# Semantically Aware Contention Management for Distributed Applications

Matthew Brook, Craig Sharp, Graham Morgan

# Semantically Aware Contention Management for Distributed Applications

Matthew Brook, Craig Sharp, and Graham Morgan

School of Computing Science, Newcastle University
{m.j.brook1, craig.sharp, graham.morgan}@ncl.ac.uk

**Abstract.** Distributed applications that allow replicated state to deviate in favour of increasing throughput still need to ensure such state achieves consistency at some point. This is achieved via compensating conflicting updates or undoing some updates to resolve conflicts. When causal relationships exist across updates that must be maintained then conflicts may result in previous updates also needing to be undone or compensated for. Therefore, an ability to manage contention across the distributed domain to pre-emptively lower conflicts as a result of causal infringements without hindering the throughput achieved from weaker consistency is desirable. In this paper we present such a system. We exploit the causality inherent in the application domain to improve overall system performance. We demonstrate the effectiveness of our approach with simulated benchmarked performance results.

**Keywords.** replication, contention management, causal ordering

## 1 Introduction

A popular technique to reduce shared data access latency across computer networks requires clients to replicate state locally; data access actions (reads and/or writes) become quicker as no network latency will be involved. An added benefit of such an approach is the ability to allow clients to continue processing when disconnected from a server. This is of importance in the domains of mobile networks and rich Internet clients where lack of connectivity may otherwise inhibit progress.

State shared at the client side still requires a level of consistency to ensure correct execution. This level is usually guaranteed by protocols implementing *eventual consistency*. In such protocols, reconciling conflicting actions that are a result of clients operating on out-of-date replicas must be achieved. In this paper we assume a strict case of conflict that includes out-of-date reads. Such scenarios are typical for rich Internet clients where eventual agreement regarding data provenance during runtime can be important.

Common approaches to reconciliation advocate compensation or undoing previous actions. Unfortunately, the impact of either of these reconciliation techniques has the potential to invalidate the causality at the application level within clients (semantic causality): all tentative actions not yet committed to shared state but carried out on the

local replica may have assumed previous actions were successful, but now require reconciliation. This requires tentative actions to be rolled back. For applications requiring this level of client-local causality, the impact of rolling back tentative actions has a significant impact on performance; they must rollback execution to where the conflict was discovered.

Eventually consistent applications exhibiting strong semantic causality that need to rollback in the presence of conflicts are similar in nature to transactions. Transactions, although offering stronger guarantees, abort (rollback state changes) if they can't be committed. In recent years transactional approaches have been used for regulating multi-threaded accesses to shared data objects. A contention manager has been shown to improve performance in the presence of semantic causality across a multi-threaded execution. A contention manager determines which transactions should abort based on some defined strategy relating to the execution environment.

In this paper we present a contention management scheme for distributed applications where maintaining semantic causality is important. We extend our initial idea [10] by dynamically adapting to possible changes in semantic causality at the application layer. In addition, we extend our initial idea of a single server based approach to encompass n-tier architectures more typical of current server side architectures.

In section 2 we describe background and related work, highlighting the notion of borrowing techniques from transactional memory to benefit distributed applications. In section 3 we describe the design of our client/server interaction scenario. In section 4 we describe our contention management approach with enhanced configurability properties. In section 5 we present results from our simulation demonstrating the benefits our approach can bring to the system described in section 3.

## 2 Background and Related Work

Our target application is typically a rich Internet client that maintains replicas of shared states at the client side and wishes to maintain semantic causality. Such an application could relate to e-commerce, collaborative document editing or any application where the provenance of interaction must be accurately captured.

### 2.1 Optimistic replication

Optimistic protocols allow for a deviation in replica state to promote overall system throughput. They are ideal for those applications that can tolerate inconsistent state in favour of instant information retrieval (e.g., search engines, messaging services). The guarantee afforded to the shared state is eventual consistency [3], [4].

