RC24960 (W1003-054) March 17, 2010
Computer Science

IBM Research Report

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

Gong Su, Arun Iyengar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

Gong Su
IBM T.J. Watson Research Center
gongsu@us.ibm.com

Arun Iyengar
IBM T.J. Watson Research Center
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 algorithm that utilizes a small amount of shared memory accessible to the nodes and a simple compare-and-swap protocol which allows the system to progress at the speed of the fastest node. We have implemented our system on an IBM z990 zSeries eServer mainframe and show experimentally that our system can transparently handle node failures without causing delays to transaction processing.

1. 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 at least two drawbacks for environments such as stock trading. The first is that stock trading requests must be directed to specific nodes due to the fact that the nodes have local in-memory state information typically not shared between the primary and secondary for handling specific transactions. For example, a primary node handling trades for IBM stocks would have information in memory specifically related to IBM stocks. If a buy or sell order for IBM stock is directed to a secondary node, the secondary node would not have the proper state information to efficiently process the order. The primary node should store enough information persistently to allow stock trading for IBM to continue on another node should it fail. However, the overhead for the secondary node to obtain the necessary state information from persistent storage would cause delays in processing trades for IBM stock which are not acceptable. The second 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 the primaries.

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 which our system has access to between the nodes, and a simple but efficient synchronization protocol.

The key contributions of this paper include the following:

- We have developed a new architecture for highly available transaction processing system which does not incur delays when a node fails.
- We have implemented our system on an IBM z990 zSeries. Experimental results show that our system achieves fast recovery from failures and good performance.
- We have developed a new 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 between 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.

2. 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 known as a coupling facility.

For environments such as stock trading, response times have to be extremely fast. Therefore, state information needed to perform transaction processing is cached in the main memories of nodes handling transactions. A key drawback of the primary-secondary approach of having a back-up node take over in case the primary node fails is that the back-up node will not have the necessary state information in memory in order to restart processing right away. There are also delays in detecting failures. A common method for detecting failures is to periodically exchange heart beat messages between nodes and listen for failed responses. It is generally not feasible to set the timeout period before a node is declared failed to too small an interval (e.g. less than several seconds) due to the risk of erroneously declaring a functioning node down. This means that it often takes several seconds to detect a failure. The delays that would be incurred in detecting the failure of a primary node and getting a secondary node up and running by obtaining the necessary state information from persistent storage are thus often too high using this conventional high availability approach.

For this reason, it is essential to have at least two nodes with updated in-memory data structures for handling orders for the stock. That way, if one of the nodes fails, the other node will still be functioning and can continue handling trades for the stock.

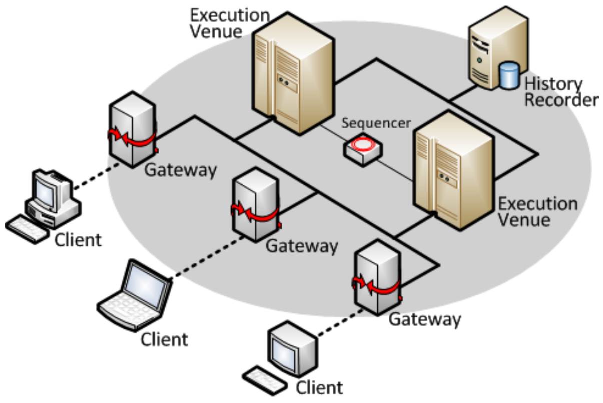
One way to achieve high availability would be to have a primary node handling requests for a stock in a certain order and to have the primary node send the ordered sequence of requests that it is processing to a secondary node. The secondary node then executes the transactions in the same order as the primary node but a step or two behind the primary node. The secondary node would avoid performing many updates to persistent storage already performed by the primary processor since the whole reason for the secondary processor executing transactions is to keep its main memory updated.

While this approach eliminates some of the overhead of simply having a cold standby taking over for the primary, it still incurs some overhead for both detecting the failed primary node and handling the failover from the primary node to the secondary node. As we mentioned previously, detecting the failed primary node can take several seconds. The secondary node also needs to figure out exactly where the primary node failed in order to continue processing at exactly the right place. If the failover procedure is not carefully implemented, the secondary node could either repeat processing the primary node has already done or leave out some of the processing the primary node performed before failing; either of these two scenarios results in incorrect behavior.

