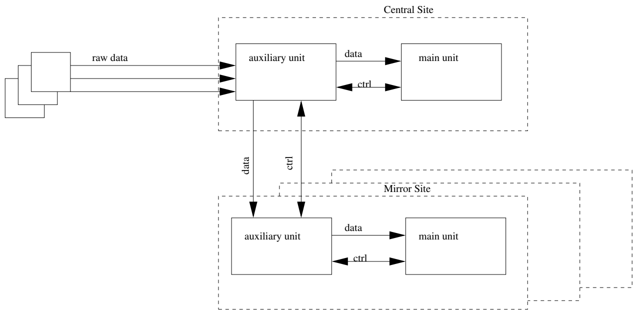

Figure 2. Architecture of the mirroring framework

events being captured from, processed, 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). In the case of Delta Air Lines, events are captured at system entry points like gate readers or radar data collected by the FAA, distributed via company-internal or leased network links, and processed by servers located at the company's processing center in Atlanta. The outputs generated by the center's server are used by a myriad of clients, ranging from simple airport flight displays to complex web-based reservation systems.

The code executed by the OIS server is what we term its Event Derivation Engine (EDE). EDE code performs transactional and analytical processing of newly arrived data events, according to a set of business rules. 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 EDE also services incoming clients' requests. For instance, it provides clients with initial views of the states of operational data on demand. Once they receive these initial views, clients maintain their own local views of the system's state, which they continuously update based on events received from the OIS server.

3 Event Mirroring – Software Architecture and Implementation

3.1 Software Architecture

Basic architecture. The prototype architecture of our mirroring framework is depicted in Figure 2. Events may be provided by any number of data sources, and they are processed by a 'central' site and its set of mirror machines. In our current implementation, a single central site located on the cluster machine acts as the primary mirror, with other sites acting as secondary mirrors. In an actual, deployed operational system, it should be possible to separate data mirroring from processing functionality, thereby permitting us to use application-specific extensions [14, 15] of operating system kernels of communication co-processors (i.e., network interface boards) to reduce mirroring overheads. The resulting software architecture used for data mirroring, then, separates the application-specific code (i.e., business logic) executed by the Event Derivation Engine from the code that implements mirroring (labelled as 'main' and 'auxiliary' units in the figure). Events produced by each mirror site are typically generated by business logic, in response to receiving multiple incoming update events. All mirrors produce the same output events, and produce identical modifications to their locally maintained application states.

The main functionality of the central site is to distribute continuous state updates, while a mirror site's primary task is to respond to client requests. Three separate tasks execute within the auxiliary unit of the central site: (1) a receiving task, (2) a sending task, and (3) a control task. Their execution is synchronized via accesses to common data queues, which include the *ready* and the *backup* data queue, a *status table* containing relevant status information for application-level processes (e.g., flight status), and a set of parameters determined by the application and embodying the *semantic rules* that are to be followed by the mirroring process. A sample semantic rule is one that tells the mirroring process to eliminate certain incoming events when mirroring them (examples appear below). The receiving task retrieves events from the incoming data streams, performs the timestamping and event conversion when necessary, and places the resulting events into the ready queue. Events are removed from the ready queue, sent onto all outgoing channels, and temporarily stored in the backup queue by the sending task. The control task runs various control routines, including (1) checkpointing, upon which all successfully checkpointed events are removed from the backup queue, and (2) the exchange of values of monitored variables and new settings for the mirroring parameters during the adaptation process, as explained further in Section 3.2.2. Additional control tasks under development include recovery in the presence of ill-formed communications or failures in EDE processing.

3.2 Mirroring Mechanisms

This section describes the mechanisms with which data mirroring is performed, the API used by applications to modify mirroring, and the checkpointing procedures that allow individual nodes to advance their application views while maintaining desired consistency levels across nodes.

3.2.1 Application-Specific Mirroring

Mirroring is triggered by the auxiliary unit at the central site, and its execution involves multiple tasks within this unit. For default mirroring, the 'receiving task' computes for each event a timestamp and places it onto the ready queue, the 'sending task' removes events from the ready queue, mirrors them onto all of its outgoing channels and preserves a copy in the backup queue, and the 'control task', upon exchanging appropriate control events during the execution of the checkpointing procedure, performs synchronization to ensure consistent views of application state across all sites. It also updates the backup queues appropriately.

