Liana: A Framework That Utilizes Causality to Schedule Contention Management across Networked Systems

Short Paper

Yousef Abushnagh, Matthew Brook, Craig Sharp, Gary Ushaw, and Graham Morgan

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

Abstract. In this paper we tackle the problem of improving overall throughput for shared data access across networked systems. We are concerned with those systems that must maintain the causal ordering of events across participating nodes in a network. By utilising the causality inherent within such systems we create a scheduler that can inform contention management schemes in such a way as to reduce the rollback associated to conflicting access of shared data.

Keywords: message ordering, contention management, causal ordering.

1 Introduction

In networked environments concurrent access to shared data is achieved with a degree of data replication to offset network latencies and improve data availability. Processes operate on their own private representation of shared data and at some point the result of such operations are shared (appear in the private space of other processes). However, sometimes accesses originating from different processes are irreconcilable. This requires a choice to be made to decide which process accesses succeed and which do not. Making such a choice in a fair manner is the job of a contention manager.

In this paper we present a framework, Liana, which provides contention management. We are concerned with improving throughput in client/server architectures. A client may progress independently of the server using a local replica of server state. Periodically, such state is reconciled across the server and all clients. Liana is specifically designed for environments where preserving causality.

We use causality inherent in the application layer to our advantage in that we assume a degree of predictability in a clients actions. We use predictability in two ways: (1) managing contention for popular items of shared data via a backoff scheme; (2) pre-emptively update a clients shared state of the data items they are likely to access in the future. Although backoff has been demonstrated in message ordering to attain replication schemes with fault-tolerant properties (e.g., [1], [2]), this is the first time it has been used with predictability of causality in a contention manager.

2 Background and Related Work

Eventual consistency [3], [4] is the term used to describe the property that guarantees the convergence of replicated states within an optimistic replication scheme. In principle, 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.

Achieving an implementation of an optimistic replication scheme that enforces eventual consistency balances the requirements of consistency, availability and timeliness. Increasing consistency (moving towards a more pessimistic approach) negates availability and timeliness. However, increasing inconsistency places a greater burden on the application as exceptions that are the result of irreconcilable differences in shared state must be handled.

The amount of work required by an application to retrospectively handle exceptions increases as the need to maintain causality across actions on shared state increases. This is because an inability to satisfy an action that resulted in an irreconcilable difference in shared state means that subsequent actions carried out by a client may also be void. This is the type of application we are primarily concerned with.

Popular optimistic solutions such as Cassandra [5] and Dynamo [6] do not enforce such strong causality requirements. However, earlier academic work like Bayou [7] and Icecube [8], [9] do attempt to maintain a degree of causality, but only at the application programmers discretion. Bayou and Icecube rely on the application programmer to 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.

If the causality requirements of an application span many actions then the complication of injecting application specific exception handling will increase. In some instances, such exception handling defaults to an undesirable rollback scenario as ignoring irreconcilable actions or attempting alternative actions may not be viable. This is the worst-case scenario for applications with strong causality requirements, as the reconciliation of optimistic protocols does nothing but increase overhead, both in terms of throughput and complexity of the programming model. In fact, the model becomes transactional in nature.

We require a contention manager that operates in a transactional like environment (as clients rollback in the presence of irreconcilable data accesses), but unlike transactions can make use of application level semantic knowledge (probability of future causal relations) similar to optimistic approaches. However, such semantic knowledge is only maintained at the server side meaning we want a contention manager that does not rely on a clients understanding of how to best exploit such knowledge (similar to transactions as clients are told to simply rollback or continue).

3 Approach

For the purposes of clarity we assume a client/server architecture in which the server maintains all shared state and clients maintain replicas of such state. However, there is no reason why this cant be extended to peer-to-peer (assuming each client holds shared state with epidemic message propagation as in Bayou). Communication channels between clients and server exhibit FIFO qualities but may lose messages. Clients are fail-stop but may return to execution (no byzantine).