Popular optimistic solutions such as Cassandra [5] and Dynamo [6] may be capable of recognising causal infringement, but do not provision rollback schemes to enforce semantic causality requirements at the application layer within their design. They are primarily designed, and work best, for scalable deployment over large clusters of servers. They are not designed for distributed clients to maintain replication of shared state. However, earlier academic work did consider client-based replication. Bayou

[7] and Icecube [8] [9] do attempt to maintain a degree of causality, but only at the application programmer's discretion. In such systems the application programmer may specify the extent of causality, preventing a total rollback and restart. This has the advantage of exploiting application domains to improve availability and timeliness, but does complicate the programming of such systems as the boundary between protocol and application overlap. In addition, the programmer may not be able to predict the semantic causality accurately.

### 2.2 Transactions

Transactions offer a general platform to build techniques for maintaining causality within replication accesses at the client side that does not require tailoring based on application semantics. Unfortunately, they impose a high overhead to an application that negates the scalable performance expected from optimistic replication: maintain ordering guarantees for accesses at the server and clients complete with persistent fault-tolerance.

Transactional memory [12] approaches found in multi-threaded programming demonstrate fewer of such guarantees (e.g., persistence). In addition, unlike typical transactions in database processing multi-threaded programs present a high degree of semantic causality (threads executing and repeatedly sharing state). Therefore, it is no surprise to learn that they have been shown to help improve overall system throughput by judicial use of a contention manager [1] [2] [13]. Although no single contention manager works for all application types [14], dynamism can be used to vary the contention strategy.

### 2.3 Contribution

In our previous work we successfully borrowed the concept of contention management from transactional memory and described a contention manager that may satisfy our rich Internet client setting [10]. We derived our own approach based on probability of data object accesses by clients. Unfortunately, the approach had limitations: (1) it was static and could not react to changes in application behaviour (i.e., probability of object accesses changing); (2) it worked on a centralised server (i.e., we could not utilise scalability at the server side). In this paper we propose solutions to both these problems and present a complete description of our eventually consistent protocol (which we didn't present earlier) with the dynamic version of our semantically aware contention manager.

## 3 System Design

Our contention management protocol is deployed within a three-tier server side architecture as illustrated in Fig. 1. The load balancer initially determines which application server to direct client requests to. A sticky session is then established such that all messages from one client are always directed to the same application server. Commu-

nication channels exhibit FIFO qualities but message loss is possible. We model this type of communication channel to reflect a typical TCP connection with the possibility of transient connectivity of clients, a requirement in many rich Internet client applications.

Data accesses are performed locally on a replication of the database state at the client side. In our evaluation we used a complete replica, but this could be partial based on a client's ability to gain replica state on initialisation of a client. Periodically clients inform the application server of these data accesses. An application server receives these access notifications and determines whether these accesses are valid given the current state as maintained at the database. Should the updates be valid, the database state is updated to reflect the changes the client has made locally. However, if the update results in an irreconcilable conflict then the client is notified. When the client learns that a previous action was not successful, the client rolls back to the point of execution where this action took place and resumes execution from this point.

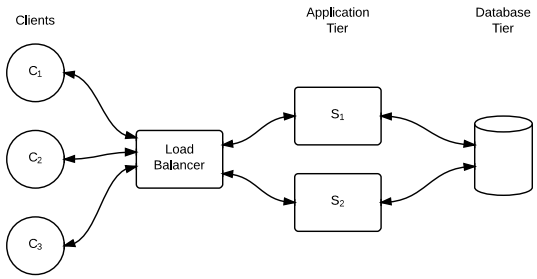


**Fig. 1.** System Design

### 3.1 Clients

Each client maintains a local replica of the data set maintained by the database. All client actions enacted on the shared data are directed to their local replica. The client uses a number of logical clocks to aid in managing their execution and rollback:

- *Client data item clock (CDI)* – exists for each data item and identifies the current version of the data item's state held by a client. The value is used to identify when a client's view of the data item is out-of-date. This value is incremented by the client when updating a data item locally or when a message is received from an application server informing the client of a conflict.
- *Client session clock (CSC)* – this value is attached to every request sent to an application server. When a client rolls back this value is incremented. This allows the application server to ignore messages belonging to out of date sessions.