If no arguments are specified during the initialization process, the default mirroring function mirrors each event to the auxiliary units of all mirrors sites, using *mirror()* calls. It also forwards all such events to the main unit, using *fwd()* calls. The checkpointing mechanism is invoked at a constant frequency of once per 50 processed events, using *checkpoint()* calls. The distinction between event 'mirroring' and 'forwarding' serves to distinguish the events recovering clients receive from mirror sites from the events received by 'regular' clients on the main site.

Default mirroring can be modified during the initialization process or dynamically, using an API that affects several mirroring parameters. Changes in mirroring parameters trigger changes in the semantic rules associated with mirroring. Parameters include (1) an indicator whether events are to be mirrored independently, or whether multiple events are to be coalesced before mirroring, (2) the maximum number of events that can be coalesced, (3) whether overwriting is allowed for a particular event type, (4) the maximum length of the sequence of overwritten events, (5) the frequency at which the checkpointing procedure is invoked, (6) and parameters associated with the adaptation mechanism explained in Section 3.2.2. These can either be passed as arguments of the *init()* call, or they can be set directly with the calls *set_params()*, *set_overwrite()*, *set_complex_seq()*, *set_complex_tuple()* and *set_adapt()*.

The implementation of mirroring uses state to keep track of event history, such as the number of overwriting events or the values of combined events. Event coalescing is performed by the sending task. The receiving task is responsible for discarding events in an overwriting sequence of events, or for combining events based on event values. The status table is used during this process to keep track of number of overwritten flight updates for a particular flight, value of a particular event that has an action associated with it, etc.

If an overwrite action is set for an event type, for example the event containing a certain flight's position data, with a sequence length *max_length*, then the mirroring function will send one event for each flight, followed by discarding the next *max_length-1* many events of that type for the same flight. This is implemented using history information recorded in a status table maintained at the main site. If FAA position update events arrive after a Delta 'flight landed' event has already been received for the same flight, then the FAA events can be discarded with a call to *set_complex_seq(event_type Delta, event *target_value, event_type FAA)*, where target_value is Delta event whose status field value is 'flight landed'. Similarly, multiple events like 'flight landed', 'flight at runway', and 'flight at gate' can be collapsed into a single complex event, termed 'flight arrived'. The presence of such an event implies that all position events for that flight can be discarded from the queues.

Programmers can provide their own mirroring or forwarding functions, thereby customizing how these actions are performed, using the calls *set_mirror()* and *set_fwd()*. A complete description of the API supported by our infrastructure appears in Table 1.

Checkpointing. The current implementation of mirroring is coupled with a slightly modified version of a standard checkpointing mechanism [16]. This version assumes reliable communication across mirror sites. Lack of event loss, combined with the fact that application-specific information is used in the mirroring process to discard or combine data events, implies that a new checkpoint can be selected before all events with in-sequence timestamps have been received.

The checkpointing procedure is invoked by the control task at the central site's auxiliary unit. It executes at a rate expressed in terms of the number of events sent. This rate can be dynamically set with the call *set_params()*. The auxiliary unit is also the coordinator of a modified 2-phase commit protocol in which all mirroring sites participate. The protocol is non-standard in that during the voting phase, the main unit issues a CHKPT control event and suggests a timestamp value up to which the consistent view can be advanced; this value is usually the most recent value found in its backup queue. When replies are received from the

function	description
init(int c, int number, int l)	initialize the mirroring with default or optional parameters
mirror()	execute mirroring function
fwd()	executes forwarding function
set_mirror(void* func)	set new mirroring function *func*
set_fwd(void* func)	set new forwarding function *func*
set_params(int c, int number, int f)	coalesce (*c*) up to *number* events; set checkpointing at *f*
set_overwrite(ev_type t, int l)	allow overwriting of events of *t* with max length of sequence *l*
set_complex_seq(ev_type t1, event *value, ev_type t2)	discard events of *t2* after event of *t1* has *value*
set_complex_tuple(ev_type *t, event *values, int n)	combine *n* events with respective types and values
set_adapt(int p_id, int p)	modify parameter *p_id* by *p* percent
set_monitor_values(int index, int p, int s)	for monitored variable *index* set primary *p* and secondary *s* threshold

