Volatility Management of High Frequency Trading Environments

Matthew Brook, Craig Sharp, Gary Ushaw, William Blewitt, Graham Morgan
School of Computing Science, Newcastle University, UK
(m.j.brook1, craig.sharp, gary.ushaw, w.g.blewitt, graham.morgan)@newcastle.ac.uk>

***Abstract*—High frequency trading (HFT) environments provide technologies that enable algorithmic trading within automated marketplaces. The most prominent example of an HFT environment is within equity trading, where many millions of trades are achieved at a high volume to gain a reasonable cumulative profit. Such environments rely on low latency/high performance technologies to allow trades to react in a timely manner to market volatility. However, sometimes the volatility of the market goes beyond what supporting infrastructure can allow, resulting in erroneous behaviour of the marketplace. In this paper we tackle the problem of managing market volatility to limit erroneous market behaviour. Our approach is unique in that it is non-dependent on the trading environment itself and self-regulates based only on trading frequency and contention. We demonstrate our results and show that by managing trade injection rates and contention of shared state the volatility of HFT environments can be managed appropriately and in an automated manner.**

Keywords—replication; eventual consistency; high frequency trading;

I. INTRODUCTION

In distributed systems research there are few application domains as sensitive to timeliness as *high frequency trading* (HFT) environments. The requirement to trade at intervals measured in a fraction of a second need to be satisfied to attain a fractional profit. Over time the accumulation of such trades can afford a substantial profit. The equation is simple; increase the frequency of trades increases profits.

Issues arise in all equity markets when contention for equities rises. We use the term contention to refer to the degree to which simultaneous trades are requested on the same equity. Such scenarios in human driven trades reflect volatility in the market and could lead to undesirable outcomes (significant drops, possibly economically unwarranted). In a HFT scenario the result could be disastrous due to the lack of human intervention coupled with the trading frequency. Research into algorithmic trading expends great effort in integrating safety measures into algorithms to avoid such scenarios.

In this paper we consider an alternative approach to managing volatility in HFT environments. We propose pushing responsibility for managing volatility to the supporting software infrastructure on which HFT algorithms execute. We use contention management techniques that exploit the semantics of HFT trading (patterns) combined with forcibly varying the trading rates of clients. This removes volatility and provides a safety net for algorithmic trading, alleviating such algorithms from having to encode their own, sometimes very complex, solutions to avoid unchecked volatility.

Our solution at first glance appears unintuitive (back off trades and lower the trading frequency). However, we show that a supporting platform that is semantically aware of its application domain can actually improve successful trade throughput for HFT environments. We achieve this by utilizing techniques often associated with high performance, scalable replication schemes used in cloud computing and server side technology (e.g., search engines, cloud resource management). With appropriate client side extensions and increased reach of state replication, our scheme provides an eventually consistent solution for HFT deployment.

As we are combining two distinct areas of research (HFT environments and scalable replication schemes) we afford a generous background and related work section. This is to provide the reader with all required terminology and understanding for the remainder of the paper. Section 3 presents our approach in general terms. Section 4 presents a reproducible description of our approach for experimental purposes. We also show, via simulation, how successful our approach is in achieving high trading success rates. Section 5 presents our concluding remarks and discusses our future work in this area.

II. BACKGROUND AND RELATED WORK

In this section we describe how existing scalable replication schemes in modern day data retrieval systems may be adaptable for use within HFT environments. To do this we first describe HFT environments in terms of their computational requirements and technological shortcomings. We then describe scalable data replication schemes and the benefits they bring to a number of different application domains. We finish this section with our argument suggesting how scalable replication schemes may be tailored to bring a cost efficient, yet improved, solution to the technical demands of HFT environments.

A. High Frequency Trading

Equity markets are traditionally the domains of human directed trading. However, with the advent of computers, attempts were made to encode human style trading within automated procedures. From the 1970s the introduction of computerization has increasingly provided the opportunity for algorithmic trading to occur with ever reducing dependence on human-in-the-loop intervention [1].

In the 1980s there were a number of advances in computerization that provided the first recognizable fully automated trading environments (e.g., pair trading utilizing statistical arbitrage and convergence trading strategy [2]). Such devices were considered low-risk, yet yielded profits [1]. However, there is debate, academically, as to how much such systems contribute to undesirable market conditions (e.g., bubbles, crashes, flash crashes). Although an interesting academic discussion, it is beyond the scope of this paper to consider the appropriateness of algorithmic trading. We are only interested in the fact that such systems exist and require software support.