- *Client action clock (CAC)* – this value is incremented each time a message is sent to an application server. This allows the application servers to recognize missing messages from clients.

The result of an action that modifies a data item in the local replicated state results in a message being sent to the application servers. This message contains the data item state, the CDI of the data item, the CSC and the CAC. An execution log is maintained and each client message is added to it. This execution log allows client rollback.

A message arriving from the application server indicates that a previous action, say $A_n$, was not possible or client messages are missing. All application server messages contain a session identifier. If this identifier is the same or lower than the client's CSC then the application server message is ignored (as the client has already rolled back – the application server may send out multiple copies of the rollback message). However, if the session identifier is higher than the client's CSC the client must update their own CSC to match the new value and rollback.

If the message from the application server was sent due to missing client messages then only an action clock and session identifier will be present (we call this the *missed message request*). On receiving this message type, the client should rollback to the action point using their execution log. However, if the application server sent the message because of a conflicting action then this message will contain the latest state of the data that $A_n$ operated on and the new logical clock value (we call this the *irreconcilable message request*). On receiving such a message the client halts execution and rolls back to attempt execution from $A_n$.

Although a client will have to rollback when requested by the application server, the receiving of an application server message also informs the client that all their actions prior to $A_n$ were successful. As such, the client can reduce the size of their execution log to reflect this.

### 3.2 Application Server

The role of an application server is to manage the causal relationship between a client's actions and ensure a client's local replica is eventually consistent. The application server manages three types of logical clock to inform the client when to rollback:

- *Session identifier (SI)* – this is the application server's view of a client's CSC. Therefore, the application server maintains an SI for each client. This is used to disregard messages from out of date sessions from clients. The SI is incremented by one each time a client is requested to rollback.
- *Action clock (AC)* – this is the application server's view of client's CAC. Therefore, the application server maintains an AC for each client. This is used to identify missing messages from a client. Every action honoured by the application server on behalf of the client results in the AC for that client being set to the CAC belonging to the client.
- *Logical clock (LC)* – this value is stored with the data item state at the database. The value is requested by the application sever when an update message is received from a client. The application server determines if a client has operated on

an out-of-date version using this value. If the action from the client was valid then the application server updates the value at the database. Requests made to the database are considered transactional; handling transactional failure is beyond the scope of this paper (we propose the use of the technique described in [11] to handle such issues).

A message from a client, say $C_1$, may not be able to be honoured by the application server due to one of the following reasons:

- *Stale session* – the application server's SI belonging to $C_1$ is less than the CSC in $C_1$'s message.
- *Lost message* – the CAC in $C_1$'s message is two or more greater than the application server's AC for $C_1$.
- *Stale data* – the LC for the data item the client has updated is greater than the CDI in $C_1$'s message.

When the application server has to rebut a client's access, a rollback message is sent to that client. Preparation of the rollback message depends on the state of the client as perceived by the application server. An application server can recognize a client ($C_1$) in one of two modes:

- *Progress* – the last message received from $C_1$ could be honoured.
- *Stalled* – the last message received from $C_1$ could not be honoured or was ignored.

If $C_1$ is in the progress state then the application server will create a new rollback message and increment the SI for $C_1$ by one. If the problem was due to a lost message then the AC value for $C_1$ is incremented by one (to indicate that rollback is required to just after the last successful action) and is sent together with $C_1$'s updated SI value (this is the *missed message request* mentioned in section 3.1). If the problem was due to an irreconcilable action the message sent to the client will contain the latest LC for the data item the action attempted to access (retrieved from the database), and the application server's SI value for $C_1$ (this is the *irreconcilable message request* mentioned in section 3.1). The application server moves $C_1$ to the stalled state and records the rollback message sent to $C_1$ (this is called the *authoritative rollback message*).