Table 1. Mirroring API

mirror sites, a common timestamp value is computed, and in the second phase, a commit is issued for this value. There are no 'No' votes in our implementation, and no 'ABORT' messages.

```
Central Aux. Unit:

init_CHKPT: {
  chkpt = last on backup queue
  sent to all chkpt_event
}
CHKPT_REP: {
  commit = min from all chkpt_reply
  send to all commit_event
}
```

```
Main Unit:

CHKPT: {
  chkpt_rep = min{chkpt, last in backup}
  send to aux. chkpt_rep
}
COMMIT: {
  if commit in backup queue
    update backup queue
}
```

```
Mirror Aux Unit:

CHKPT: {
  forward to main unit
}
CHKPT_REP: {
  if chkpt_rep in backup queue
    forward to central cite
}
COMMIT: {
  if commit in backup queue
    update backup queue
    forward to main unit
}
```

Figure 3. Checkpointing

Since the auxiliary units depend on the main units to make decisions about safe, committable timestamp values, control messages are exchanged between mirror and main sites. These control messages attempt to agree upon the most recent event processed by the sites' business logic, prior to the one indicated in the CHKPT message. These values are collected at the central site, their minimum is computed, and during the commit stage, each unit can discard events from its backup queue up to this event.

It is not necessary to wait for additional acknowledgments in the commit phase, since (1) if a mirror site fails, these events have already been processed by all main units, or (2) if a control event is lost, the subsequent checkpointing calls will result in commits of more recent events. These events can be used to update sites' backup queues. Timeouts are not used, since if a checkpointing procedure has not completed a commit before the following one is initiated, the later commit will encapsulate the earlier one. If a unit receives a commit identifying an event no longer in its backup, this event is ignored. Reliable underlying communications imply that checkpointing will commit eventually. Figure 3 summarizes the checkpointing activities undertaken by different participants.

3.2.2 Adaptive Mirroring

The motivation behind application-specific mirroring is to create mechanisms through which the level of consistency across the system can be traded against the quality of service observed by 'regular' clients. Since system conditions change dynamically, it is important to change mirroring parameters at runtime, as well. That is, mirroring itself should be adapted to current system conditions, application behavior, and application needs. One interesting change in application behavior is the sudden receipt of many simultaneous client requests for new initial states, perhaps due to a power loss and recovery they have experienced. In cases like these, one useful response is to reduce the level of consistency maintained across mirror sites, in order to free up execution cycles and network bandwidth for quickly responding to such client requests and to reduce the perturbation in the level of service observed by 'regular' clients.

Dynamic tradeoffs in system consistency vs. quality of service are implemented by dynamically modifying a mirroring function and/or its parameters. In addition, the runtime quantities that impact such adaptation decisions are dynamically monitored. The simple adaptation strategy implemented in this paper is one that maintains for such monitored variables two values: a *primary* and a *secondary* threshold value, both of which are specified by the application through *set_monitor_values()*. The primary value, when reached, triggers the modifications of the mirroring algorithm. The secondary value indicates the range within which this modification remains valid. That is, the re-installation of the original mechanism takes place when the monitored value falls below *(primary - secondary)*.

In our current implementation, while the monitored decision variables are dispersed across mirror sites, adaptation decisions are made at the main site, thereby ensuring that all mirrors are adapted in the same fashion. To avoid additional 'adaptation traffic' between main and mirror sites, adaptation messages are piggybacked onto checkpointing messages. Changes in client behavior and/or changes in system conditions that result in degraded client service are monitored by inspecting the lengths of the ready and backup queues in mirror sites. Clearly, the lengths of these queues are correlated to the request load at each mirror. The same is true for the length of an application level buffer holding all pending client requests. Future work will consider other ways of measuring current system performance.

The adaptations of mirroring currently implemented in the system include the following:

- coalesce multiple events vs. mirroring them independently;
- set the maximum number of events to be coalesced;
- set the maximum number of events that can be overwritten in a sequence;
- vary checkpointing frequency; and
- install a different mirroring function.

With the API call *set_adapt()*, we identify which of these parameters are to be modified if the threshold value is surpassed, and by what extent.

3.3 Implementation

We use the ECho event communication infrastructure [6] to efficiently move data across machines. Thus, communication is carried out via multiple logical event channels, both between mirrors and among sources, mirrors, and clients. Two separate event channels exist between the auxiliary and the main unit on each site, as well as among the auxiliary units: a 'data' channel carries application-specific information, captured in data events, and a bi-directional 'control' channel carries control events, necessary for guaranteeing consistent state across mirrors. Control channels are also used to make changes to the way in which mirroring is performed.

