Scalable Performance through Cooperative Data Shipping

S. Banerjee[*]
Information Science & Telecommunications
University of Pittsburgh
Pittsburgh, PA 15260

P. K. Chrysanthis and **A. Deshpande**[‡]
Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Scalable performance is perhaps the most desirable property in any Internet service implementation in the face of unpredictable surges in demand. In our previous work, we have proposed a data access protocol group two-phase locking (g-2pl), that scales well when the data contention levels are relatively high. In reality, a system may experience severe data contention only a fraction of the total time. In this paper, we propose three optimizations to the g-2pl protocol, such that its performance is superior to other comparable protocols under all system scenarios.

1 Introduction and Motivation

The emergence and initial deployment of several electronic commerce (e-commerce) applications has spurred the need for high performance distributed data servers. Several big corporations such as IBM, JCPenny and Walmart are continuing to build their e-business infrastructure with the assumption that a large percentage of their customers will increasingly shop on-line. Currently, most consumers tend to browse items virtually on the web to mainly do price comparisons or availability, but shop physically in stores. This trend will change as people gain more confidence in the e-commerce infrastructure. It is anticipated that in the future, e-commerce sites will connect a large number of clients that will access these sites over high speed wide area networks. Further, these e-commerce sites will serve a very large volume of transactions that may not have predictable data access patterns. Thus an important challenge is to design data server systems that can scale, and perform well in a dynamic environment as the environment shifts from a "virtual browse, physical buy" to a "virtual browse and buy" environment. In the Internet environment, the network delays between the server and clients are relatively large, and under high data contention environments, this can lead to a throughput bottleneck. Systems need to scale with the number of clients, the geographic span of the network as well as with the data contention levels [2]. In recent years there has been a research thrust in exploring mechanisms to hide the network latency and other processing delays in innovative data access protocols.

Client caching of data items and locks is a popular mechanism that has been proposed to reduce the network latency overheads. In local area networks (LAN) environments, three families of caching algorithm have been proposed to preserve data consistency in the presence of concurrent requests, all derived from the widely used *strict two-phase locking* protocol (2PL) [6], namely, *Server-based 2PL, Optimistic 2PL* and *Callback Locking* [11, 18, 13, 17, 4, 9]. While caching can significantly improve performance [8], the marginal gains decrease rapidly as the network speed is increased and when data items are frequently updated rendering the caches invalid. In fact, the *server-based 2PL* (s-2pl) protocol was found to have the best performance in situations with high data contention in LANs [7]. Further, in a high-speed wide area network (WAN) environment, efforts should be focused on reducing the number of sequential message passing rounds (since each round can incur a significant delay, even if the transmission time is negligible) rather than the data transmission time. Our strategy, embodied in the *group two-phase locking* (g-2pl) protocol [1, 2] assumes a stronger inter-client cooperation, intra-transaction caching at the clients and message grouping. While the group 2PL protocol outperformed other comparable protocols under high data contentions, its performance was comparatively poorer when the data contention was low. In this paper, we propose three optimizations to the g-2pl protocol that enables it to exhibit scalable performance under the entire range of data contention levels, at different network latencies and with a larger number of clients.

In the next section, we provide a brief description of the g-2pl protocol, followed by Section 3 that proposes three optimizations to provide scalable performance.

Section 4 describes our simulation framework followed by Section 5 containing numerical results and their analysis.

2 Data Access Protocols

In this section, the s-2pl and g-2pl protocols are briefly reviewed and they are quantitatively compared in the following section. Although we do not deal with the recovery aspects of the g-2pl protocol here, it is assumed that the sites follow the standard protocol adopted by the s-2pl protocol where each site uses *Write-Ahead Logging* (WAL) and garbage collects its log once the data are made permanent at the server [14].