If $C_1$ is in the stalled state all the client's messages are responded to with $C_1$'s current authoritative rollback message. The exception is if the received message contains a CSC value equal to $C_1$'s SI value held by the application server. If such a message is received then the CAC value contained in the message is compared with the AC value of $C_1$ held by the application server. If it is greater (i.e., the required message from $C_1$ is missing) the application server increments $C_1$'s SI by one and constructs a new authoritative rollback message to be used in response to $C_1$. If the CAC value in the message is equivalent to the AC value of $C_1$ as held by the application server, and the application server can honour this message (logical clock values are valid), then $C_1$'s state is moved to progress and the authoritative rollback message is discarded. If the message cannot be honoured (it is irreconcilable), then the application server increments the SI for $C_1$ by one and uses this together with the contents of the received message to create a fresh authoritative rollback message, sending this to the client.

### 3.3 Database

The database manages the master copy of the shared data set. The data set comprises of data items and their associated logical clock values. The data item reflects the state while the logical clock indicates versioning information. The logical clock value is requested by application servers to be used in determining when a client's update message is irreconcilable. The database accepts requests to retrieve logical clock values for data items or to update the state and logical clock values (as a result of a successful action as determined by an application server). We assume application servers and databases interact in standard transactional ways.

### 3.4 System Properties

The system design described so far can be reasoned about in the following manner:

- *Liveness* – Clients progress until an application server informs them that they must rollback (via an authoritative rollback message). If this message is lost in transit the client will continue execution, sending further access notification to the application server. The application server will continue to send the rollback message in response until the client responds appropriately. If the client message that is a direct response to the authoritative rollback message goes missing the application server will eventually realize this due to receiving client messages with the appropriate SI values but with CAC values that are too high. This will cause the application server to respond with an authoritative rollback message.
- *Causality* – A client always rolls back to where an irreconcilable action (or missing action due to message loss) was discovered by the application server. Therefore, all actions that are reconciled at the application server and removed from a client's execution log maintain the required causality. Those tentative actions in the execution log are in a state of reconciliation and may require rolling back.
- *Eventually Consistent* – If a client never receives a message from an application server then either: (i) all client requests are honoured and states are mutually consistent; or (ii) all application server or client messages are lost. Therefore, as long as sufficient connectivity between client and application servers exists, the shared data will become eventually consistent.

The system design provides opportunity for clients to progress independently of the application server in the presence of no message loss and no irreconcilable issues on the shared data. In reality, there will be a number of irreconcilable actions and as such the burden of rolling back is much more substantial than other eventually consistent optimistic approaches. This does, however, provide the benefit of not requiring any application level dependencies in the protocol itself; the application developer does not need to specify any exception handling facility to satisfy rollback.

## 4 Semantic Contention Management

We now describe our contention management scheme and how it is applied to the system design presented in the previous section. The aim of the contention management scheme is to attain a greater performance in the form of fewer irreconcilable differences without hindering overall throughput.

Like all contention management schemes, we exploit a degree of predictability to achieve improved performance. We assume that causality across actions is reflected in the order in which a client accesses shared data items. The diagram in Fig. 2 illustrates this assumption.

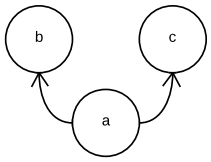


**Fig. 2.** Relating client actions progressing to data items

In the simple graph shown in Figure 2 we represent three data items (*a*, *b* and *c*) as vertices with two edges connecting *a* to *b* and *c*. The edges of the graph represent the causal relationship between the two connected data items. So if a client performs a successful action on data item *a* there is a higher than average probability that the focus of the next action from the same client will be either data item *b* or *c*.

