A Highly Available Transaction Processing System with Non-Disruptive Failure Handling

Gong Su
IBM T.J. Watson Research Center
Hawthorne, New York, USA
gongsu@us.ibm.com

Arun Iyengar
IBM T.J. Watson Research Center
Hawthorne, New York, USA
aruni@us.ibm.com

***Abstract*— We present a highly available system for environments such as stock trading, where high request rates and low latency requirements dictate that service disruption on the order of seconds in length can be unacceptable. After a node failure, our system avoids delays in processing due to detecting the failure or transferring control to a back-up node. We achieve this by using multiple primary nodes which process transactions concurrently as peers. If a primary node fails, the remaining primaries continue executing without being delayed at all by the failed primary. Nodes agree on a total ordering for processing requests with a novel low overhead wait-free algorithm that utilizes a small amount of shared memory accessible to the nodes and a simple compare-and-swap like protocol which allows the system to progress at the speed of the fastest node. We have implemented our system and show experimentally that it performs well and can transparently handle node failures without causing delays to transaction processing. The efficient implementation of our algorithm for ordering transactions is a critically important factor in achieving good performance.**

Keywords- computer-driven trading, fault tolerance, high availability, total ordering algorithm, transaction processing.

I. INTRODUCTION

Transaction-processing systems such as those for stock exchanges need to be highly available. Continuous operation in the event of failures is critically important. Failures for any length of time can cause lost business resulting in both revenue losses and a decrease in reputation. In the event that a component fails, the systems must be able to continue operating with minimal disruption.

This paper presents a highly available system for environments such as stock trading, where high request rates and low latency requirements dictate that service disruptions on the order of seconds in length can be unacceptable. A key aspect of our system is that processor failures are handled transparently without interruptions to normal service. There are no delays for failure detection or having a back-up processor take over for the failed processor because our architecture eliminates the need for both of these steps.

A standard method for making transaction processing systems highly available is to provide a primary node and at least one secondary node which can handle requests. In the event that the primary node fails, requests can be directed to a secondary node which is still functioning. This approach, which we refer to as the primary-secondary approach, has drawbacks. A key problem with the primary-secondary approach is that there can be delays of several seconds for detecting node failures during which no requests are being processed. For systems which need to be continuously responsive under high transaction rates, these delays are a significant problem. Therefore, other methods are desirable for maintaining high availability in transaction processing systems which handle high request rates and need to be continuously responsive in the presence of failures.

Our system handles failures transparently without disruptions in service. A key feature of our system is that we achieve redundancy in processing by having multiple nodes executing transactions as peers concurrently. If one node fails, the remaining ones simply continue executing. There is no need to transfer control to a secondary node after a failure because all of the nodes are already primaries. A key advantage to our approach is that after a primary failure, there is no lost time waiting for the system to recover from the failure. Other primaries simply continue executing without being slowed down by the failure of one of them.

One of the complications with our approach is that the primaries can receive requests in different orders. A key component of our system is a method for the primaries to agree upon a common order for executing transactions, known as the total ordering, without incurring significant synchronization overhead. We do this by means of a limited amount of shared memory accessible among the nodes, and a simple but efficient synchronization protocol.

The overall concept of the primary-primary approach has been proposed previously [18]. However, previous methods proposed for achieving total ordering among requests are often complex and not wait-free. In our work, we show how to achieve total ordering of requests using a relatively simple wait-free protocol. In addition, we have implemented and thoroughly tested our system using a stock trading application. A considerable effort is needed to go from ideas proposed in past papers to an efficient working system.

The key contributions of this paper include the following:

- We show how the primary-primary approach can be used for transaction processing applications such as stock trading in which the primary nodes must agree upon a common order for processing the requests.
- We have developed and implemented a new efficient algorithm for nodes in a distributed environment receiving messages in different orders to agree on a total ordering for those messages. This algorithm is used by our system to determine the order for all nodes to execute transactions and makes use of a small amount of shared memory among the nodes. The total ordering algorithm imposes little overhead and proceeds at the rate of the fastest node; it is not slowed down by slow or unresponsive nodes.
- We have implemented our approach on an IBM z990 zSeries. Experimental results show that our system achieves fast recovery from failures and good performance. Average latencies for handling

transactions are well below 10 milliseconds. The efficient total ordering algorithm is a critically important factor in achieving this performance.

II. System Architecture

Our system makes use of multiple nodes for high availability. Each node contains one or more processors. Nodes have some degree of isolation so that a failure of one node would not cause a second node to fail. For example, they run different operating systems and generally do not share memory to any significant degree. In our implementation, nodes can communicate and synchronize via a small amount of shared memory. Multiple primary nodes execute the same sequence of transactions as peers concurrently. Normally, two primary nodes would be sufficient. If failure of more than one node within a short time period is a concern, more than two primaries can be used. In the event that a primary node fails, the remaining primaries keep executing without being hindered by the failed primary.

We depict the overall primary-primary stock exchange trading architecture in Fig. 1.