Algorithmic trading software can be constructed based on pattern recognition technology and may utilise modern day artificial intelligence techniques (e.g., genetic algorithms, neural networks). The basic premise is that patterns of trading, possibly quite complex, can inform future trades in a positive/profitable manner. Trends can be predicted (with a degree of probability) for a period of time into the future. Such prediction models have higher probabilities of success if their time horizon is shorter. However, the shorter the time horizon between trades (buy an equity then sell) the lower the profits per trade. Other trading mechanisms may also be employed (e.g., short selling), but the principals related to profits remains the same.

Recently the low latency high performance capabilities of modern computer systems have allowed algorithmic trading systems to trade at high frequencies (HFT). HFT allows fractional profits to be made on trades occurring many times a second. This suits algorithmic trading as time horizons are low while the volume of trades is high enough to generate a suitable profit over a reasonable time. Reports suggest that HFT is widespread; accounting for approximately 35% of UK and 70% of US equity trades [3].

There are a number of techniques used to construct algorithmic trading solutions. However, all solutions are embodied within the algorithms employed to determine trading bounds on equity value. That is, the supporting technology simply affords the high frequency required to allow different styles of algorithms to succeed in an HFT environment. In this respect, a detrition of the underlying technology in terms of performance would hinder the success of the algorithms themselves.

There are numerous articles published with respect to the algorithmic nature of automated trading of equities. However, such literature stands alone from the supporting technology itself. The only assumption made by the literature in terms of software/hardware platform is an ability to attain so many trades per-second at the platform level. When technology deteriorates in performance the supported algorithms may underperform as latency of trades increase. As such, HFT environments are expensive as the latest technology is employed to provide robust high performance solutions to ensure a minimum latency is guaranteed.

B. Scalable Networked Replication Schemes

Scalability in data retrieval systems implemented across computer networks primarily depends on replication strategies. In essence, data is replicated and it is these replicas with which clients interact. In many instances, such data is co-located with the clients themselves. This lowers access times for such data as no or little network latency is involved. Even if the data is not geographically close to a client overall performance will improve as replicas afford many more opportunities for data access (there are more of them) than a single, non-replicated data item. This is because to maintain correctness (in terms of consistency) of a data item concurrent accesses need to be regulated in some way [4]. As concurrency control has no scalable solution for general data access strategies then minimising its usage is a necessity if a scalable system is desired.

The replication of data brings about higher availability of a system; as replicas fail or are taken down for maintenance other replicas may continue to satisfy the demands of clients. Flexibility may also be introduced by allowing replicas to be added to the system or taken away depending on the load of the system. This makes economic sense as when load is light (e.g., low client numbers or infrequent client requests) power consumption can be saved as machines are removed. This is similar to the commodity cost model found in cloud computing. Most data retrieval systems for cloud-based applications are based on data replication policies for increased performance [5].

The problem with data replication schemes is the degree of consistency they exhibit. If replicas were required to be always viewed the same by all clients (mutually consistent views) then we would have a pessimistic approach. Unfortunately, pessimistic approaches are non-scalable, as all replicas would need to agree what their state is (to make sure that if two clients simultaneously request state from replicas representing the same data item that they get the same value). Therefore, in all scalable solutions to data replication the optimistic approach is favoured.

Optimistic replication schemes allow parts of a distributed application to progress in the presence of transient unreachability of one or more geographically separated sub-systems. This property also ensures that optimistic replication may scale to a large number of replicas due to the lightweight synchronisation requirement compared to that found in their pessimistic counterparts.

Optimistic replication suits those applications where inconsistencies occur rarely. This may be due to the static nature of data (i.e., elements of data do not change – reads are much more common than writes) or the compartmented nature of the data itself (e.g., interference across replicas is low). As such, optimistic replication has become a popular solution in

many well known distributed applications that share these traits (e.g., search engines, resource discovery).

Eventual consistency [6, 7] is the term used to describe the property that guarantees the convergence of replicated states within an optimistic replication scheme. In principal, all replicas will converge, as past inconsistencies will be reconciled at some point during future execution. It follows that an absence of writes coupled with a window of full connectivity across replicas is required to ensure all replicas become mutually consistent. This is a version of the consensus problem with well-understood provable properties, providing a solid theoretical basis for optimistic replication schemes based on eventually consistent protocols.