Each application server manages their own graph configuration representing the data items stored within the database. Because of this graphs will diverge across application servers. This is of no concern, as an application server must reflect the in-session causality of its own clients, not the clients of others. We extend the system design described in the previous section by adding the following constructs to support the contention management framework:

- *Volatility value (VV)* – a value associated to each vertex of the graph indicating the relative popularity for the given data item. The volatility for a data item in the graph is incremented when a client's action is successful. The volatility for the data item that was the focus of the action is incremented by one and the neighbouring data items (those that are connected by outgoing arcs of the original data item) volatilities are incremented by one. Periodically, the application server will decrement these values to reflect the deterioration of the volatility for nodes that are no longer experiencing regular data access.
- *Delta queue (DQ)* – for those actions that could not be honoured by the application server due to irreconcilable conflicts (out-of-date logical clock values) a backoff period is generated as the sum of the volatility for the related data. These related

data items include the original data item for which the conflict occurred along with the data items with the highest volatilities up to three hops away in the graph. This client is now considered to be in a stalled state and is placed in the delta queue for the generated backoff period. The backoff period is measured in milliseconds given a value generated from the volatility values.

- *Enhanced authoritative rollback message* – when a backoff period expires for a client residing in the delta queue, an enhanced authoritative rollback message is sent to the client. This is an extension of the authoritative rollback message described in the system design that includes a partial state update for the client. This partial state update includes the latest state and logical clock values for the conflicting data item and the data items causally related to the original conflicting access. Based on the assumption of causality as reflected in the graph configuration, the aim here to pre-emptively update the client. As a result, future update messages will have a higher chance of being valid (this cannot be guaranteed due to simultaneous accesses made by other clients).

The approach we have taken is a server side enhancement. This decision was taken to alleviate clients from active participation in the contention management framework. The client needs only to be able to handle the enhanced authoritative rollback message that requires additional state updates to the client's local replica.

As each application server manages their graph structure representing the data items, should a single application server crash, client requests can be directed to another working application server will little loss. Clients that originally had sessions belonging to the crashed application server will require directing to a different application server and there will be some additional conflicts and overhead due to the lost session data.

### 4.1 Graph Reconfiguration

To satisfy the changing probabilities of causal data access over time our static graph approach requires only minor modifications.

We introduce two new values that an application server maintains for each client:

- *Happens Before Value (HBV)* – the vertex representing a data item a client last successfully accessed.
- *Happens After Value (HAV)* – the vertex representing a data item a client successfully accessed directly after HBV.

If there does not exist a link between HBV and HAV then one is created. 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 (e.g., increased sized enhanced authoritative rollback message). Therefore, to allow for the deletion of edges as well as the creation of edges we make use of an additional value to record the popularity of traversal associated to each edge in the graph:

- *Edge Popularity Value (EPV)* – The cumulative number of times, across all clients, a causal occurrence has occurred between a HBV and HAV.

If there already exists a link between HBV and HAV then the associated edge's EPV is incremented by one. This provides a scheme within which the most popular edges will maintain the highest values. However, this may not reflect the current popularity of the causal relations, therefore, the EPVs purpose is to prune the graph. Periodically, the graph is searched and EPVs below a certain threshold result in the removal of their edges. After pruning the graph all remaining edges are reset to the value 0.

Periodic pruning and resetting of EPVs provides our scheme with a basic reconfiguration process to more appropriately reflect current semantic causal popularity. 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 e-commerce site with items for sale then they may be introduced as graph reconfiguration occurs. This has two benefits: (i) introduction of items may well alter the causal relationships dramatically (e.g., timed flash sales) 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 items have on existing data items.

# 5 Evaluation

Three approaches were evaluated to determine performance in terms of server side conflicts and throughput: (1) the basic protocol as described in the system design with no contention management; (2) the enhanced protocol with contention management but without graph reconfigurations; (3) the enhanced protocol with both contention management and graph reconfiguration. To create an appropriate simulation scenario we rely on a pattern of execution for rich Internet clients similar to that described in [16] (ecommerce end client sales).