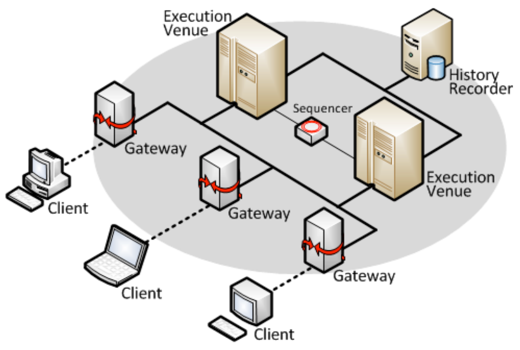


Figure 1. Primary-Primary Architecture

An electronic stock exchange, as illustrated by the shaded ellipse area in the diagram, typically consists of 3 tiers,

- *Gateways* (GW) collect buy/sell requests from clients (e.g., traders and brokers) and perform preprocessing such as validation.
- *Execution Venues* (EV) carry out the actual trading by matching incoming requests against an in-memory list of outstanding requests, which is called an order book. Each EV is implemented by a node.
- *History Recorder* (HR) is used for persistently storing the result of every trade carried out by the EVs. It is typically implemented by a file system or database management system (DBMS). It is essential to store the result of computations persistently so that information is not lost in the event of a system failure.

Each of the tiers has its own recovery mechanism, and working together, they make the entire system fault-tolerant. The main focus of this paper is on EV recovery so we only mention GW and HR recovery briefly.

GWs must persistently store every incoming trade request before they can notify clients of the reception of their requests and send the requests to the EVs. If a GW fails before persistently storing a request, the client would fail to receive an acknowledgement for the request and would thus know to resend the request. GWs typically employ DBMS in order to take advantage of DBMS fault-tolerant features. File systems can also be used and may offer better performance but fewer features. HRs, like GWs, typically also employ DBMS or file systems. DBMS failures can be minimized by using conventional techniques for highly available DBMS such as replication.

Let us now turn our attention to the fault tolerance of EVs. Today's stock exchanges typically employ a primary-secondary architecture (not what is depicted in the diagram) that, at a high level, works as follows:

- All incoming trade requests are sent to a primary EV.
- A secondary EV "eavesdrops" on the traffic between the EV and the HR in order to learn the ordering of trade requests and duplicate the primary EV's processing.
- In the event that the primary EV fails, the secondary EV initiates a recovery protocol to coordinate with the GWs and HRs and takes over as the primary.

It is evident that with a primary-secondary architecture, from the time the primary EV fails until the time the secondary EV takes over, no trade request is being processed therefore causing disruption. Due to the fact that the secondary EV needs to first detect the failure of the primary EV, plus the time it takes to complete the recovery protocol, the disruption can be on the order of seconds. In today's electronic stock exchange, EVs are typically processing trade requests at a rate of tens of thousands per second for one symbol and hundreds of thousands per second aggregated across all symbols. Thus, it is extremely costly for a stock exchange to have seconds of disruption. In fact, primary EV failure is one of the main causes of disruption in stock exchanges today. Our primary-primary architecture avoids this problem by transparently handling an EV failure without delays in normal processing.

As illustrated in the architecture diagram, the overall system, at a high level, works as follows:

- Multiple primary EVs exist. We describe how our system works for two primary EVs. It can easily be extended to handle more primary EVs.
- GWs send incoming trade requests to both primary EVs. The ordering of requests seen by the two EVs may differ.
- Both primary EVs process trade requests concurrently, using a sequencer to negotiate an ordering of trade requests agreed upon by both.
- In the event that one of the primary EV fails, the other simply continues as if nothing happened.
- EVs process the requests by matching them against the order books, and send the results to HRs.
- HRs persistently store the results and notify EVs.
- Upon receiving acknowledgements from HRs, EVs notify GWs of trade completion.
- Upon receiving acknowledgements from EVs, GWs notify the clients of trade completion.

With the primary-primary architecture, one primary EV need not act upon the failure of the other; neither need it carry out a recovery protocol (as other components in the system will detect and restart the failed primary). The only "disruption" when one primary EV fails is that it may be processing several trade requests ahead of the other so the live EV will first "catch up" in processing those trade requests before new trade requests will be processed.

A single EV can handle multiple stock symbols. While requests for a single stock symbol are processed sequentially, requests corresponding to different stock symbols can be processed concurrently using multiple threads or multiple processes. Each stock symbol on the EV has its own order book and is processed *independently*. For simplicity, our discussions and examples in Sections III and IV use one stock symbol on a pair of primary EVs. However, we have implemented multiple stock symbols on the same EV, and we provide performance results for multiple symbols in Section VI.

It is also possible to scale the system further by having multiple pairs of primary EVs. The sequencer in our scheme does not adversely affect the scalability of the stock trading system since a different sequencer can be used for a different pair of primary EVs.