Two types of event streams exist in our application: one that carries FAA flight position information, and another that carries Delta's internal flight information, such as current flight status (landed, taxiing), passenger and baggage information, etc. For both channels, events themselves are uniquely timestamped when they enter the primary site. We use vector timestamps in which each vector component corresponds to a different incoming stream, and we assume that the event order within a stream is captured through event identifiers unique to each stream. If an incoming event to be mirrored already has the appropriate format, then no mirror-related processing is necessary in the 'auxiliary unit'. That may not be the case for the 'application-specific' and 'adaptive mirroring' demonstrated in Section 3.2, where incoming data is first extracted from the event stream, then filtered, and then converted into the appropriate outgoing event format.

4 Experimental Results

Experiments with data mirroring first diagnose the resulting overheads encountered during OIS execution. Adaptive mirroring is introduced to reduce these overheads. Toward this end, we first evaluate the benefits of using certain application-specific information to reduce mirroring traffic and frequency, including the effects of this approach on sustainable request loads and on the update delays experienced by the application. Finally, we analyze the performance implications of dynamic modifications to the mirroring function on overall application performance, in terms of the predictability of the service levels offered to clients.

Experiments are performed with a mirrored server running on up to eight nodes of a cluster of 300MHz Pentium III dual-processor servers running Solaris 5.5.1. The 'flight positions' data stream used in these experiments originates from a demo replay of original FAA streams, and it contains $100,000$ 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 [17]. Httperf clients run on 550MHz Pentium III Xeon nodes connected to the server cluster via 100Mbps ethernet.

4.1 Mirroring Overheads

A set of microbenchmarks measures mirroring overheads as functions of the data size and the number of mirror sites. No additional load is assumed at any of these sites as a result of incoming client requests. The predictability of the server's execution is observed throughout the microbenchmarks' duration. The experiment terminates after the entire sequence of events presented to the mirroring system has been processed.

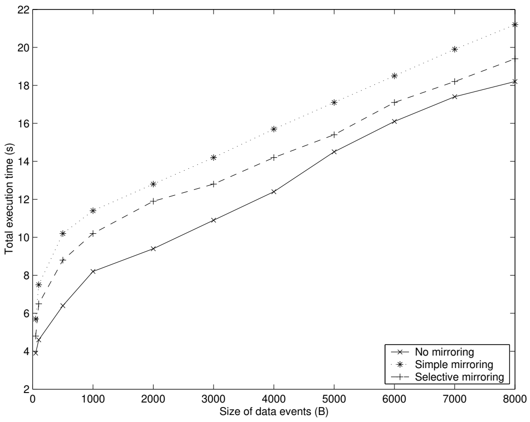

Figure 4. Overhead of mirroring to a single site with 'simple' and 'selective' mirroring

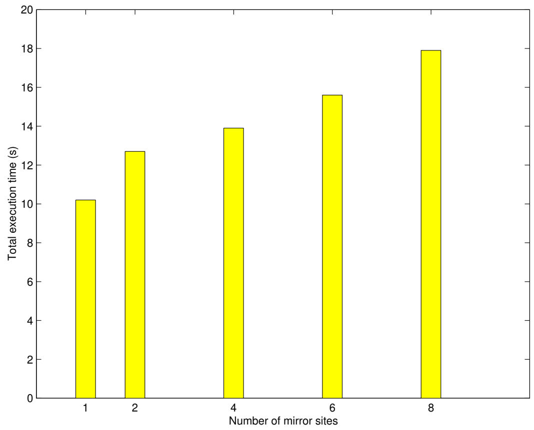

Figure 5. Overheads implied by additional mirrors

Results presented in Figure 4 indicate that an approximately 15-20% increase in execution time is experienced when all incoming events are mirrored to a single mirror site, the overhead being greater for larger event sizes. This increase is due to event resubmission, thread scheduling, queue management and execution of the control mechanism. Overheads can be reduced by replacing the simple mirroring function with one that mirrors events selectively. In the case of FAA data, when using a selective mirror function that mirrors only the most recent event in a sequence of up to 20 overwriting events, overhead is reduced significantly, with reductions again becoming more pronounced for larger event sizes (see the dashed line in Figure 4).