Clients carry out actions on their local replica representing server state and periodically inform the server of their shared data accesses. A server receives these access notifications from a client and attempts to carry out all the clients actions on the master copy of the shared state. However, if this is not possible due to irreconcilable actions then clients are informed. On learning that a previous action was not achieved at the server, a client rolls back to the point of execution where this action took place and resumes execution from this point. Subsequent actions from such a restart may not be the same (the system is dynamic in this sense).

Like all contention management schemes, we exploit a degree of predictability to improve performance. We assume that causality across actions is reflected in the order in which a client accesses shared data items. The diagram in Fig. 1 describes this assumption.

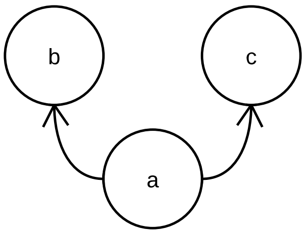

Fig. 1. Relating action progression to data items

Considering the example shown in Fig. 1 we now describe the essence of causality exploitation in our contention management scheme. We show three data items (a, b, c). If a client, say C_1, has carried out an action that successfully accessed data item a we can state the following: there is a higher than average chance that data item b will be the focus of the next action carried out by C_1; there is a higher than average chance that data item c will be the focus of the next action carried out by C_1; there is a chance (less than going to b or c) that another data item (could be anywhere in the shared state) will be the focus of the next action carried out by C_1.

The server maintains shared data as a directed graph. The graph indicates the likelihood of clients accessing individual items of shared state given the last item of shared state they accessed. To achieve this we use the following meta-data:

– *A Logical clock (LC)* - this value represents the version number of a data item to determine if a clients view of a data item is out of date. Therefore, the server maintains an LC for each data item. Whenever an action successfully accesses a shared data item, the LC of that data item is incremented by one.
– *Volatility value (VV)* - a value associated to each data item indicating its popularity. Whenever a successful action accesses a data value its volatility value is incremented by 2 and the neighbouring data items volatility values are incremented by 1.
– *Delta queue (DQ)* - those actions that could not be honoured by the server due to out of date LC values are stored for a length of time on the DQ. This length of time is calculated as the sum of the volatility value of the data item where the LC out of date was discovered together with the highest volatility values of data items up to three hops away on the graph. We call this the DQ time of a client.
– *Enhanced rollback message* - when an action request is de-queued from the DQ an enhanced rollback message is sent to the client who initially sent the action request. An enhanced rollback message includes shared state and LC values of up to 3 hops away in the graph of the most volatile nodes. On receiving such a message a client updates its replica and rolls back.

We now use an example to clearly describe how our backoff contention management works. In Fig. 2 we show a graph held by the server with volatility values shown in the lower right of nodes. Assume the server receives a client message from C_1 indicating access to shared data item *a*. Unfortunately, the request was irreconcilable d

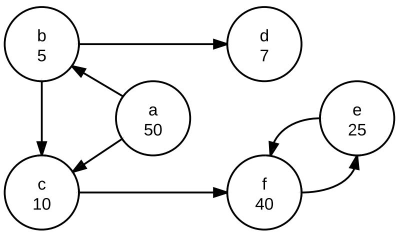

Fig. 2. Graph with volatility values

The action request from C_1 is placed on the delta queue. The DQ time for C_1 is the summation of the volatility values of *a*, *c*, *f* and *e* (50 + 10 + 40 + 25 = 125). These are the highest volatility values up to three hops away.

We chose to measure time as server control loops (i.e., checking the incoming message buffer, attempting action requests from clients, sending messages to clients). Therefore, each time the client carries out an action all DQ time values are decreased by one. For C_1, the server will have to loop through its control process 125 times before its action request can be de-queued. Once de-queued the enhanced rollback message is sent to C_1 complete with all the data item values and LC descriptors of *a*, *c*, *e* and *f* (the LC values are not shown on the diagram). On receiving the enhanced rollback message C_1 can update *a*, *c*, *e* and *f* in its own local replica and rollback as instructed.