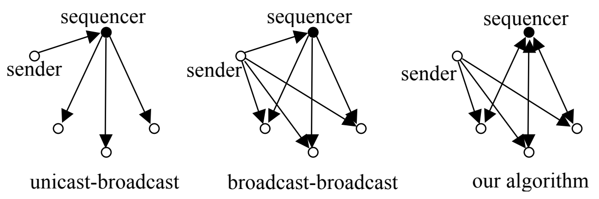
Keen readers will notice that in our primary-primary architecture, the sequencer can potentially be a single point of failure. Key to our design is to handle failover of the sequencer transparently from the EVs. We achieve this by using a fault-tolerant system for the sequencer. Our implementation uses fault-tolerant IBM hardware called the Coupling Facility [4] that runs two sequencers simultaneously and handles failover transparently in case one of them fails. This logical "single reliable sequencer" view to a pair of EVs is important. If we had exposed multiple sequencers to the EVs, the EVs would have to explicitly manage the failover of the sequencers resulting in a more complex protocol. Handling sequencer failover transparently from the EVs allows us to design a total ordering algorithm that requires simple logic in the sequencer. As a result, the sequencer is well-suited to be efficiently implemented with a highly reliable system.

A key reason the sequencer is fault-tolerant is that the algorithm it uses is relatively simple and thoroughly tested. When a more complicated component such as an EV is implemented on fault-tolerant hardware, the component would still be susceptible to software failures. It is also possible to implement the sequencer on fault-tolerant hardware such as HP NonStop (formerly Tandem) [1]. However, a key reason for using the coupling facility is the low latency communication that exists between EVs and the coupling facility.

III. The Total Ordering Algorithm

In the primary-primary architecture, all peer EVs must process incoming trade requests in exactly the same order. However, when multiple GWs multicast trade requests to multiple EVs, there is no guarantee that all EVs will receive the trade requests in the same order. Therefore, there must be a mechanism to work out a total ordering amongst all EVs.

Our total ordering algorithm is applicable not just to our stock trading system but also to other scenarios in which multiple nodes which may receive messages in different orders need to agree on a total ordering for the messages; such algorithms have been referred to as total order broadcast and multicast algorithms [2]. Our total ordering algorithm employs a centralized sequencer as a rendezvous point for peer EVs to negotiate a total ordering for processing trade requests, regardless of how each individual EV sees its local ordering of incoming trade requests. The main difference between our algorithm and the traditional unicast-broadcast and broadcast-broadcast [2] variants of fixed sequencer algorithms is that, as shown in the figure below, our algorithm involves no communication between the senders and the sequencer, only communication between the receivers and the sequencer. In an environment such as stock exchanges where the number of senders far exceeds the number of receivers, our algorithm is advantageous in terms of reducing the load on the sequencer.



Another advantage of our algorithm is that the computation of sequence numbers is performed by the receivers instead of by the sequencer. The sequencer is a shared-memory like passive entity that implements a compare-and-swap like protocol. This further reduces the complexity of the sequencer. The simplicity makes it relatively easy to both analyze and test the sequencer for correctness; it also facilitates a very efficient and fault-tolerant implementation of the sequencer.

A third advantage of our algorithm, compared to past algorithms in which the receiving nodes agree on a total ordering, is that our algorithm allows the system to progress at the speed of the fastest receiver and can proceed rapidly even in the presence of slow receivers. In many previous algorithms, multiple receivers must provide input before an ordering decision can be made [2]. While this may have advantages in some environments, the delays that these algorithms introduce are problematic for transaction processing systems with low latency requirements. We avoid these delays in our algorithm by immediately assigning a sequence number to the first correct request by a node asking for the sequence number.

The use of a small amount of shared memory for communication between the nodes results in a considerably faster sequencer than algorithms which exchange messages between nodes such as those described in [2]. We quantify the overhead imposed by our sequencer in Section VI.

The basic idea of our algorithm is simple: each EV competes to propose to the sequencer its own local ordering as the total ordering, and whichever gets to the sequencer first (the leader) wins. The losers (the followers) must shuffle their local ordering to conform to the leader. We use an example to illustrate the idea.

Assume two gateways, GW_1 and GW_2, are multicasting trade requests to two peer execution venues, EV_1 and EV_2. GW_1 multicasts trade requests p_0, p_1, p_2; and GW_2 multicasts trade requests q_0, q_1, q_2. Let's further assume that EV_1 sees the incoming trade requests as $\{q_0, q_1, p_0, p_1, q_2, p_2\}$ and EV_2 sees the incoming trade requests as $\{p_0, p_1, q_0, p_2, q_1, q_2\}$. Initially, the total ordering at the sequencer is empty $\{\}$.

(1) Assume EV_1 proposes q_0 to the sequencer before EV_2 proposes p_0. So the sequencer takes q_0 and updates its total ordering to be $\{q_0\}$. When EV_2 comes to propose p_0, the sequencer tells EV_2 that its proposal is rejected and it should process q_0 instead. So EV_2 shuffles q_0 in front of p_0, p_1, $\{q_0, p_0, p_1, p_2, q_1, q_2\}$, to conform to EV_1.

(2) After both EV_1 and EV_2 process q_0, EV_1 proposes q_1 and EV_2 proposes p_0. Assume this time EV_2 wins and the sequencer takes p_0 and updates its total ordering to be $\{q_0, p_0\}$. EV_1 is told to process p_0 instead of q_1. So EV_1 shuffles p_0 in front of q_1, $\{q_0, p_0, q_1, p_1, q_2, p_2\}$, to conform to EV_2.

Due to space constraints, we omit the rest of the steps (see [23] for a more complete description). However, it is not difficult to see that through this "compete, compare, and shuffle" process, the two EVs will arrive at exactly the same total ordering regardless of the ordering in which they received the requests locally.