Our system avoids the problems of both detecting failures and transferring control from a failed node to a back-up node by having multiple primary nodes executing 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 now describe our architecture in more detail.

2.1 Primary-Primary Architecture

We depict the overall primary-primary stock exchange trading architecture in the diagram below.



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) are the heart of the stock exchange. They 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.

A typical stock trading transaction involves the following steps:

- GWs receive trade requests from clients, persistently store the requests, and send the requests to EVs. Different EVs may receive requests GWs in different orders. Therefore, there is the need to agree on a total ordering for the requests.
- EVs agree on a total ordering for the requests by communicating with the sequencer. In our implementation, the sequencer includes a limited amount of shared memory that EVs can use for communication.
- EVs process the requests by matching them against in-memory state known as 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.

Each of the tiers sports its own recovery mechanism and working together, they make the entire system fault tolerant. We first briefly describe the recovery mechanism of GW and HR, and then in more detail the recovery mechanism of EV since that is one of the main focus areas of this paper.

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, it can simply ask the client to resend it. 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. In order to improve performance, HRs may use "group commit" instead of committing every single trade individually. However, this raises the possibility that a group of trade results can be lost if a HR fails. This danger is guarded against by requiring that: (1) a HR cannot notify an EV of trade completion until all trade results in the group have been committed; and (2) an EV cannot notify a GW of trade completion until it has been notified by the HR. So in the event that a HR fails, the three tiers can coordinate to have the GWs replay those trades for which a trade completion was not received.

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

- All incoming trade requests are sent to a primary EV, which also acts as the sequencer.
- 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 EVs.
- All incoming trade requests are sent to both primary EVs.
- 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.

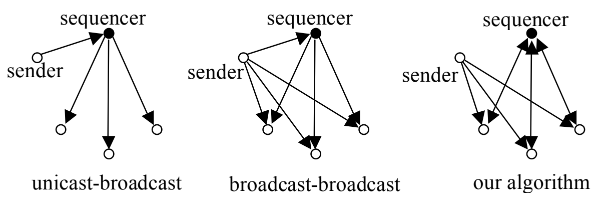
With the primary-primary architecture, one primary EV need not detect the failure of the other; neither need it carry out a recovery protocol. The only "disruption" when one primary EV fails is that it may be processing several trade requests ahead of the other so the live EV will first "catch up" in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

Keen readers will point out 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; it is also possible to use highly fault-tolerant hardware such as HP NonStop (formerly Tandem) [1]. This "single reliable sequencer" view to 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.

3. 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 peer 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.



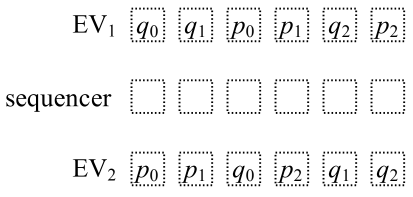
Another advantage of our algorithm is that the logic of generating the next sequence number is in the receivers rather than in the sequencer. As we can see in the detailed description of the algorithm below, the sequencer in our algorithm is essentially a shared-memory like passive entity that implements a compare-and-swap like protocol. This further reduces the complexity of the sequencer, which lends itself well to 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 6.

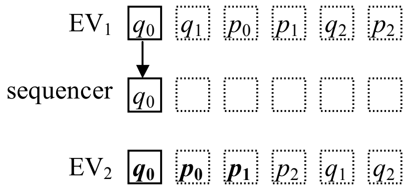
The basic idea of our algorithm is simple: each EV competes to propose to the sequencer its own local ordering as the total ordering, and whichever gets to the sequencer first (the leader) wins. The losers (the followers) must shuffle their local ordering to conform to the leader. We use an example to 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. So initially, the local ordering at EV_1 and EV_2, and the total ordering at the sequencer are as follows (dashed box indicates received but not yet processed trade requests):

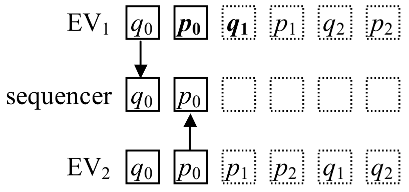


We now show how EV_1 and EV_2 compete to negotiate a total ordering through the sequencer. At each step of the example, we will give the state of the local ordering at EV_1 and EV_2, and the total ordering at the sequencer.