2.1 The s-2PL Protocol

In the basic server-based two-phase locking (s-2pl) protocol, a data-server preserves data consistency by following the *strict two-phase locking* protocol [6]. The s-2pl protocol ensures data consistency as defined by *serializability* which requires the concurrent, interleaved, execution of requests to be equivalent to some serial, non-interleaved, execution of the same requests [3]. In the s-2pl protocol, each transaction goes through a *growing phase* and a *shrinking phase.* During the growing phase, a transaction requests data items which are shipped to it after the data-server acquires a lock on them. In the shrinking phase, all the locks are released when the transaction is either aborted or committed and all modified data items are returned to the data-server. The clients are not allowed to cache locks across transaction boundaries and a client can be viewed as executing one transaction at a time. A variation of s-2pl that allows caching of locks across transaction boundaries is called *caching 2PL* (c-2PL) protocol [18, 9, 7, 8]. For brevity, in the rest of the paper we focus only on the s-2pl protocol but the results can be extended to the c-2PL protocol.

Access to some data may be done in a shared fashion, with multiple clients *reading* the data item simultaneously. However, in the interest of strict consistency, while multiple clients may read the data simultaneously, no client may write on it. Hence, locks are distinguished into read (shared) and write (exclusive) types and a client cannot acquire a write lock on a data item until the clients reading the data have released their shared locks and vice versa. If the data-server cannot acquire a lock on a data item because another transaction is holding a conflicting lock on the same data, the request is enqueued and the requesting transaction is forced to wait until the lock is released.

2.2 The g-2PL Protocol

The core of the g-2pl protocol [1, 2] is to apply *grouping of messages sent to multiple sites* to the s-2pl protocol, thus reducing the message passing rounds. Grouping of actions involving a single site has been previously used successfully in other situations (e.g., in group commit [5, 10] multiple transactions are committed and acknowledged at a single site). Specifically, the lock (data) granting and release messages are grouped as follows. The data-server collects the lock requests for each data item and creates a *forward list* (FL) of all the clients that have pending lock requests for that data item. When a lock becomes available, the lock is granted to the first client on the forward list and the data item is sent to the client along with the forward list. When a transaction commits, the client sends the new version of the committed data items to the clients next on the respective forward lists. A copy of the forward list is also sent with each data item. If the transaction aborts, the client forwards the unchanged data to the next client. Finally, when the last client on the forward list terminates, it sends the new version of the data to the data-server with the outcome of each transaction executed on the clients on the forward list.

Thus the lock release message of the previous client is combined with the lock grant message of the next client, thereby eliminating one sequential message required by the s-2pl protocol. While the data items have been sent out to a group of clients, the server continues to collect requests. We define the period during which the server does not possess the lock on a data item and collects requests as the *collection window* for the data item. Once the lock is returned and a data-server receives and installs the new version of a data item in the database, the previous collection period ends, a new forward list is created, using which, the server dispatches the data item to the first client on the new forward list. Initially at start-up time and during periods of extremely light loading, the forward-list will contain a single client.

For each data item required in the shared mode by multiple (reading) clients, a copy of the data item is sent to each of the reading clients. At the same time, it also sends a message containing the data item and the list of the shared-mode clients to the next client C_i on the forward list that requires exclusive access. In this way, C_i is enabled to execute and update the data item concurrently with the reading clients. However,

C_i cannot release its updates until it receives a *release* message from all the reading clients. As before, if there are no waiting transactions that need exclusive access, the release messages are returned to the server. This is termed the MR1W (*Multiple Reads One Write*) optimization. To improve performance further, the forward list for each data item may be created according to one of several ordering rules (such as First-in First Out (FIFO), Order by transaction priority, Order by the number of locks held by each transaction, serve read requests first, etc.). Since we showed in [2] that the effect of the various re-ordering schemes is minimal at less than 1% from the FIFO case, the default rule adopted here is FIFO. We next describe a deadlock-avoidance FL ordering rule that is an integral part of the g-2pl protocol.

2.2.1 Deadlock avoidance by FL reordering

Two-phase locking protocols are susceptible to deadlocks [6], and so is the g-2pl protocol. However, in the case of the g-2pl protocol, some deadlocks can be avoided by intelligently creating the forward lists. Specifically, some deadlocks can be avoided if in each of the forward lists, the order of the transactions is the same. Formally, the forward list for each data item can be represented by a transaction precedence graph. The transaction precedence graph is a directed graph which determines the order in which each data item will *move* from one client site to another. In order to ensure linear ordering, transaction precedence graphs need to be made consistent. That is, two transactions T_i and T_j must follow the same order $< T_i, T_j >$ or $< T_j, T_i >$ in every precedence graph involving T_i and T_j. The precedence graph is consistent with the lock granting order and hence the serialization order.