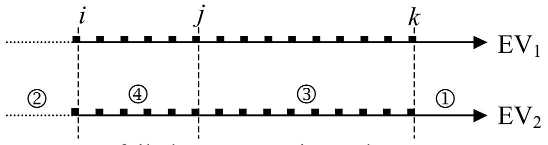
Because our algorithm allows the system to progress at the speed of the fastest EV, one EV may fall behind the leader by a significant amount. We must bound this "distance" between the leader and other EVs. Otherwise, if the leader fails, it will take too long for the followers to "catch up", thus effectively causing a disruption. We solve this problem by limiting the amount of memory the sequencer uses to store total ordering numbers assigned. Instead of storing the entire history of total ordering numbers assigned such as $[0, \infty]$, the sequencer will only store a fix-sized sliding window such as $[n, n+100]$. This means that when $n+100$ has been assigned to the leader, request n-1 will be removed. If the follower is behind the leader by more than 100 requests and tries to propose a request for n-1, the sequencer will notify the follower that it is too far behind and some action should be taken (e.g., kill the follower and restart a new one).

IV. Non-Disruptive Failover

There are two ways an EV can fail. One is what we call hard failure, where the EV completely stops processing trade requests due to hardware or software failure. Hard failure can be detected through conventional mechanisms such as heart beats and determining that the EV is not being responsive. The other is what we call soft failure, where the EV continues to process trade requests but, due to system load, etc., is falling behind the leader EV further and further. Soft failure is detected through the sliding window of requests maintained by the sequencer described at the end of the previous section.

Regardless of how an EV fails, by the nature of our primary-primary architecture, other peer EVs continue unaffected. The only effect is that there is one fewer EV competing for the total ordering via the sequencer. Therefore, as long as there is still one working EV left, failure of one or more peer EVs causes no disruption at all to the processing of trade requests.

However, this is only half of the high-availability story. When an EV fails, a new one must be started and synchronized with the working ones in order to maintain the level of availability. This process must also be done without any disruption to the working EVs. We now briefly describe how this is accomplished. To keep the description simple and without loss of generality, our system consists of one GW, two EVs (EV_1 and EV_2), and one HR.



Assume EV_2 failed at some point and we start a new one. As shown in the figure above:

(1) When EV_2 starts, EV_1 has received trade requests up to k, and has processed trade requests up to j. Therefore, EV_2 can receive all trade requests after k.

(2) Periodically, EV_1 takes a checkpoint of its entire order book and sends it to the HR. Assume the last checkpoint EV_1 took included trade requests up to i. By asking HR for the latest checkpoint, EV_2 can immediately recover all trades occurred up to i.

(3) For each trade request after j processed by EV_1, a persistent storage request is sent to HR. The reply from HR, which includes a copy of the original trade request, is multicasted to both EV_1 and EV_2. By "listening to" the reply from HR, EV_2 can recover trade requests between j and k.

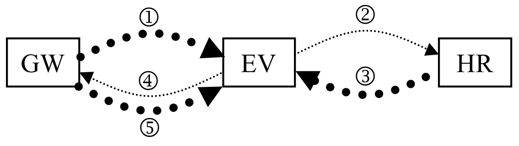
(4) By asking HR for the persisted trade requests between i and j, EV_2 can finally recover all missing trade requests.

It is not difficult to see that the entire process causes no disruption to EV_1. Note that the four steps above are how missing trades are recovered in parts and "stitched together". They are *not* the order in which the missing trades are processed. All four steps actually happen concurrently. EV_2 can start processing trades from i once it receives the checkpointed information. Recovered missing trades that are out of sequence are queued.

V. Prototype Design and Implementation

To verify the feasibility of our architecture, we have designed and implemented a prototype on the IBM zSeries eServer mainframe [3]. The reason for choosing the zSeries eServer is that the function of our sequencer is readily available with a special hardware called Cross-System Coupling Facility (XCF) [4], which allows high performance data sharing across different logical partitions (LPARs) of a single eServer or across multiple eServers. The key is to maintain the "single reliable sequencer" view to the EVs.

Communications among GW, EV, and HR are through LLM (Low Latency Messaging) [5], which is an IBM product that provides reliable and ordered multicast and unicast messaging services. The message flow is depicted in the figure below:



(1) Trade request from GW to EV, multicast
(2) Persistent storage request from EV to HR, unicast
(3) Persistent storage ack from HR to EV, multicast
(4) Trade completion from EV to GW, unicast
(5) Completion ack from GW to EV, multicast

Essentially, the list services allow peer EVs to implement a "compare-and-swap" protocol to support the total ordering algorithm. The protocol is simple and only requires one trip to XCF.

In our primary-primary architecture, the HR will receive duplicated requests from the primary EVs to persistently store the trading information. This is easily handled by the monotonically increasing sequence numbers. The HR records the highest sequence number that has been persistently stored. It ignores any requests having a smaller or equal sequence number. The overhead introduced by this approach is very small.