An early example of a practical solution using a basic optimistic replication strategy over the Internet can be found in DNS [7]. However, it was the popularity of mobile networks (transient connectivity) and the need to access unreliable data repositories with low latencies (availability, scalability) that brought about increased research activity in optimistic replication schemes.

Two of the most popular, and well-known, optimistic replication techniques were developed to satisfy industry demands. Dynamo [8] was developed to provide Amazon with availability support for its server infrastructure. As Amazon consists of many thousands of components, failures occur continuously and therefore consistency is sacrificed to achieve availability. Cassandra [9] was developed to allow scalable management of user messages on facebook. Similar to Dynamo, Cassandra is expected to run on thousands of nodes where consistency can be sacrificed to aid availability. These recent works find their foundations in earlier academic research. We now describe two of these earlier systems to highlight the requirements tackled that led to modern day server-side optimistic replication solutions.

Bayou [10] is an optimistic replication scheme designed with the goal of satisfying consistency requirements of shared state that resides across mobile devices. In essence, devices can access and update their local copy of shared state with synchronisation occurring once connectivity between other devices has been re-established. During synchronisation there is a chance that some of the local actions enacted on shared state cannot be honoured. In such circumstances devices attempt an alternate request (determined by the application programmer). The challenging aspect of Bayou is the ability to propagate synchronisation across peer-to-peer mobile networks epidemically (no central server).

The IceCube system [11, 12] goes beyond Bayou in creating a framework within which an optimal reconciliation of replica states may occur in the context of application dependencies. That is, minimise those local actions that cannot be honoured. In essence, the application developer identifies constraints between actions that act upon shared state that in turn are used to create a single re-playable schedule of actions. By using constraints the developer has the ability to inform what type of reconciliation would be most appropriate before reconciliation occurs (the optimal schedule). Such a schedule may be replayed at replicas to not only arrive at a consistent state, but a state that best satisfies the intent of the actions occurring in the overall system. This reduces the number of those actions that could not be honoured. Irreconcilable actions are dropped from the schedule and the system must handle these exceptions in an application dependent manner.

C. Dsicussion

In the literature the supporting software/hardware of HFT environments is not considered nor described. Articles within this domain concentrate on the algorithms themselves. However, we do know they reside on propriety, expensive platforms providing latency guarantees. Therefore, one assumes consistency of state is achieved via dedicated shared access mechanisms operating strict concurrency control. We can assume this as the published algorithms rely on accuracy of achieving trades as indicated by the underlying technology.

A low cost solution to provisioning a HFT environment could be derived using optimistic replication techniques. The performance would be adequate (given current cloud based techniques that scale to millions of simultaneous users). Unfortunately, the system would suffer, as inconsistency would result in past trades not honored due to out-of-date quotes on equities. This is compounded because a brief amount of time will pass where such trades would be assumed honored by a client. Only when the underlying system informs that the trade was unsuccessful would a client realize that there is a problem.

Existing scalable optimistic replication schemes providing eventually consistent data retrieval systems are not suitable for HFT support. For example, Bayou, IceCube, would require compensation; Dynamo and Cassandra would not care. However, if we extend such systems with two additional properties there may be a practical solution:

- *Semantic contention management* – utilize the patterns of trading exhibited by HFT environments to minimize number of inconsistencies present in the system.
- *Client injection rate balancing* – alter the rate at which clients enact trades.

We can exploit the prediction properties within HFT environments to limit inconsistencies in the replicated state. If we know which equities a client may access in the near future we can supply a client with updated state before they may need them. In addition, if we have preemptively provided a client with such state we can prevent others from accessing that state in a manner that will result in inconsistency. This process of determining who should and should not access shared state is termed contention management and is commonly found when a shared resource is oversubscribed (e.g., network protocols, process scheduling, transactional memory – [13, 14, 15]). The ability to exploit application behavior in such a decision brings us the term semantic contention management.

Providing semantic contention management alone would not provide an adequate solution. Clients in modern day HFT environments base their frequency of trades on available

resources of the platform and the latency of the network. As the contention manager actively resolves heightened contention for shared resources the overall system will decrease in performance, resulting in higher failed trades. If clients continue to proceed at too high a frequency then the overall system will continue to deteriorate to a level of unacceptable failure in terms of the number of lost trades. Therefore, clients need to slow their trading frequency to achieve an equilibrium that allows contention to be resolved adequately.