The next set of measurements diagnose the effects of increasing the number of mirrors, while maintaining constant data sizes for the events being mirrored. Results indicate that on the average, there is a less than 10% increase in the execution time of the application when a new mirror site is added (see Figure 5).

4.2 Application-Specific Mirroring

Next, we investigate our infrastructure's impact on server performance and scalability, by analyzing the performance of a mirrored server under varying client requests loads, using different mirroring functions. Each experiment compares the total processing time for the same event sequence, for mirrored servers with one, two and four mirror sites, respectively. In all cases, we operate under a constant request load evenly distributed across mirror sites. Our results presented in Figure 6 consider the total time taken for both processing the entire event sequence and also servicing all client requests. The results show that for data sizes larger then some cross-over size (where experimental lines intersect), mirroring overheads can be outweighed by the performance improvements attained from mirroring. Smaller total execution times also indicate that the 'regular' state updates issued by the server are executed more promptly. This indicates the improved server predictability we are seeking to attain by data mirroring.

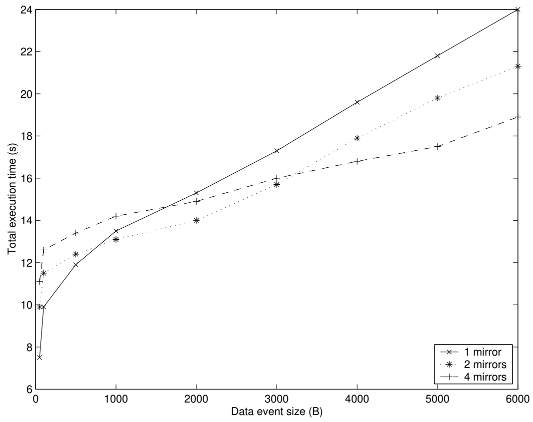

Figure 6. Mirroring to multiple mirror sites, under constant request load of 100 req/sec balanced across the mirrors

Server scalability may be improved by using an application-specific mirroring function that performs small amounts of additional event processing in order to filter out some of the events, thereby reducing total mirroring traffic. For varying request loads and for one mirror, we compare the use of a simple vs. a 'selective' mirroring function. Results indicate that selective mirroring can improve performance by more than 30% under high request loads (see the dotted line in Figure 7). Further adjustments of other mirroring parameters can result in additional improvements. For instance, by decreasing the checkpointing frequency by 50%, total execution time is reduced by another 10%, resulting in a total reduction of more than 40% in the total time required to both process our event sequence and service some fixed, total number of client requests. This is shown with the dashed line in Figure 7.

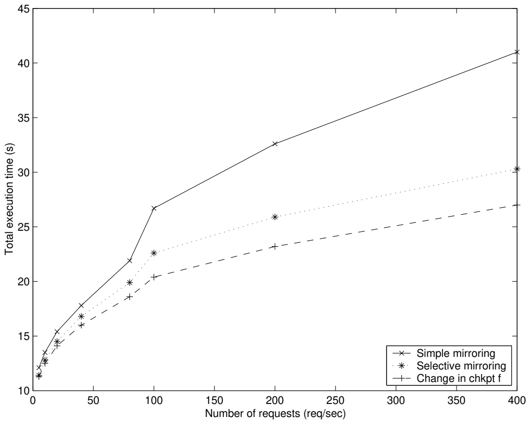

Figure 7. Comparison of three mirroring functions: 'simple', 'selective', and 'selective' with decreased checkpointing frequency

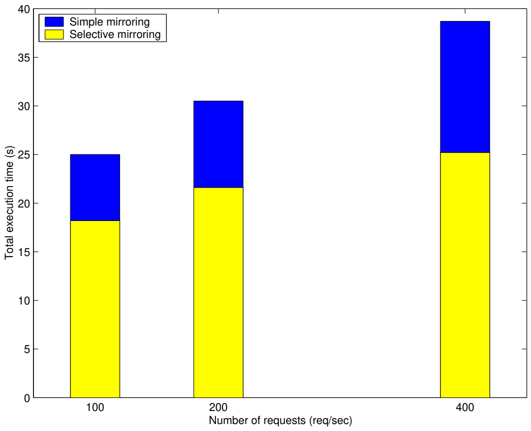

Figure 8. Update delays with 'selective' vs. 'simple' mirroring

Selective mirroring also affects the update delays experienced by operational data clients. Experimental results suggest that the experienced 40% reduction in the total execution time corresponds to a decrease in the average update delay experienced by clients of more than 50%, as presented in Figure 8.