VI. Experimental Results

Our experiments are conducted on an IBM zSeries eServer mainframe model z990 [3] with a total of 32 1.2GHz CPUs and 256GB memory. Each GW, EV, and HR runs in its own LPAR with dedicated CPUs and memory. LPAR is a way to virtualize hardware resources such that each partition functions as if it were an independent physical machine while transparently sharing hardware resources. In our experiments, each GW and HR has 2 CPUs and 2GB memory, and each EV has 4 CPUs and 4GB memory. All the LPARs are running z/OS version 1.8, IBM's proprietary mainframe OS. Connectivity among the tiers is through HiperSockets [6], which is a direct memory-to-memory copy between two LPARs that involves no actual network interface and provides much better performance than Gigabit Ethernet. The link between EV and XCF is a special fiber optic link called Integrated Cluster Bus (ICB) with speed up to 2GB per second [3]. Our test bed is depicted in the diagram below.

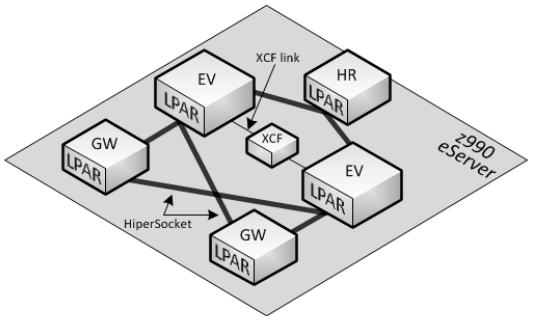


Figure 2. Prototype Testbed

A. High Availability Characteristics

We first present the non-disruptive availability results that show the strength of our primary-primary architecture. A checkpoint of EV in-memory state information is sent to the HR periodically. The checkpoint interval is controllable via a tunable parameter. In these experiments, the EVs take a checkpoint after every 1024 requests are processed. There is a tradeoff between checkpoint overhead and recovery time after a failure (presented next). The more frequently we checkpoint, the higher the overhead but the shorter the recovery time. In our tests, we observed that the typical size of the EV in-memory state is about several hundred kilobytes. At the chosen checkpoint interval of every 1024 requests, the checkpoint overhead in our tests is negligible.

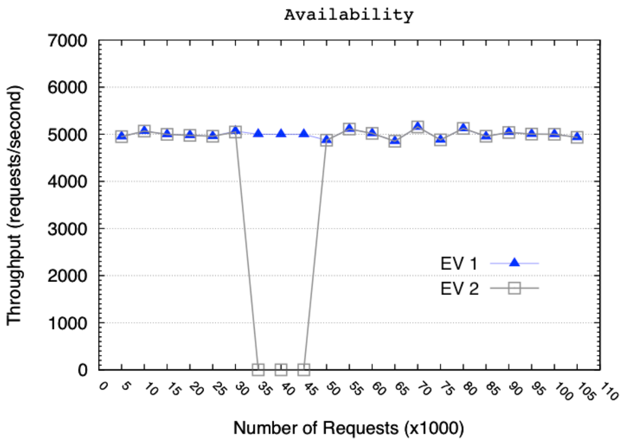


Figure 3. Availability test

Fig. 3 corresponds to one GW sending trade requests to two EVs at a throughput of roughly 5000 requests per second. Each request is to either buy or sell a certain number of shares of a stock symbol. Half of the requests are buy orders, while the other half are sell orders. After about 30,000 requests, EV_2 fails. After about 50,000 requests, EV_2 is restarted; it then synchronizes with EV_1 and resumes processing as before. We can see that during the entire period, EV_1 continues to process the trade requests at roughly 5000 requests per second as if nothing happened. The throughput for EV_1 and EV_2 closely overlap except during the failure and recovery period for EV_2.

We now show how long it takes for a newly started EV to synchronize with a live EV non-disruptively in the middle of trade processing. Synchronization time is the duration from when a new EV is started until it recovers the states of all symbols (as described in section IV) and becomes fully operational. Table I shows the synchronization time of a GW sending trade requests of a single stock symbol to two EVs at different throughput. We can see that the synchronization times for all the cases are under 5 milliseconds.

TABLE I. Sync Time vs Throughput

Throughput (msgs/sec)	100	500	1000	9000
Sync time (ms)	3.50	3.65	4.30	4.61

Fig. 4(a) shows the synchronization time of one GW sending trade requests of 10 stock symbols to two EVs at throughputs of 1000 per second. Synchronization times for each of the 10 symbols are under 30 milliseconds; the total synchronization time for all 10 symbols is under 35 milliseconds. Since different symbols are recovered concurrently, the total synchronization time is only slightly more than the individual symbol synchronization time.

Fig. 4(b) shows the synchronization time of 4 GWs sending trade requests of 10 symbols to two EVs at a throughput of 1000 messages per second. We see that there is much more variation in the synchronization time from symbol to symbol, ranging from 30 to 121 milliseconds. The variation is due to the fact that with 4 GWs, the chance of symbols coming to each EV with a different ordering is much higher. The total synchronization time for all 10 symbols is about 150 milliseconds. We remind the readers that for all the cases, there is no disruption to the live EV during the synchronization.