Varying client injection rates will, at first thought, reduce overall performance. However, this is not the case as the successfulness of the overall system in terms of successful trades per-second should rise (less failure).

D. Contribution

This paper proposes a semantically aware approach for managing the volatility commonly found in HFT environments. We achieve this by varying client-trading frequencies while exploiting the semantic properties inherent in algorithmic trading. We summaries our contribution as follows in terms of novelty and usefulness:

- *Self-balancing technologies for HFTs* – In the literature there are no publications relating to HFT software support techniques. We propose a supporting technology that can balance the required success rate of trades against available resources during run-time. Even existing systems may benefit, as there appears to be little or no consideration of what happens if the resource allocation is dynamic during runtime.
- *Client promotion of eventually consistent systems* – We balance, for the first time, client injection rates against inconsistencies in optimistic replication schemes. Thus, we provide an avenue for a system to determine injection rates to lower inconsistencies.
- *HFT technological requirements* – The considerations of HFT technological support is a topic yet to be described in detail. Here we describe HFT requirements in the context of distributed systems research. For the first time we identify such requirements and determine how they require specialist consideration in terms of existing technology (i.e., optimistic replication schemes).

III. Approach

In this section we present a description of our approach in terms of client side and server side responsibilities. A detailed description of our initial implementation of semantic contention management is provided in [16]. However, [16] lacks the variability of client side injection and does not account for changing trends within the application domain (two key features required for HFT support). We present the description in general terms, as there are numerous options for specific implementations. The purpose here is to describe the combination of techniques to achieve the supporting infrastructure. The section on evaluation details the implementation details chosen by us.

A. Architectural Overview

We assume network clients represent traders and are measured in the hundreds. Clients are geographically distinct from a server that maintains the correct and consistent state of a trading market. Each client maintains a local replica of the trading market. A client enacts trades on their local trading market copy; this affords a highly responsive environment for clients. Trades enacted locally at client replicas are propagated to the server. On receiving such trades the server either updates the server side trading market based on a client's trade or informs the client that the trade was not possible due to inconsistencies. An inconsistency occurs if a client enacted a trade on an equity in their local database that was out-of-date compared to that of the same equity in the server replica.

Clients enact trades on their local replica without waiting for server notifications. Clients may enact many trades on a local replica before been informed by the server that one of its past trades was invalid. When such a scenario occurs the client rolls back to the failed trade and voids all subsequent trades enacted after the failed trade. The client then continues trading from this point. The reason for voiding trades is because successful trades inform the algorithmic process. If the assumption was incorrect (trade was good when it actually failed) one can assume subsequent trades should have been different in nature. By rolling back we avoid this scenario and do not negate the benefits of the algorithm governing client trading.

We assume that all client replicas are duplicates of the server before any trades occur within the system. This is an initialization issue and can be achieved trivially in existing software systems. The server replica represents the authoritative state of the trading market and trades successfully enacted at the server side cannot be undone.

Client replicas are updated as and when a failed trade is recognized at the server. When a server informs a client that a trade was unsuccessful the server also informs the client of what parameters are associated to an equity's valuation status. This allows a client to update its own replica and then proceed with a more up-to-date view of an equity's state after rollback has occurred.

An assumption made in the overall progression of the system is that clients are only informed of failed trades. Consequently, this has the same effect as indicating that all proceeding trades to the failed trade were successful. This allows clients to realize that such trades will never be rolled back, allowing resources reserved for accommodating such an event to be released.

B. Semantic Contention Management

The system is extended at the server side to handle semantic contention management and exploit the patterns associated with algorithmic trading. Clients play no role in such contention management; they simply receive notification of failed trades.

We assume a basic model of patterns inherent within a series of trades. We don't actually tune the system to best reflect the advanced prediction mechanisms in any one specific algorithm. We do this to separate our system from a particular implementation (as there are many).

The server maintains a directed graph representing trade progression over time. The graph contains a vertex per-equity. Edges between vertices represent the likelihood of a client trading a series of equities. In figure 1 we represent a world with only 5 equities and show the likelihood of progression between successful trades. For example, a client has a higher rating of progressing from B to E than from B to D. The exclusion of an arc does not mean one trade can't be traded after another (e.g., C following A), it simply means that the patterns modeled rarely allow it.