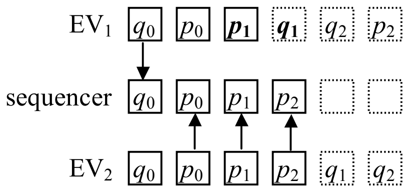
1. When EV_1 receives q_0, it proposes to the sequencer that it would like q_0 to be processed at the 1st position of the total ordering. Similarly, when EV_2 receives p_0, it proposes to the sequencer that it would like p_0 to be processed at the 1st position of the total ordering. Assume EV_1 gets to the sequencer first and wins (indicated by the arrowed line from EV_1 to the sequencer). So the sequencer takes q_0 at its 1st position and, when EV_2 comes to propose p_0, 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 (shown in bold font) to conform to EV_1 (solid box indicates processed trade requests):



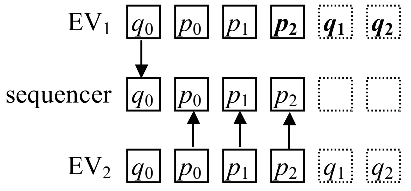
2. After both EV_1 and EV_2 process q_0, EV_1 proposes q_1 and EV_2 proposes p_0. Assume this time EV_2 wins and the sequencer takes p_0 at its 2nd position and tells EV_1 to process p_0 instead of q_1. So EV_1 shuffles p_0 in front of q_1 to conform to EV_2:



3. Assume that, after processing p_0, EV_2 wins in proposing both p_1 and p_2 for the 3rd and 4th position of the total ordering. So when EV_1 proposes q_1, it is told to process p_1 instead and has to shuffle p_1 in front of q_1 to conform to EV_2:



4. When EV_1 proposes q_1 after processing p_1, it is told to process p_2 instead and has to shuffle p_2 in front of q_1, q_2 to conform to EV_2:



We can see that at this point, the local ordering on EV_1 and EV_2 are exactly the same. For processing q_1 and q_2, it doesn't really matter which EV wins the proposal. So the total ordering negotiated through the sequencer is: $q_0, p_0, p_1, p_2, q_1, q_2$.

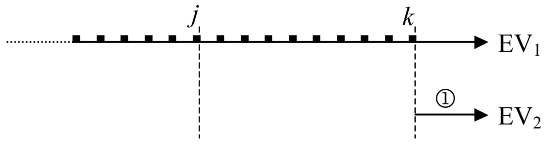
Because our algorithm allows the system to progress at the speed of the fastest EV, one EV may fall behind the leader by a significant amount. We must bound this "distance" between the leader and other EVs. Otherwise, if the leader dies, it will take too long for the followers to "catch up", thus effectively 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, ∞], 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).

4. Non-Disruptive Failover

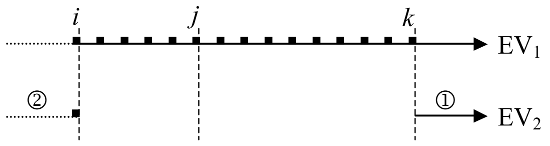
There are two ways an EV can fail. One is what we call hard failure, where the EV completely stops processing trade requests due to hardware or software failure. The other is what we call soft failure, where the EV continues to process trade requests but, due to system load, etc., is falling behind the leader EV further and further.

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.

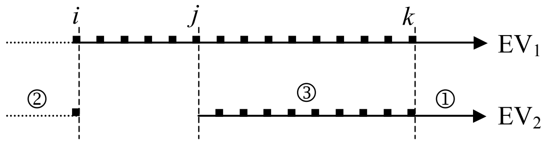
However, this is only half of the high-availability story. When an EV fails, a new one must be started and synchronized with the working ones in order to maintain the level of availability. This process must also be done without any disruption to the working EVs. We now describe how this is accomplished. 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.



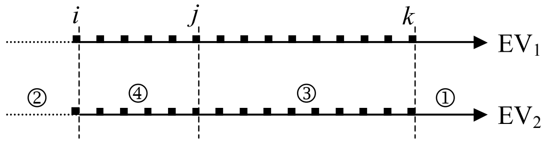
(1) Assume, as shown in the figure above, when EV_2 starts, EV_1 has received trade requests up to k, and has processed trade requests up to j. Therefore, EV_2 can receive all trade requests after k, but needs to recover all trade requests up to k.



(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, as shown in the figure above. By asking HR for the latest checkpoint, EV_2 can immediately recover all trades up to i. Now it needs to recover trade requests between i and k.