## 5.1 Simulation Environment

We produced a discrete event simulation using the SimJava [15] framework. We modeled variable client numbers, a load balancer, three application servers and a database as processes.

Graph layouts are randomly created and client accesses are pre-generated. The initial graph layouts include vertices with and without edges. In the dynamic scenario such a vertex may at some point become connected, but not in the static graph.

In the dynamic graph periodic reconfiguration occurred every thirty seconds with a relaxed threshold of one. This period was determined over experimentation and was found to provide reasonable balance between accurate causality representation and overhead induced by reconfiguration. The relaxed threshold simply indicated edges that had shown any causal interest would be guaranteed a starting presence in the graph after reconfiguration.

We simulated message delay between client and application servers (load balancer) as a random variable with a normal distribution between 1 - 50 milliseconds. Each client performs 200 data accesses then leaves the system. Each experiment was run five times to generate the average figures presented. The arrival rate of client messages to the application server was set as ten messages per second for each client process. The simulation was modeled with a 2% message loss probability. Database read and writes were 3 and 6 microseconds respectively.

### 5.2 Evaluation 1 – Irreconcilable Client Updates (Conflicts)

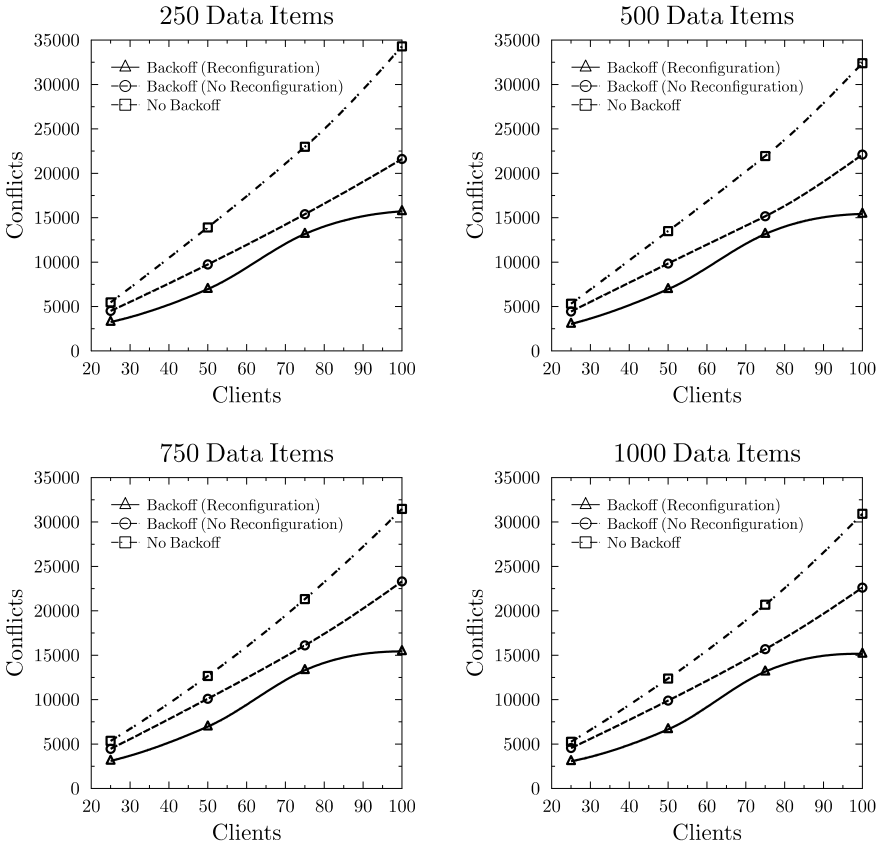


**Fig. 3.** Irreconcilable conflicts for varying graph sizes

The graphs in figure 3 show that the inclusion of contention management lowers conflicts. The results also show the added benefit of graph reconfiguration over a static graph. In addition, reconfiguration appears to approach a stable state as the contention increases. Reconfiguration allows for the system to adapt to the changing client interactions resulting in the graph more accurately reflecting semantic causality over time. Without reconfiguration the conflicts continue to rise rather than stabilize. What has little impact on the results is the number of data items represented in the graph. This is due to the predictability exhibited in the client accesses: if clients accessed data at random we would expect that graph size mattered, as there would naturally be less conflicts.