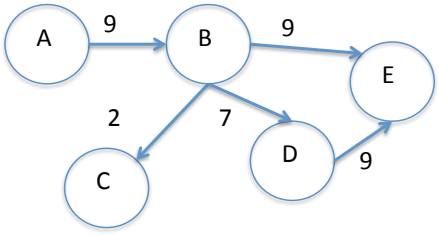

Figure 1 – Directed graph with probabilities

For clarity we only consider single trades, however, the model can be extended to multiple trades. Multiple graphs can be aggregated or a single graph may represent trading patterns of block trades. Although maintaining a graph in this manner may appear expensive in terms of resources, the size of trading markets in terms of equities is small compared to data repositories common in modern day computing environments (e.g., search engines, cloud computing).

When notifying a client of a failed trade the server not only reports back to the client the up-to-date value of the failed trade, but the neighboring values of the failed trade in the graph. This allows the pre-emptive updating of the client replica in the hope that the client will act predictably. The detailed algorithm for achieving this is presented in [16].

In addition to maintaining such a graph the server also maintains a volatility measure for each vertex. This indicates how many times per second a particular equity is traded (its popularity). When a server recognizes a failed trade it waits to inform the client for a period of time based on the popularity of the equity. The more popular the equity is, the longer the wait. This prevents multiple clients from simultaneously attempting the same trade in quick repetition and is a classic back-off approach common in shared resource access strategies.

The graph can be dynamic or static in nature. That is, a static graph will remain unchanged over time whereas the weightings of edges may change over time in a dynamic graph to represent the changing popularities of trading progressions. As trading patterns do change subtly over time then the graph should represent this. Therefore, we advocate a dynamic graph that recognizes changes in patterns and adapts to them. This will be based on threshold values and require periodic evaluation of historical data to determine which edges and vertices are popular over recent trades.

C. *Client Injection Rate Variation*

The server side contention management scheme ensures equilibrium of update rates for popular equities by utilizing a back-off strategy. However, if clients are left to continue trading at the same frequency their rollbacks will increase as their trades are backed-off. Therefore, to balance the equilibrium of updates achieved at the server side we need to adjust the injection rates of trades across clients. In essence, we wish to slow down or speed up trading frequencies on a per-client basis to match the ability of the server to achieve successful trades. If this is not achieved the ability of the prediction present within algorithmic trading will be wasted as all those trades rolled-back would have to be ignored.

To vary injection rates on a per-client basis a scheme relating to rollback threshold may be employed. Given that a client knows the degree of rollback, this can be used when calculating injection rate. As rollback is relative to the original injection rate (too high and rollback will be high, too low and rollback will be low), then the new injection rate must be relative to the previous inject rate.

Calculating client back off is not a new paradigm and has been researched extensively and showed to work effectively for different application domains (e.g., [17]). In our scenario, any exponential back-off scheme may be employed with a degree of success. As the model of execution occurs primarily in a single application domain we propose the adoption of a simple back-off approach. However, it may be that algorithmic trading properties do lend themselves to particular back-off strategies. Unfortunately, it is not possible to explore such scenarios within the scope of this paper.

D. *Balancing act*

Successful implementation of our approach should result in a scenario where client injection rates and server side back-off balance over time (become fairly static). The expectation is that the overall system automatically tunes itself to achieve the injection rate that best reflects the ability of the underlying technology to achieve the optimum number of successful trades (manage contention). The semantic nature of the contention manager should provide an opportunity to exploit patterns of trading to ensure successful trades are higher than simply providing a static platform or variable injection rates alone.

IV. EVALUATION

In this section we describe the evaluation of our approach. We first describe the actual techniques we used to accommodate semantic contention management and client injection rate back-off. We then describe the parameters associated to the experiments. Results are then presented with explanatory text.

A. *Techniques*

The techniques used in our approach are relatively straightforward. We present here an overview to enable reproducibility of our results.

1) *Server side back-off*

Each equity (vertex in the graph) has a FIFO queue associated to it within which failed trades are placed. Items are removed from the queue based on the volatility of the equity. Volatility is measured as the number of failed trades over a period of time (e.g., second). The higher the volatility value the lower the frequency items are taken from the queue. Items taken from the queue represent a failed message, with such information returned back to a client to inform them that their trade was unsuccessful. Hence, when volatility is high the frequency of informing clients of failed trades lowers, reducing the rate of reattempted trades providing a reduction in contention.