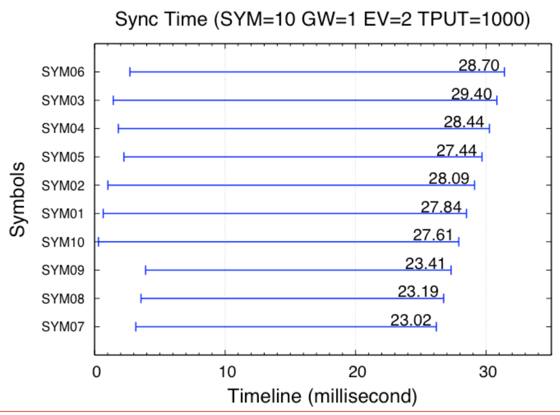


Figure 4(a). Sync time, 10 symbols, 1 GW, 1000 rqsts/s

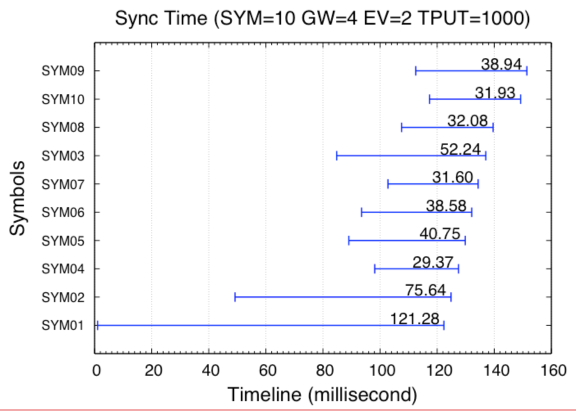


Figure 4(b). Sync time, 10 symbols, 4 GWs, 1000 rqsts/s

Another important property of our system (or any primary-primary system) is the progress difference between the EVs when processing requests. Even though there is no disruption when one of the EV fails, if the failed EV is the leader, the follower EV does need to make up the gap between the two before new requests can be processed. Therefore, the gap determines the slight delay before new requests can be processed when a leader EV fails. Obviously, this delay cannot be too large. Otherwise, the delay essentially causes a disruption.

We measure the processing speed difference between two EVs at different throughput and the results are shown in Fig. 5. The tests are done with trade results being persistently stored on the HR to be realistic. The figure shows the difference of the total ordering sequence number of processed requests on the two EVs sampled every 5 seconds during a period of roughly 2 minutes. A positive bar means EV_1 is ahead of EV_2 by that many processed requests, while a negative bar means the opposite. When no bar appears at a time mark, it means the two EVs are processing a request with exactly the same sequence number. Note that the two EVs take turns being the leader. This is a nice property since this means that only 50% of the time there will be a delay in processing new requests when an EV fails, assuming each EV has an equal chance to be a leader. We also observe that at throughput 9000 requests/second, the largest gap is about 21 requests at the 100^{th} second. This means that if the leader failed, the delay incurred by the follower's making up the gap of 21 requests would be roughly 2.3 milliseconds.

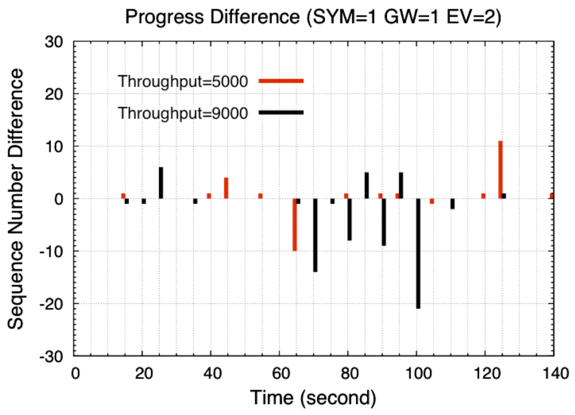


Figure 5. Progress difference between two EVs

We also present the maximum and average gap in requests processed between two EVs over a long period of time to give a sense of how synchronized the two EVs are over time. Table II shows the numbers for a period of 30 minutes with a sampling interval of 5 seconds. Max. Gap and Avg. Gap are the maximum and average total ordering sequence number difference between the two EVs. Max. Delay and Avg. Delay are the time needed for a follower to make up the gap in case the leader fails. Note that for throughput of 5000, Max. Delay and Avg. Delay are obtained by dividing Max. Gap and Avg. Gap by 9000, not 5000, respectively. This is because 9000 is the maximum processing rate of the EVs, while 5000 is the incoming rate.

TABLE II. MAXIMUM AND AVERAGE GAP OVER TIME

Throughput (msgs/sec)	Max. Gap (msgs)	Max. Delay (ms)	Avg. Gap (msgs)	Avg. Delay (ms)
5000	57	6.3	3	0.3
9000	81	9.0	7	0.8

The numbers in Fig. 5 and Table II indicate that the two EVs stay closely synchronized, a key reason why failure of one EV causes no disruption in processing the requests.

B. *Throughput and Latency Characteristics*

Since end-to-end latency within the system (simply referred to as latency hereafter), which is from the time when a GW sends a trade request to the EVs to the time when the GW receives a trade completion notification from the EV, is one of the key performance measurements, we also present latency related measurements. These measurements show that our prototype can meet performance standards required by stock exchanges. Typically, today's stock exchanges require that the latency be less than ten milliseconds.