### 5.3 Evaluation 2 – Throughput of successful client actions (commits)

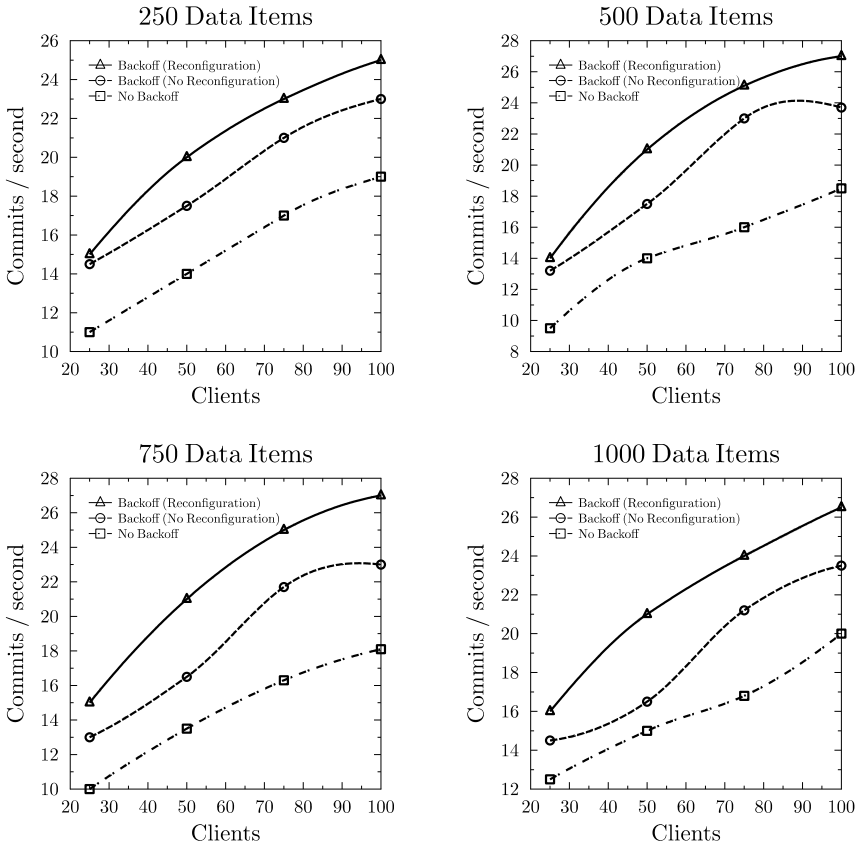


**Fig. 4.** Throughput measured as commits per second for varying graph sizes

Similar to the previous set of the results, the graph size plays little role given the predictive behaviour of the clients. The results show our backoff contention management

providing the best throughput. Interestingly, without reconfiguration the system appears to reach saturation point early. With reconfiguration we still have improvement occurring, although it is tailoring off towards 100 clients.

The results presented here indicate that backing off clients and updating their replicas in a predictive manner actually improves performance: conflicts lowered and throughput is increased. In terms of throughput alone, this is seen as a significant 30% improvement when combined with reconfiguration. Therefore, we conclude that causality at the application layer can be exploited to improve performance for those applications where causality infringement requires rollback of local replica state.

## 6 Conclusion

We have described an optimistic replication scheme that makes use of dynamic contention management. We base our contention manager on the popularity of data accesses and the possible semantic causal relation this may hint at within the application layer. Our approach is wholly server based, requiring no responsibility for managing contention from the client side (apart from affording rollback). Our approach suits applications where causality is important and irreconcilable accesses of shared state may cause a client to rollback accesses tentatively carried out on a local replica. Such scenarios occur in rich Internet clients where provenance of data access is to be maintained or where actions of a client's progress must be rolled back in the context of achieving a successful client session. We describe our approach in the context of n-tier architectures, typical in application server scenarios.