2) Client injection rates

When clients receive a message informing them of a failed trade they rollback all trades they have carried out since the failed trade, resetting the appropriate values in their local replica of the trading market. A threshold for rollback indicates at which point injection rates are lowered. For example, we may deem rollbacks of over 20 trades to require lowering of the injection rate (rollback threshold of 20). Lowering of a client's injection rate is via some pre-determined algorithm. Rollback threshold and the pre-determined algorithm may vary depending on market circumstances and the supporting technology capabilities. In our approach we halved the injection rate when rollback threshold was breached.

We expect failed trades to exist, even in a balanced system. We increment the injection rate whenever a failed trade is reported that does not breach the rollback threshold. This is important as we acknowledge that inconsistency will persist to some extent within the system. There is a need to raise the injection rate to ensure the injection rate can achieve balance with the semantic contention management scheme. Backing off then slowly increasing the threshold is one way of achieving this.

3) Dynamic graphs

To satisfy the changing trading patterns of clients over time the graph representing the probabilistic progression of client trades may alter during runtime. Therefore, the graph is modified as and when trades occur.

A client trading two equities in succession will create an edge between such equities in the graph. Unfortunately, if we were to continue in this manner we may well end up with a fully connected graph, unnecessarily increasing the load in the overall system and not allowing distinguishable patterns to be observed. Therefore, to allow for the deletion of edges as well as the creation of edges we associate a popularity value to an edge. If clients progress along an edge they increase its popularity value by one.

Identifying the popularity of an edge provides a scheme within which the most popular edges will maintain the highest values. Over time this will reflect edge popularity within the graph. Unfortunately, we actually want to reflect the current popularity of an edge, not the complete historic view. Therefore, the graph must be pruned as and when required. Periodically, the graph is searched for values below a certain threshold. This results in the removal of their edges. After pruning the graph all remaining edges are reset to the value 0.

Periodic pruning and resetting of popularity values provides our scheme with a basic reconfiguration process to more appropriately reflect current trading patterns. We acknowledge that this process of reconfiguration will incur a performance cost relative to the number of data items (vertices) and edges present in the graph. The decision on the time between periodic reconfiguration will be based on a number of factors: (i) the relative performance cost of the reconfiguration; (ii) the number of client requests within the period. If the number of requests is low but reconfiguration too frequent then edges may be removed that are still popular. Therefore, we dynamically base our reconfiguration timings on changes in load.

An interesting observation of reconfiguration is it also presents a window of opportunity to alter the data items present. If this was a typical trading market, then new equities may be introduced as graph reconfiguration occurs. This has two benefits: (i) introduction of items may well alter the dynamics of client trading patterns and so waiting for reconfiguration would not result in unnecessary overhead as graph values change significantly; (ii) one can apply some application level analysis on the effect new equites have on existing equities items.

B. Experiments

We now describe the experiments carried out to determine the appropriateness of our approach. Our main requirement is to determine if the system balances, and the semantic backoff combined with variation in client injection rates secures more successful trades than if no semantic backoff was present.

1) Environmental Parameters

We produced a discrete event simulation using the SimJava [18] framework to generate the results presented here. We modeled a realistic representation of the platform, as one would expect in n-tier architectures (clients, load balancer, application servers and database). All communication is performed through message passing. Trade requests are sent from client processes to the local replica then forwarded to the load balancer process that in turn forwards the requests to the appropriate application server. To service a trade request, an application server will communicate with the database as required to determine whether the trade is possible.

In each experiment, we simulated 100 clients, three application servers and one database. A graph with 500 vertices is randomly generated along with client accesses being pre-generated based on the layout of the graph. The initial graph layout contains vertices with and without edges. Over the duration of the experiment, reconfiguration of the graph will occur changing the structure to match that of the client accesses. Reconfiguration occurs every 30 seconds with a popularity threshold of one. This period was chosen after experimentation and was found to provide a balance between

graph consistency and reconfiguration overhead. As HFT is a rather rapidly unfolding scenario we found typical values too high for a realistic simulation. The rollback threshold was chosen to allow for any edges that were used at least once to appear in the graph after the next reconfiguration.