4.3 Adaptive Mirroring

The experiments described in Section 4.2 suggest that there is no single 'best' strategy for mirroring events under certain client loads. Instead, such mirroring should be adjusted dynamically, as client request loads increase or decrease, thus continuously adjusting the mirroring system to provide suitable OIS services while also responding to clients in a timely fashion.

Adaptive mirroring continually monitors the lengths of the ready and backup queues, as well as the sizes of the application-level buffers of pending clients requests. Based on these monitored values and using a set of predetermined threshold values, we alternately use two mirroring functions. The first one coalesces up to 10 events and then produces one mirror event, thus overwriting up to 10 flight position events. Checkpointing is performed for every 50 events. The second function overwrites up to 20 flight position events and performs checkpointing every 100 events. We then compare the OIS execution that adaptively selects which one of these functions to use, depending on the sizes of monitored queues and buffers, with an OIS execution that performs no such runtime adaptation. Comparisons are performed with bursty clients requests pattern. The performance metric is the processing delay experienced by events from the time they enter the OIS system until the time they are sent to clients by the EDE at the central site. In other words, we are evaluating the OIS' contribution to the perturbation experienced by OIS clients receiving state updates.

The results in Figure 9 show the benefits of even the simple method of runtime adaptation described above. Specifically, total processing latency of the published events is reduced by up to 40%, and the performance levels offered to clients experience much less perturbation than in the non-adaptive case.

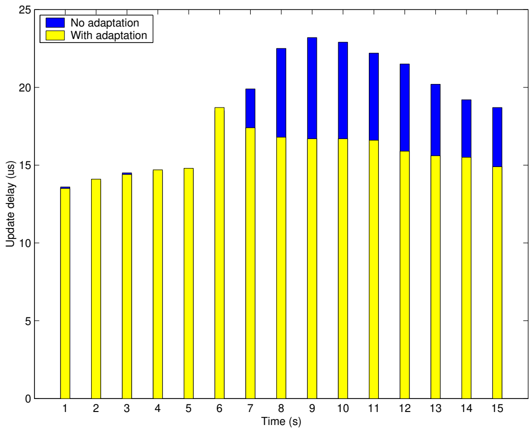

Figure 9. Performance implications of dynamic adaptation of the mirroring function based on the current operating conditions

5 Related Work

Mirroring and replication are known to be useful techniques for increasing the reliability, scalability, and availability of systems and applications. Classical CBCAST implementations [11] of replication strictly rely on message orderings, without incorporating the application-level information used for mirroring in our infrastructure. Our checkpointing implementation relies on past work on transactional processing in distributed systems [18, 19].

Slice uses mirroring techniques to reliably mirror files and thereby support failure-atomic file operations in a distributed, scalable network storage system [20]. The TACT [21] project also chooses replication as a means of increasing service availability, and it provides a middleware layer that enables application-specific consistency requirements to be enforced between replicas, which would in turn affect the experienced availability. This project explores certain consistency metrics in depth. In comparison, our goal is not to provide a general mechanism for bounding consistency, but instead, to provide a general mechanism through which these consistency requirements can be expressed in terms of application-level semantics.

The Porcupine electronic mail server [2] is a replication-based solution towards increased availability and scalability. We share similar goals, such as automated manageability and increased performance, but differ in that Porcupine does not offer possibilities for application-specific, dynamic system adaptation.

Finally, we share with other recent efforts the fact that we are exploring applications that differ from the high performance codes traditionally investigated by the cluster computing community [21, 22].

6 Conclusion and Future Work

This paper is concerned with Operational Information Systems. The scalable construction of such systems on cluster machines is facilitated by what we term *data mirroring*. Data mirroring differs from multicast or broadcast functionality offered at the network level by its use of application-level code to decide what data to mirror and how such mirroring should be performed. The resulting application-level approach to data mirroring is shown to have two benefits on cluster machines that run OIS applications: (1) the continuous processing of incoming data events and subsequent generation of outgoing events is more easily guaranteed when unusual or bursty client requests are directed to mirror sites, rather than to the cluster node(s) that perform OIS processing, and (2) mirroring can help deal with the bursty nature of such requests, by effectively parallelizing client request processing.