Clearly, this reordering of requests does not require predeclaration and because it occurs within a collection window, the problem of starvation is not encountered. In the worst case, some transactions will be pushed towards the end of the forward list but they will have the chance to access the data. In the case that such reordering of forward lists is not possible, some transactions may have to be aborted and restarted. Repeated (cyclic) restarts can be avoided in a similar way using an aging mechanism as in deadlock detection algorithms. It should be stressed that all these reordering computations are done while the server is waiting for the data items to be returned from the clients in the previous window. Thus, these computations do not increase the transaction blocking time on a lock and in fact increases the utilization of data-server CPU while reducing the transaction response time.

3 Read Optimizations

There are two related issues associated with read-only transactions and the g-2pl protocol. The first concerns a potential deadlock situation caused by *read-only* dependencies and the other is the response time of read-only transactions. This potential deadlock situation is better illustrated using an example.

Example: Consider two transactions $t_1 : read_1(x)\ read_1(y)$ and $t_2 : read_2(y)\ read_2(x)$ both of which request data items x and y for reading in a serial manner but in the opposite order. As soon as the data-server gets the requests $read_1(x)$ and $read_2(y)$, it will release x to t_1 and y to t_2. Now, both transactions have one data item and will not release it until they commit or abort. Subsequently, the data-server will get the requests $read_1(y)$ and $read_2(x)$ but neither data item can be released until t_1 or t_2 either commits or aborts returning x and y back to the server respectively. This is a deadlock situation where t_1 waits for t_2 to release the read lock for y and t_2 waits for t_1 to release the read lock for x. The only way to resolve the situation is to abort one of the transactions.

The second issue concerns the response time of read-only accesses. The data server responds to the next set of requests only when the data item (and lock) is returned back to it by the previous set of clients. This can unnecessarily delay read requests that have no conflict with the previous requests. In this section, three main optimizations are proposed, each of which contributes towards improving the performance of the g-2pl scheme. They try to reduce the number of read-read deadlocks as much as possible, and dispatch reads as soon as possible. A few other optimizations, which did not produce expected improvements are also discussed, with possible reasons for their poorer showing. The schemes mentioned ensure that g-2pl, with its read optimizations always performs better than s-2pl, and increase the performance gains by grouping even further.

3.1 Basic Read Optimization

The g-2pl-robasic protocol is the first and most obvious read optimization. When a read request arrives at the server, the server checks if the last forward list sent was **read-only** (A forward list is **read-only** if it consists only of read lock requests). Since read locks are shared, it makes sense to dispatch this read request immediately, provided that there are no writes requests for that data item at the server waiting to be dispatched. Write locks are exclusive, and a read request following the write can only be serviced after the write lock is released. Thus, in this optimization, when a read request for a data item arrives, if the server finds that the previous forward list for that data item was read-only, and that no write is waiting to be serviced for that data item (the wait list is empty), then the server dispatches the read request immediately.

The server in this case has to keep track of whether the previous forward list was read-only. However as a copy of the forward list that was sent out last is kept at the server site, this involves just a simple check. Whenever possible, the server dispatches read requests immediately, keeping track of any such read requests that it may have sent out by appending them to the copy of the forward list. All this work is being carried out in the *spare time* of the server when it is waiting for the forward list to return to it after all the requests have been serviced. Thus the slightly extra overhead entailed by the server does not decrease its performance.

3.2 LastWrite Read Optimization

The g-2pl-rorw protocol, like g-2pl-robasic services and dispatches read requests immediately when the previous forward list for that data item was read-only, and the waiting list does not contain any writes. Additionally, it also forwards these reads when the previous forward list was a *read-write* (**RW**) list consisting of both read and write requests, and having trailing reads (that is a bunch of read requests at the end of the forward lists), and the last write in the RW list has released its lock and indicated this to the server. The forward list at this instant consists only of read requests, and the server can take advantage of this to forward new read requests immediately.