Message delay between clients and application servers (load balancer) was simulated as a random variable with a normal distribution between 1 - 25 milliseconds. This is slightly higher than one would expect in HFT environments. However, as the purpose here is to determine the suitability of a low cost platform for HFT support these figures were considered typical of cheap commodity cloud hardware. A fixed injection rate of 100 messages per second at each client was chosen as an initial figure but is modified over time given the desired rollback threshold. Processing overheads at the server were set for each trade request as 20 microseconds and for graph reconfiguration as 100 microseconds. At the database, read and writes were modelled as 3 and 6 microseconds respectively.

Three evaluations were performed: (1) the average length a client is required to rollback when receiving a message from the server; (2) a measurement of throughput for successful trades per second (measured at the server side database); and (3) the total number of unsuccessful trades. For each experiment the threshold for the injection rate modification is increased in intervals of ten. Each client performs 200 trade requests and leaves the system. The results presented in each graph were produced from the average of 10 executions.

C. *Results*

Figure 2 shows the graph representing successful trades per second across the whole system. When the rollback threshold initially rises there is an improvement in performance. When the rollback threshold is too low clients are persistently rolling back too frequently to allow the client injection rates to rise sufficiently high to achieve their optimum values. Each line displays an optimum point at which injection rates reach an appropriate value for maximizing successful trades.

The approach including semantic contention management (back-off) (exploiting predictable trading behaviors) clearly outperforms the approach where no back-off management is present. Furthermore, the optimum injection rate for semantic back off is higher; indicating that overall throughput of the system is significantly better. An interesting observation is that optimum injection rates are much more significant when semantic back off is present, indicating that the two schemes are working collaboratively in a self-tuning manner.

Figure 3 shows the average rollback of clients and clearly shows that when semantic contention management is present higher rollback thresholds result in longer rollbacks. This is to be expected as the server will delay informing a client of an unsuccessful trade in the presence of high contention for an equity.

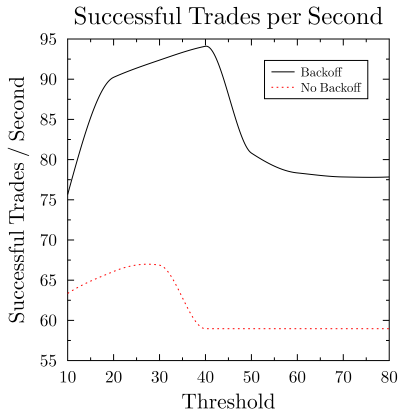

Figure 2 – Trades across whole system with varying rollback thresholds.

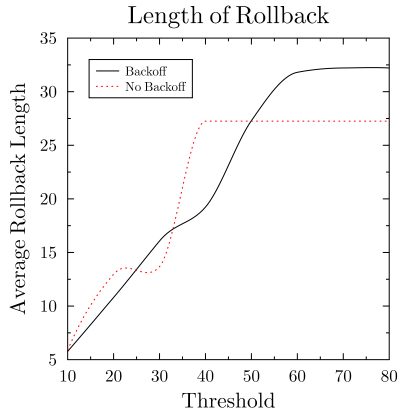

Figure 3 – Average rollback of clients across the system.

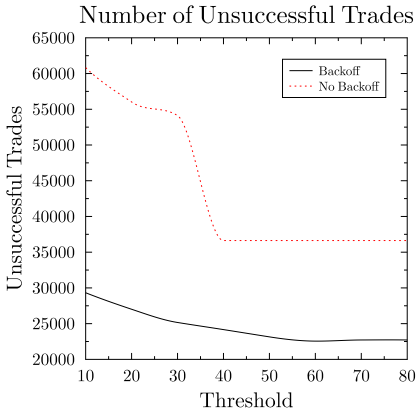

Figure 4 – Total of unsuccessful trades.

An interesting conclusion may be drawn from considering the graphs in figures 2 and 3: an increased average rollback does not result in poorer trading success for a client as long as semantic back off is employed. This can be explained, possibly, by the greater amount of information provided by the server (updated values for equities that a client may trade in the future). In addition, by also lowering the contention of an

equity there is a greater chance of achieving a successful trade as other clients are backed off.

Figure 4 shows the total number of unsuccessful trades over a run of the system. The variable injection rate has a dramatic effect on improving clients when no semantic back off is present. This is to be expected as most schemes that use back off only schemes share this property. The most interesting aspect of this graph is how successful combining variable injection rates with semantic back off is. Even for low injection rates, the system has approximately 50% fewer unsuccessful trades.