(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. Therefore, by "listening to" the reply from HR, EV_2 can recover trade requests between j and k, as shown in the figure above. The only missing trade requests now are those between i and j.



(4) By asking HR for the persistently stored trade requests between i and j, EV_2 can finally recover all missing trade requests, as shown in the figure above. It's 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.

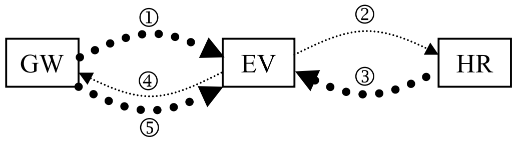
5. Prototype Design and Implementation

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 prototype consists of the following three functional components needed for a stock exchange:

- GW, which generates trade requests for one or more stock symbols;
- EV, which executes stock trading by maintaining in-memory state known as an order book for each stock symbol and matching incoming trade requests against the order book;
- HR, which persistently stores information for all trades to a file system.

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 (thin dashed lines indicate unicast messages, and thick dashed lines indicate multicast messages).



(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

The functions of our sequencer are implemented through the list services provided by XCF, which allow applications to share data organized in a list structure. List entries can have ID, key, etc., and be kept in sorted order by certain attributes. For an EV to propose a total ordering number for a trade request, it simply asks XCF to create a list entry with [*ID=total ordering number*, *key=trade request*]. Using the sample example in section 3,

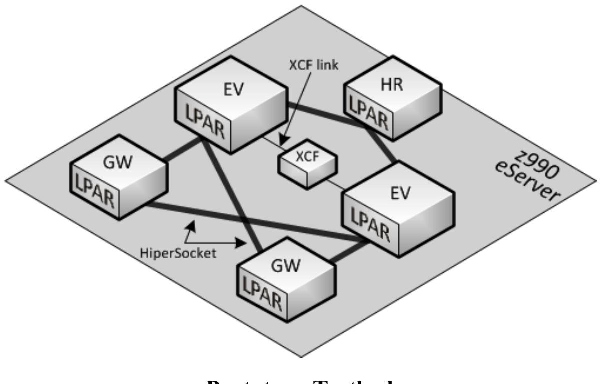
- EV_1 attempts to create an entry [*ID=0*, *key=*q_0]
- EV_2 attempts to create an entry [*ID=0*, *key=*p_0]
- EV_1 gets to XCF first so entry [*ID=0*, *key=*q_0] is created successfully
- EV_2 gets to XCF next and is informed an entry with *ID=0* already exist and its current *key=*q_0

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 the next section, we will present the experimental results of our prototype.

6. 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 testbed is depicted in the diagram below. Note that in our experiments, without loss of generality, we did not use separate clients but rather have the GWs generate trades directly.



Prototype Testbed

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.

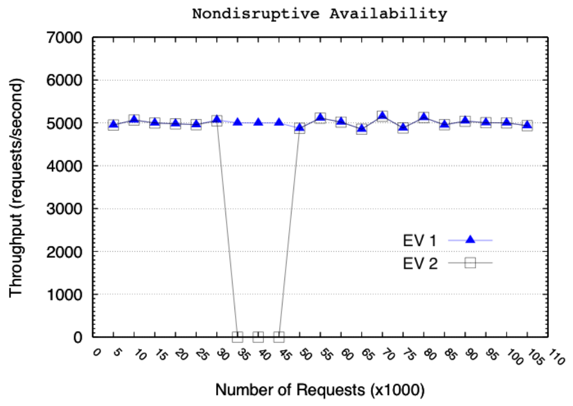


Figure 6-1 Non-disruptive availability

Figure 6-1 shows 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.

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. Figure 6-2 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.

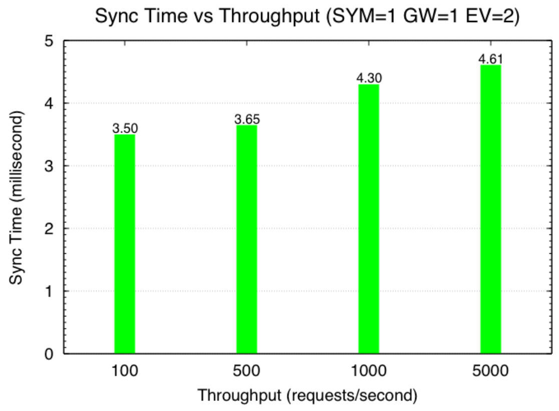


Figure 6-2 Sync time, 1 symbol

Figure 6-3(a) and 6-3(b) show the synchronization time of a GW sending trade requests of 10 stock symbols to two EVs at throughputs of 1000 and 9000 requests per second. For a throughput of 1000 requests 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. For a throughput of 9000 requests per second case, synchronization times for each of the 10 symbols are under 700 milliseconds; the total synchronization time for all 10 symbols is about 800 milliseconds.