We first measure the overhead of our total ordering algorithm (sequencer) which uses the XCF. We compare the latency at different throughput with 10 symbols, 1 GW, and 2 EVs. In our implementation, messages from a single GW will be sent to EVs in the same order (although our architecture is capable of handling situations when this is not the case). Thus, with only one GW, agreeing on a total ordering is not necessary so we can turn off the sequencer (implemented by the XCF) to make the comparison. As shown in Fig. 6, the XCF overhead is at most 1.38 milliseconds at 9000 with persistent storage. The efficiency of our total ordering algorithm and low communication latency between EVs and the XCF are critically important factors for achieving good performance.

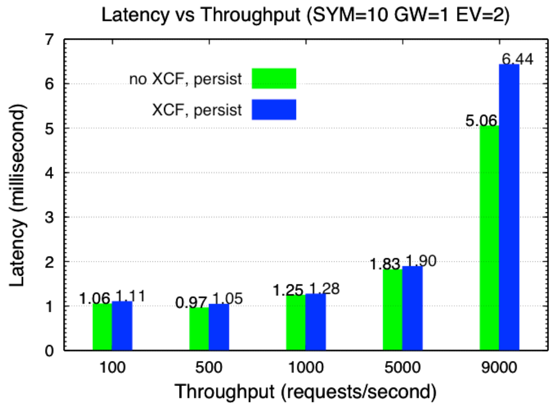


Figure 6. Latency with persistent storage, 10 symbols

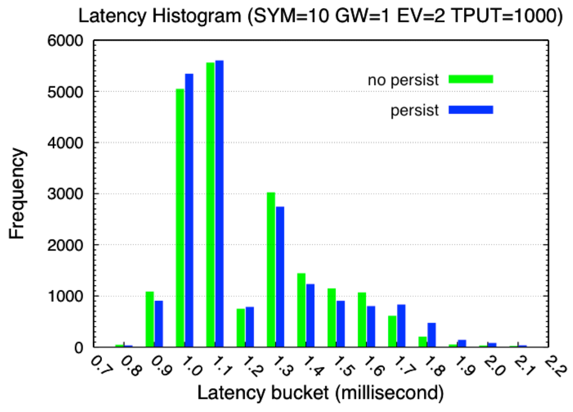


Figure 7. Latency histogram, 10 symbols

Next we plot the latency distribution to check and make sure that the average latency numbers in Fig. 6 are indeed representative. A latency histogram over 20,000 requests for 1 GW sending trade requests for ten symbol to two EVs at 1000 requests per second is shown in Fig. 7. Note that we only show a histogram for one of the 10 symbols as the others are quite similar. The bars marked "persist" correspond to storing the results persistently. The results show that the majority of the latency numbers are 1.0 and 1.1 milliseconds with an average between 1.25 and 1.28 milliseconds (not shown).

Finally, we measure the latency with 2 GWs sending trade requests for 10 symbols at different throughputs. Note that with 2 GWs, total ordering must be turned on so there are no tests for "no XCF". The results are shown in Fig. 8.

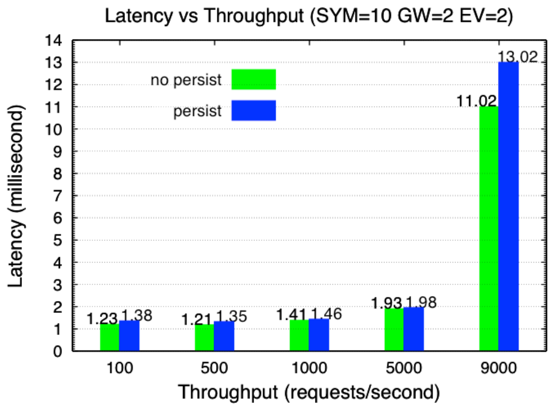


Figure 8. Latency with 2 GWs, 10 symbols

The latency numbers with two GWs are fairly similar to those with one GW, except at the throughput 9000 requests per second. At this throughput and 10 symbols, latency increased from 6.44 to 13.02 milliseconds with persistent storage. These numbers reflect the fact that with increasing throughput and number of symbols, the chance of the two EVs receiving messages from the two GWs in a different order increases; therefore, processing time increases due to the need for the EVs to shuffle their queues.

VII. Related Work

Schneider [18] describes achieving fault-tolerant state machines via replication; conceptually, this is a primary-primary approach. However, in order to achieve total ordering, the three algorithms proposed are of a general nature. Logical and real clock based algorithms require clock stability tests, while replica-generated unique identifiers require a second disseminating phase. None of the three algorithms are wait-free.

Several high availability cluster solutions exist in which a back-up node can take over for a primary node after the primary node has been determined to have failed. Examples include HACMP from IBM [7], the Microsoft Cluster Service [8], and HA-Linux [9]. There can be considerable delays in processing resulting from both detecting the node failure and transferring control to the back-up node. Our primary-primary architecture avoids these delays.

The Swiss Exchange system in the 1996-98 timeframe is discussed in [10]. This exchange uses a primary-secondary architecture unlike our primary-primary architecture.

Considerable work has been done in the area of Byzantine fault tolerance [11, 12]. Our system is not prone to the same types of failures that Byzantine fault-tolerant systems are prone to. As a result, our system incurs significantly less overhead.