V. CONCLUSION

We have described an approach for managing the volatility of HFT environments. Our approach provides an automated self-tuning platform that combines semantic contention management and managed trading frequencies to achieve an optimum trading solution. Our approach can adapt to changes in trading patterns, hardware/software resources and trading volumes during runtime to find an appropriate balance between client trading injection rates and successful trades.

Our approach is founded in optimistic replication schemes and affords high frequency trades by replicating trading markets at the clients. Such replicas maintain the property of eventual consistency. Our earlier work demonstrated the usefulness of semantic contention management for optimistic replication schemes [16]. Here we extend our previous work substantially with the ability to adapt to changes in application behavior during runtime, the addition of automated management of client injection rates (trades per second), and provisioning a solution for HFT environments ([16] was predominantly concerned with e-commerce).

We show via experimentation that our approach provides significant performance improvements in attaining successful trades. Unfortunately, benchmarking for HFT support technology is yet to be described in the literature that is suitable for our purposes. In fact, there is little beyond the relationship of HFT to publish/subscribe middleware in the literature [19]. Therefore, we hope this work brings additional participation to tackling the problem of provisioning HFT software solutions.

Future work will focus on robustness of HFT middleware. Fault-tolerance brings significant processing overheads to any solution (or great cost). We shall explore how pessimistic replication strategies we have developed within n-tier platforms [20] may be integrated into our HFT model with minimal hindrances to performance.

REFERENCES

[1] Kissell, R., & Malamut, R. (2006). Algorithmic decision-making framework.*The Journal of Trading*, *1*(1), 12-21.

[2] Gatev, E., Goetzmann, W. N., & Rouwenhorst, K. G. (2006). Pairs trading: Performance of a relative-value arbitrage rule. *Review of Financial Studies*,*19*(3), 797-827.

[3] Zervoudakis, F., Lawrence, D., Nava, N., Gontikas, G., & Al Merey, M. Perspectives on High-Frequency Trading, http://www0.cs.ucl.ac.uk/staff/f.zervoudakis/documents/Perspectives_on_High-Frequency_Trading.pdf, viewed March 2013

[4] Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency control and recovery in database systems* (Vol. 370). New York: Addison-wesley.

[5] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (pp. 143-154). ACM.

[6] Vogels, W.: Eventually consistent. Communications of the ACM 52(1), 40-44 (2009)

[7] Mockapetris, P. V.: Domain names-implementation and specification. Available at http://www.ietf.org/rfc/rfc1035.txt

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review 41(6), 205-220 (2007)

[9] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35{40 (2010)

[10] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the _fteenth ACM symposium on Operating systems principles. pp. 172{182. ACM (1995)

[11] Kermarrec, A., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube approach to the reconciliation of divergent replicas. In: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing. pp. 210{218. ACM (2001)

[12] Pregui_ca, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. On The Move to Meaningful Internet Systems 2003: CopIS, DOA, and ODBASE pp. 38{55 (2003).

[13] Metcalfe, R. M., & Boggs, D. R. (1976). Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, *19*(7), 395-404.

[14] Anderson, T. E., Lazowska, E. D., & Levy, H. M. (1989). The performance implications of thread management alternatives for shared-memory multiprocessors. *Computers, IEEE Transactions on*, *38*(12), 1631-1644.

[15] Scherer III, W. N., & Scott, M. L. (2005, July). Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing* (pp. 240-248). ACM.

[16] Abushnagh Y, Brook M, Sharp C, Ushaw G, Morgan G. Liana: A Framework That Utilizes Causality to Schedule Contention Management across Networked Systems.In: On the Move to Meaningful Internet Systems (OTM). ODBASE 2012, volume 7566 of Lecture Notes in Computer Science, page 871-878. Springer, Rome, Italy, (2012).

[17] Nutt, G., & Bayer, D. (1982). Performance of CSMA/CD networks under combined voice and data loads. *Communications, IEEE Transactions on*,*30*(1), 6-11.

[18] University of Edinburgh, SimJava. Available at: http://www.dcs.ed.ac.uk/home/hase/simjava/ (Accessed: 14 March)

[19] Jayaram, K., Jayalath, C., & Eugster, P. (2010). Parametric subscriptions for content-based publish/subscribe networks. *Middleware 2010*, 128-147.

[20] Kistijantoro, A. I., Morgan, G., Shrivastava, S. K., & Little, M. C. (2008). Enhancing an application server to support available components. *Software Engineering, IEEE Transactions on*, *34*(4), 531-545.