Experimental results show that while event mirroring results in additional overheads imposed on OIS processing, such overheads can be reduced by use of application-specific rules that determine what is being mirrored and how frequently mirroring is performed. By dynamically adapting mirroring while monitoring client request patterns, OIS execution can be made more predictable, and clients experience more timely responses.

Our future efforts include extending the mirroring infrastructure with recovery support, for both client failures, and failures of a node within the cluster server. We are also modifying the proposed architecture to include network co-processors. Towards this end, we are splitting the functionality of the ‘auxiliary’ units between a host node and a NI-resident processing unit, currently the Intel IXP1200 boards.

References

[1] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, and P. Gauthier, “Cluster-based Scalable Network Services”, In *Sixteenth ACM Symposium on Operating System Principles*, Oct.1997.

[2] Y. Saito, B. Bershad, and H.M. Levy, “Manageability, Availability, and Performance in Porcupine: A Highly Scalable Cluster-based Mail Service”, In *17th ACM Symposium on Operating System Principles*, OS Review, Volume 33, Number 5, Dec. 1999.

[3] T. Sterling, D. Becker, D. Savarese, et al., “BEOWULF: A Parallel Workstation for Scientific Computation”, In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, Aug. 1995, Volume 1, pp. 11-14.

[4] M. Beynon, T. Kurc, A. Sussman, and J. Saltz, “Optimizing Retrieval and Processing of Multi-dimensional Scientific Datasets”, In *Proceedings of the Third Merged IPPS/SPDP Symposiums*, Cancun, Mexico, May 2000.

[5] B. Plale, V. Elling, G. Eisenhauer, K. Schwan, D. King, and V. Martin, “Realizing Distributed Computational Laboratories”, *The International Journal of Parallel and Distributed Systems and Networks*, Volume 2, Number 3, 1999.

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

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

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

[9] R.E. McGrath, J. Futrelle, R. Plante, and D. Guillaume, “Digital Library Technology for Locating and Accessing

Scientific Data", *ACM Digital Libraries '99*, Aug. 1999, pp. 188-194.

[10] H. Zhu, T. Yang, Q. Zheng, D. Watson, O.H. Ibarra, and T. Smith, "Adaptive Load Sharing for Clustered Digital Library Servers", In *Proc. of Seventh Symposium on High Performance Distributed Computing (HPDC-7)*, Chicago, Aug. 1998.

[11] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast", *ACM Transactions on Computer Systems*, Volume 9, Number 3, Aug. 1991.

[12] D. Zhou and K. Schwan, "Adaptation and Specialization in Mobile Agent Environments", In *Proc of 5th Usenix Conference on Object-Oriented Technologies and Systems*, San Diego, May 1999.

[13] C. Isert and K. Schwan, "ACDS: Adapting Computational Data Streams for High Performance", In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[14] M.C. Rosu, K. Schwan, and R. Fujimoto, "Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach", In *Proc of Sixth Symposium on High Performance Distributed Computing (HPDC-6)*, Portland, Aug. 1997.

[15] C. Poellabauer, K. Schwan, R. West, I. Ganev, N. Bright and G. Losik, "Flexible User/Kernel Communication for Real-Time Applications in ELinux", In *Proc. of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop*, 2000 (in conjunction with RTSS 2000).

[16] P. Ng, "A commit protocol for checkpointing transactions", In *Proc. of the Seventh Symposium on Reliable Distributed Computing*, pp. 22–31, Columbus, Ohio, Oct 1988.

[17] D. Mosberger and T. Jin, "httperf—A Tool for Measuring Web Server Performance", *Workshop on Internet Server Performance'98*, Madison, June 1998.

[18] B. Walker et al., "The LOCUS Distributed Operating System", In *Proc. of the Ninth ACM Symposium on Operating Systems Principles*, pp. 49-70, Dec. 1983

[19] R.Haskin et.al., "Recovery Management in QuickSilver", *ACM Transactions on Computer Systems*, Feb. 1988.

[20] D. Anderson and J. Chase, "Failure-Atomic File Access in an Interposed Network Storage System", In *Proc. of Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Pittsburgh PA, Aug. 2000.

[21] H. Yu and A. Vahdat, "Design and Evaluation of a Continuous Consistency Model for Replicated Services", In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[22] Akamai Corporation. http://www.akamai.com