There are quite a number of parameter values that we have described that can be changed (e.g., increasing volatility by 2 and neighbours by 1, determining DQ time, 200 as the ceiling value DQ time). This was done to add clarity to the descriptions and these are the values we use in our evaluation. These values represent typical backoff type scheme values and the use of a cliff edge value for DQ ceiling time trades degraded backoff against simplicity. We found that these values provided a suitable environment that exhibits the benefits of our contention manager in this particular style of application under these throughput conditions.

4 Evaluation

In this section we describe how we evaluated our framework and present results for justifying the validity of our work. We implemented the framework as the protocol described in section 3 using the Java programming language (including the values used in the protocol description). Two versions were implemented: (1) without contention management, (2) with contention management. Our primary aim is to compare and contrast these two implementations.

4.1 Environment and Parameters

To afford flexibility we simulated clients and network connectivity using an execution environment deployed on a Core 2 Quad running at 2.66GHz with 4GB RAM. Our operating system was Ubuntu 11.10. Communication delays were chosen at random within a normal distribution that ranged between 100 and 1000 milliseconds (to reflect various load and access variations over the Internet).

The graph located at the server side was generated randomly with an ability to determine the maximum number of arcs leading from a node. Nodes within the graph could have no arcs leading from them, whereas others may have the maximum number of arcs. Arcs cannot loop back to their starting node (they must point to another node in the graph). We chose a maximum of three arcs for these experiments.

The progression of action requests across multiple data items originating from clients resembles the probabilities found within the graph. However, clients have a 10% chance of deviating from the graph probabilities during runtime. That is, the graph represents probabilities of future client actions with a 10% error bound. In this regard we assume a highly predictable environment.

We carried out two evaluations: (1) to determine if backoff contention management can reduce irreconcilable actions and; (2) to determine if throughput would be adversely affected by the introduction of backoff contention management. For each of these evaluations we constructed a number of experiments. We created four graphs with varying numbers of data items (100, 300, 500, 1000). For each graph and framework implementation we ran 10 experiments, each with a different number of clients from 10 through 100 in increments of 10. Each experiment was run a number of times to gain average figures for analysis. The standard deviation across experiments was negligible ($\leq$5%).

4.2 Evaluation 1 - Irreconcilable Client Actions (Conflicts)

The improvement gained by including backoff contention management shows more significantly as graph sizes increase (Fig. 3). This indicates that backoff contention management provides a significant improvement in environments where conflicts do occur, but proportionally less as the size of the replica state increases: if we consider 100 clients, the most favourable circumstance (1000 data items) shows an improvement of approximately 25%, compared to an approximate improvement of 5% in the least favourable circumstance (100 data items).

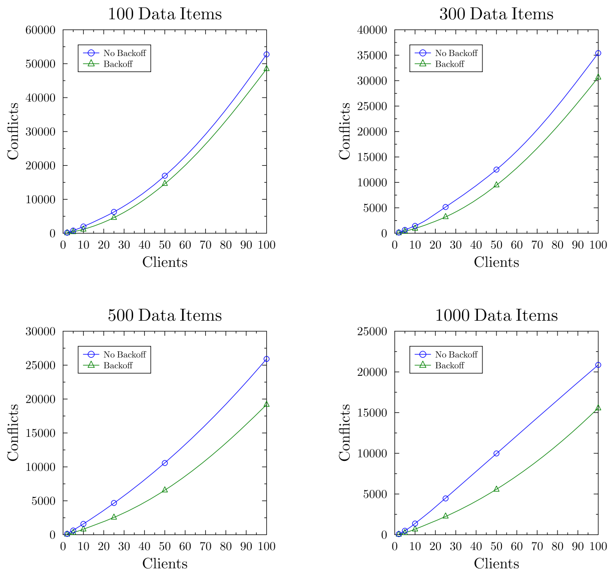

Fig. 3. Framework comparison with varying graph sizes

4.3 Evaluation 2 - Throughput of Successful Client Actions (Commits)

The first observation to make is that including backoff contention management improves throughput for all graph sizes and all ranges of client numbers (Fig. 4). Avoiding deterioration in throughput would make our approach a viable option. However, an improvement in throughput demonstrates that the backoff scheme coupled with pre-emptive updates within the Liana framework improve overall performance for those applications with strong causality requirements.