There have been several algorithms proposed for agreeing on a total ordering of messages received by different nodes in a distributed environment. A comprehensive survey of these algorithms is contained in [2]. Our approach has conceptual similarity to the "Destinations Agreement Algorithms" summarized in this paper [13, 14, 15, 16]. A key difference of our algorithm is that nodes communicate using a small amount of shared memory resulting in faster communication than the previous algorithms using message passing. Another key difference is that our sequencing algorithm can progress at the speed of the fastest receiving node. The previous algorithms generally require a consensus to be formed among multiple nodes which means that slow nodes can delay the process.

Sinfonia [20] describes a service that allows distributed applications to share data in a fault-tolerant, scalable, and consistent manner, hiding much of the complexity of designing and implementing distributed protocols with message passing. On top of Sinfonia, distributed services such as SinfoniaGCS, a group communication service with total ordering, can be built. Even though SinfoniaGCS is designed to be simpler and more efficient than the more complex protocols surveyed in [2], it is still substantially more complex than our total ordering algorithm.

Herlihy [19] (and others) have described various algorithms for constructing universal consensus wait-free data structures such as queues, heaps, stacks, etc., using primitives such as compare-and-swap. A wait-free total ordering algorithm can indeed be designed using these data structures. However, we do not need these data structures for our application. A simple compare-and-swap is all that is needed to achieve total ordering for our stock trading application. This makes our protocol simple and efficient.

Narasimhan [21] describes the Eternal system that provides transparent fault tolerance for CORBA applications through active or passive replication. It employs the Totem [22] for its total ordering messaging service. Totem is substantially more complex than our total ordering algorithm due to its distributed nature.

A Web-based financial trading system designed to handle bundle orders is described in [17]. It does not address how to handle high availability and recover from failures.

VIII. Conclusion

This paper has presented a highly available system for stock trading. Our system uses multiple primary nodes so that if one primary node fails, the remaining one (s) can keep executing without disruption. Experimental results show that our system can handle failure of a primary node without affecting the performance of the other primary node. We also implemented a recovery algorithm which allows a failed primary to be restarted and to catch up with a running primary relatively quickly.

Our system uses a new algorithm for determining a common order for processing requests. Our algorithm adds little overhead and is a critical component in achieving good performance. Average latencies for processing requests in our system are significantly below 10 milliseconds.

References

[1] HP Integrity NonStop Computing, Hewlett-Packard, http://en.wikipedia.org/wiki/Tandem_Computers

[2] X. Defago, A. Schiper, and P. Urban, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey", ACM Computing Surveys, Vol. 36, No. 4, December 2004, pp. 372-421.

[3] IBM Redbook, "IBM eServer zSeries 990 Technical Guide", May 2004.
http://www.redbooks.ibm.com/abstracts/sg246947.html?Open

[4] IBM Redbook, "z/OS Parallel Sysplex Configuration Overview", September 2006.
http://www.redbooks.ibm.com/abstracts/sg246485.html?Open

[5] LLM WebSphere MQ Low Latency Messaging, http://www.ibm.com/software/integration/wmq/llm

[6] IBM Redbook, "HiperSockets Implementation Guide", March 2007.
http://www.redbooks.ibm.com/abstracts/sg246816.html?Open

[7] HACMP High Availability Cluster Multiprocessing Best Practices, IBM Corporation, January 2008,
ftp://ftp.software.ibm.com/common/ssi/sa/wh/n/psw03025gben/PSW03025GBEN.PDF

[8] W. Vogels et al, "The Design and Architecture of the Microsoft Cluster Service", Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing, 1998.

[9] Linux-HA, http://www.linux-ha.org/

[10] X. Defago, K. Mazouni, A. Schiper, "Highly Available Trading System: Experiments with CORBA", Proceedings of Middleware '98.

[11] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, Vol. 4 no. 3, July 1982.

[12] M. Castro, B. Liskov, "Practical Byzantine Fault Tolerance", Proceedings of OSDI 1999, New Orleans, February 1999

[13] K. Birman, T. Joseph, "Reliable communication in the presence of failures", ACM Transactions on Computer Systems, vol. 5 no. 1, February 1987.

[14] S.-W Luan, V. D. Gligor, "A fault-tolerant protocol for atomic broadcast", IEEE Transactions on Parallel and Distributed Systems, vol. 1 no. 3, July 1990.

[15] T. D. Chandra, S. Toueg, "Unreliable failure detectors for reliable distributed systems", Journal of the ACM, vol. 43 no. 2.

[16] L. T. Rodrigues, M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems", Proceedings of ICDCS 2000.

[17] M. Fan, J. Stallaert, A. Whinton, "A Web-Based Financial Trading System", IEEE Computer, April 1999.

[18] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, vol. 22, no. 4, December 1990, pp. 299-319.

[19] M. Herlihy, "Wait-free synchronization", ACM Transactions on Programming Languages and Systems, vol. 13, no. 1, 1991, pp. 124-149.

[20] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems", Proceedings of SOSP 2007.

[21] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects", Proceedings of DSN 2001.

[22] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System", Communications of the ACM, vol. 39, no. 4, April 1996, pp. 54-63.

[23] G. Su and A. Iyengar, "A Highly Available Transaction Processing System with Non-Disruptive Failure Handling", IBM Research Report RC24960, 2010.