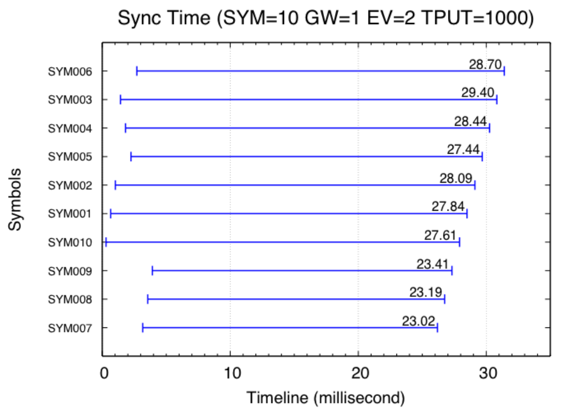


Figure 6-3(a) Sync Time, 10 symbols, 1 GW, 1000 rqsts/s

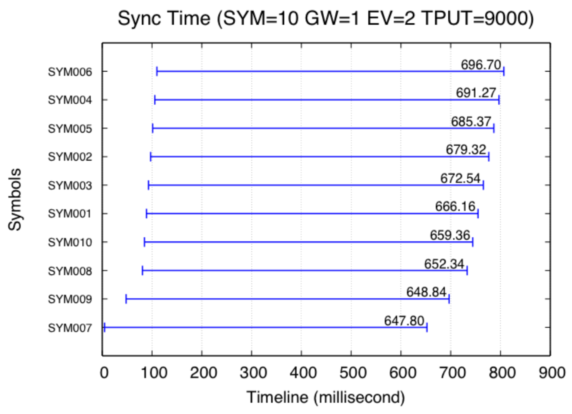


Figure 6-3(b) Sync time, 10 symbols, 1 GW, 9000 rqsts/s

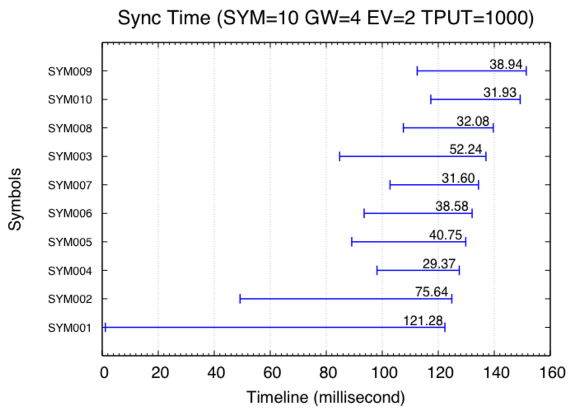


Figure 6-3(c) Sync time, 10 symbols, 4 GWs, 1000 rqsts/s

Figure 6-3(c) 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 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.