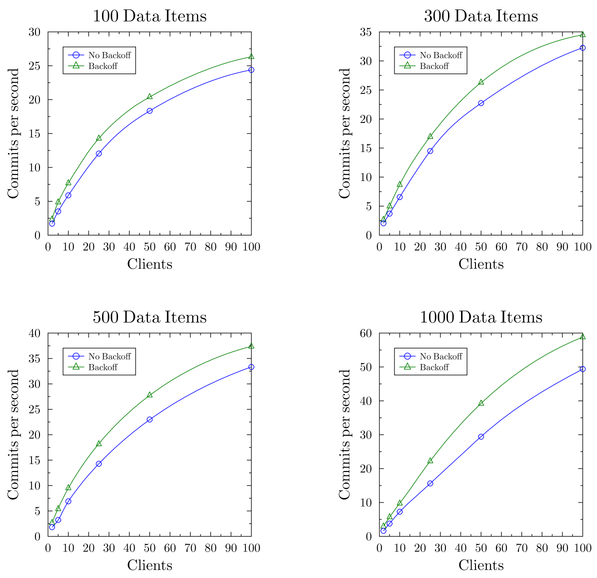

Fig. 4. Framework comparison with varying graph sizes

5 Conclusion

We have described a framework within which contention management can be provided by exploiting semantic relationships between data items. A server was enhanced to enforce our backoff based contention management scheme. A client plays no part in determining backoff, allowing the backoff scheme to appear transparent to client implementations. This also makes the contention management scheme independent from whatever access protocol is used to enforce the required consistency model.

Our evaluation, via experimentation, demonstrates how the backoff contention management scheme improves overall performance by reducing irreconcilable actions on shared state while increasing throughput.

We acknowledge that our approach will only perform better than existing optimistic approaches if causality were a critical factor. That is, those applications within which all existing client actions that occur after the rollback checkpoint are determined void when resolving irreconcilable differences in shared state. However, we believe that such applications can benefit from optimistic replication if used together with a backoff contention management scheme that utilises causality in its prediction of client actions.

Future work will focus on extending the framework for peer-to-peer based evaluation and creating backoff contention management for mobile environments (where epidemic/gossip models of communication are favoured). An interesting aspect of the proposed framework is the initial construction of the directed graph. We created the graph specifically for the purposes of experimentation and associated client behaviour to the graph. However, in a real world situation one would hope that the graph could change over time to reflect client behaviour. Achieving a dynamic graph adaptation in real-time is an avenue we will explore.

References

1. Chockler, G., Malkhi, D., Reiter, M.K.: Backoff Protocols for Distributed Mutual Exclusion and Ordering. In: 21st IEEE International Conference on Distributed Computing Systems, pp. 11–20. IEEE Press, New York (2001)
2. Felber, P., Schiper, A.: Optimistic Active Replication. In: 21st IEEE International Conference on Distributed Computing Systems, pp. 333–341. IEEE Press, New York (2001)
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, 35–40 (2010)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P.: 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. Preguiça, N., Shapiro, M., Matheson, C.: Semantics-Based Reconciliation for Collaborative and Mobile Environments. In: Meersman, R., Schmidt, D.C. (eds.) CoopIS/DOA/ODBASE 2003. LNCS, vol. 2888, pp. 38–55. Springer, Heidelberg (2003)