Our evaluation, via simulation, demonstrates how overall throughput is improved by reducing irreconcilable actions on shared state. In particular, we show how adapting to changes in causal relationships during runtime based solely on access patterns of clients provide greatest improvements in throughput.

This is the first time runtime adaptability of causality informed contention management has been demonstrated in a complete solution exhibiting eventual synchronous guarantees. As such, we believe that this is not only a useful contribution to the literature, but opens new avenues of research by bringing the notion of contention management to replication protocols.

We acknowledge that our approach is focussed on a particular application type: applications that always rollback to where conflict was detected. However, we believe that advocating contention management as an aid to performance for eventually consistent replicated state in general would be beneficial and worthy of future exploration.

Future work will focus on peer-to-peer based evaluation and creating contention management schemes suitable for mobile environments (where epidemic models of communication are favoured). A further opportunity of exploration will be in taking the semantic awareness properties of this work back to transactional memory systems themselves.

## References


1. Scherer III, W N., Scott M., L.: Contention Management in Dynamic Software Transactional Memory. In: PODC Workshop on Concurrency and Synchronization in Java programs, pp. 70-79 (2004)
2. Scherer III, W. N., Scott. M., L.: Advanced Contention Management for Dynamic Software Transactional Memory. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, pp. 240-248. ACM, New York (2005)
3. Saito, Y., Shapiro, M.: Optimistic Replication. ACM Computing Surveys. 37, 42–81 (2005)
4. Vogels, W.: Eventually Consistent. Communications of the ACM. 52, 40–44 (2009)
5. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review 44, pp. 35–40 (2010)
6. 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. In: 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, New York (2007)
7. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In: 15th ACM Symposium on Operating Systems Principles, pp. 172–182. ACM, New York (1995)
8. Kermarrec, A., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube Approach to the Reconciliation of Divergent Replicas. In: 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 210–218. ACM, New York (2001)
9. Pregui¸ca, N., Shapiro, M., Matheson, C.: Semantics-Based Reconciliation for Collaborative and Mobile Environments. In: Meersman, R., Tari, Z., Schmidt, D. (eds.) On The Move to Meaningful Internet Systems 2003. LNCS, vol. 2888, pp. 38–55. Springer, Heidelberg (2003)
10. Abushnagh, Y., Brook, M., Sharp, C., Ushaw, G., Morgan, G.: Liana: A Framework that Utilizes Causality to Schedule Contention Management across Networked Systems. In: Meersman, R. et al. On The Move to Meaningful Internet Systems 2012. LNCS, vol. 7566, pp. 871-878. Springer, Heidelberg (2012)
11. Kistijantoro, A. I., Morgan, G., Shrivastava, S. K., & Little, M. C.: Enhancing an Application Server to Support Available Components. In: IEEE Transactions on Software Engineering, 34(4), 531-545 (2008)
12. Herlihy, M., & Moss, J. E. B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, vol. 21, no. 2, pp. 289-300. ACM, New York (1993)
13. Herlihy, M., Luchangco, V., Moir, M., & Scherer III, W. N.: Software Transactional Memory for Dynamic-sized Data Structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing. pp. 92-101. ACM, New York (2003)
14. Guerraoui, R., Herlihy, M., & Pochon, B.: Polymorphic Contention Management. In: 19th International Symposium on Distributed Computing, pp. 303-323. Springer (2005)
15. University of Edinburgh, SimJava. Available at: http://www.dcs.ed.ac.uk/home/hase/simjava/ (Accessed: 16 February 2013)
16. Clarke, D., Morgan, G.: E-Commerce with Rich Clients and Flexible Transactions. In: First International Workshop on Software Technologies for Future Dependable Distributed Systems, pp. 73-77, IEEE (2009)