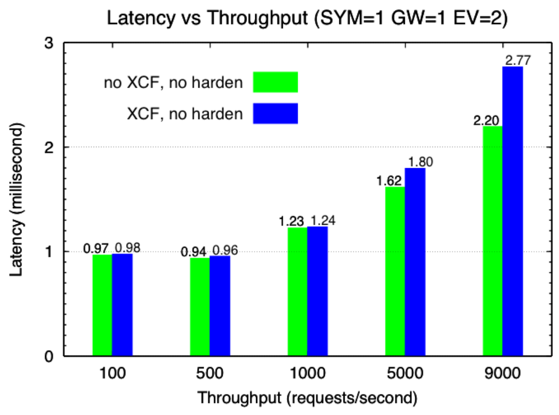


Figure 6-4(a) Latency w/o persistent storage, 1 symbol

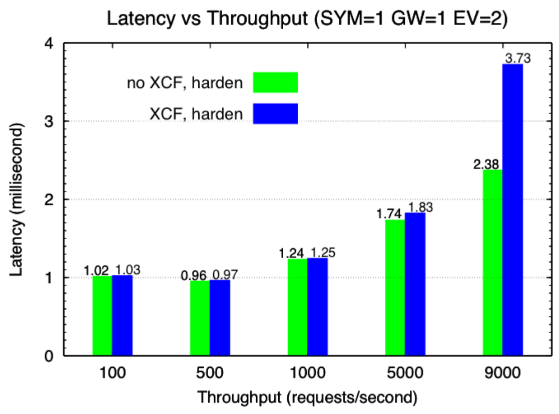


Figure 6-4(b) Latency with persistent storage, 1 symbol

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 a variety of latency related measurements which show that our prototype can meet performance standards required by stock exchanges. Typically, today's stock exchanges require that the latency to be single-digit milliseconds.

We first measure the overhead of our total ordering algorithm (sequencer) which uses the XCF. We compare the latency at different throughput with 1 symbol, 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. In Figure 6-4(a) we make the comparison without persistently storing the trade results by HR to further isolate the XCF overhead. In Figure 6-4(b) we make the comparison while persistently storing the trade results by HR to show that this does not affect the XCF overhead. We can see that in both figures, our total ordering algorithm going through XCF adds very little overhead, at most 0.35 milliseconds (at 9000 requests per second with persistent storage).

A single EV can handle throughputs up to 9000 requests per second before the response time becomes unacceptably high. z/OS has a component call USS (Unix System Services) which implements a certified UNIX (XPG4 UNIX95) environment that makes porting programs written for UNIX to z/OS much easier. In fact, the reliable multicast messaging service LLM we used in our prototype is written for UNIX and not supported by z/OS. So we have ported it to USS, and our prototype runs on top of LLM in the USS environment. This convenience, however, comes at a performance cost. Due to context switching overhead resulting from LLM, 9000 requests per second is the highest throughput we can achieve. It is possible to add more processors to an EV in order to get higher throughput rates.

We then measure the scaling behavior of our total ordering algorithm in terms of the number of stock symbols. We repeat the same measurements in Figure 6-4(a) and 6-4(b) with 10 symbols, one GW, and two EVs; the results are shown in Figure 6-5(a) and 6-5(b). As shown in the figures, the XCF overhead increased marginally (typically fewer than 100 microseconds) for all throughputs except 9000, at which point the overhead is 1.23 milliseconds without persistent storage and 1.38 milliseconds with persistent storage.

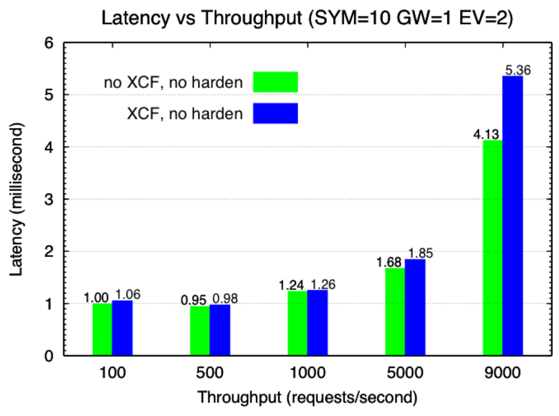


Figure 6-5(a) Latency w/o persistent storage, 10 symbols

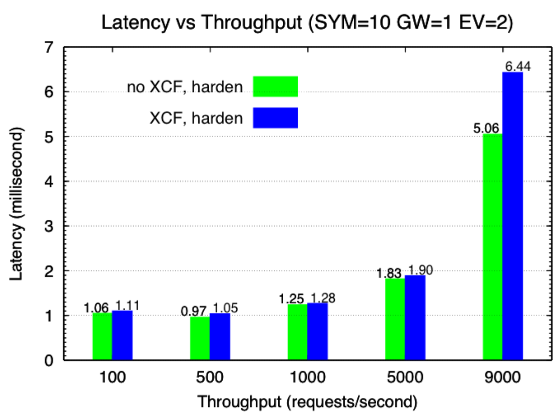


Figure 6-5(b) Latency with persistent storage, 10 symbols

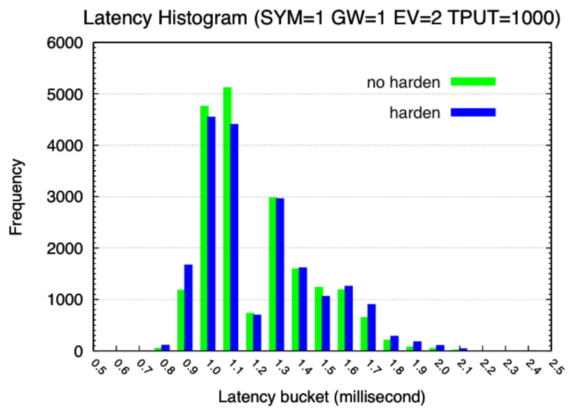


Figure 6-6(a) Latency histogram, 1 symbol

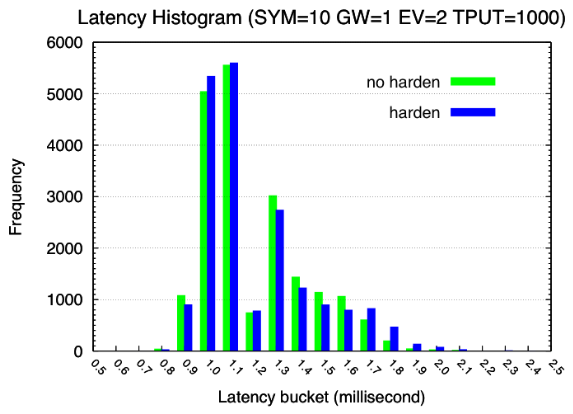


Figure 6-6(b) Latency histogram, 10 symbols

Next we plot the latency distribution for one particular configuration to check and make sure that the average latency numbers in figures 6-4(a) through 6-5(b) are indeed representative. A latency histogram over 20,000 requests for 1 GW sending trade requests for one symbol to two EVs at 1000 requests per second is shown in Figure 6-6(a), and the same measurement for 10 symbols is shown in Figure 6-6(b). Note that in Figure 6-6(b), we only show a histogram for one of the 10 symbols as the others are quite similar. For both cases, 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 one or ten symbols at different throughputs. Note that with two GWs, total ordering must be turned on so there are no measurements for “no XCF”. The results are shown in Figure 6-7(a) and 6-7(b). The bars marked “harden” correspond to storing the results persistently.

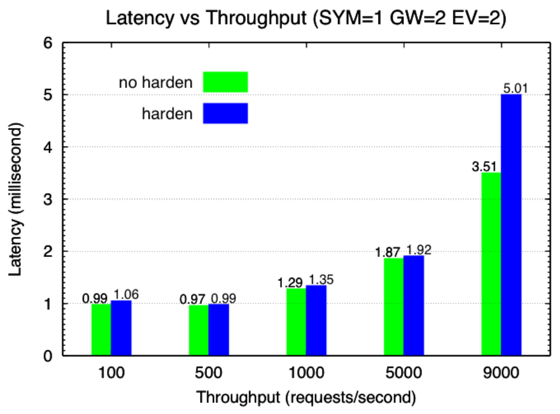


Figure 6-7(a) Latency with 2 GWs, 1 symbol

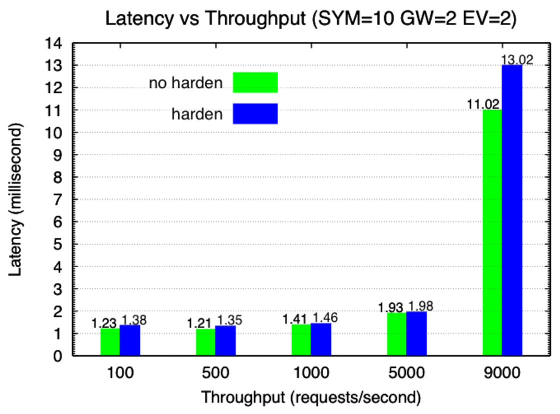


Figure 6-7(b) Latency with 2 GWs, 10 symbols

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 one symbol, latency increased from 2.77 to 3.51 milliseconds without persistent storage (27% increase), and from 3.73 to 5.01 milliseconds with persistent storage (34% increase); with 10 symbols, latency increased from 5.36 to 11.02 milliseconds without persistent storage (106% increase), and from 6.44 to 13.02 milliseconds with persistent storage (102% increase). 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.

7. Related Work

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 sequencing algorithm is that nodes communicate using a small amount of shared memory resulting in faster communication than the previous algorithms which use 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.

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

8. Acknowledgement

The authors would like to thank Paul Dantzig and Francis Parr for their helpful discussions and constructive comments regarding this work.

9. 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.