In this method, the client with the last write in a RW list, if it has trailing reads following it, sends a message TRAILING READS to the server. The server on getting this message, can treat the forward list as if it was read-only, and forward read requests. Though this method involves the overhead of an extra message from the client with the last write, it improves the overall response time relative to g-2pl-robasic, especially in environments with write requests, where such RW forward lists with trailing reads occur frequently. Again, the server is able to fit in the overhead associated with the immediate dispatch of reads in the time when it is waiting for the forward list to return.

3.3 Reordered-read Optimization

The reordering of requests in the waiting list to avoid deadlocks, that is carried out in g-2pl (See previous section) permits reads that come in later to possibly reach the front of the wait list. In such a case, it is possible to forward reads even if a write exists in the waiting list for that data item. Thus read requests are serviced and forwarded immediately either if no writes exist in the waiting list, or if the read request reaches the front of the waiting list due to reordering, and the previous forward list was either read-only, or now consists only of trailing reads. If read requests frequently reach the front of the wait list, there is a possibility of starvation of the writes already in the wait list. To avoid this, only a certain number of reads are allowed to be forwarded immediately per forward list - wait list cycle. The number of read requests that are forwarded immediately by the server are observed to increase, leading to corresponding extra book-keeping for the server, but this is in the server's spare time, when it would otherwise have been idle and unutilized.

3.4 Other Read Optimizations

If the forward list is read-only frequently, then the probability of being able to forward new read request that arrive increases. To ensure that the forward list was read-only often, we can split the forward lists, either (i) forwarding leading reads and keeping the trailing read-write part behind at the server, or (ii) forwarding the leading read-write part and keeping the trailing reads behind at the server, or (iii) a combination of both. However, our simulation experiments have shown that the optimizations that involve splitting the forward list do not provide good performance. It was observed that keeping requests behind at the server increases the average response times, far outweighing any performance advantages that might have been gained by having read-only forward lists more often. For these reason and due to space limitations we will not discuss these optimizations further.

4 Simulation Testbed

In order to evaluate the performance of the s-2pl, the g-2pl and its three read-only optimizations, namely, g-2pl-robasic, g-2pl-rorw, and g-2pl-roreorder, discrete event simulation models of all protocols were developed using the *C* programming language. We consider a data-server database system, with a single server and multiple clients connected by a high speed

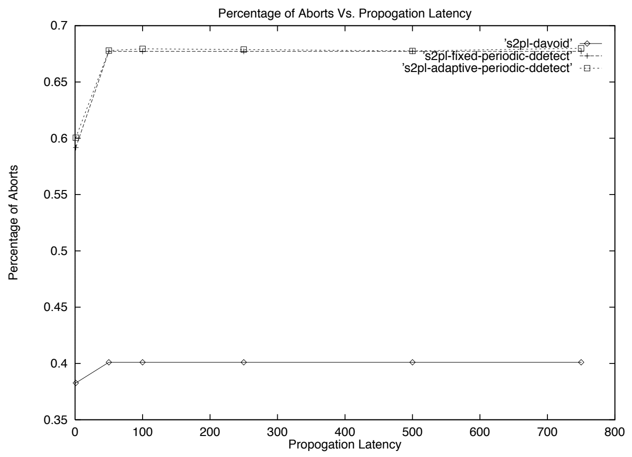

Figure 1: Percentage of Aborts Vs Propogation Latency (50 clients, Read Probability=0.6)

network. In this paper, we make the simplifying assumption that the network latency between any two sites (server-client, client-client) and in either direction is the same, and no site and communication failures occur. Also, we use the terms propagation latency and network latency inter-changeably in this paper.

All clients are assumed to be identical and run transactions that have the same statistical profile. The multi-programming level at each client is assumed to be one. Further, at the end of each transaction, it is replaced with another transaction at that client site after some idle time that is uniformly distributed between a given minimum and maximum values. Each transaction accesses between 1 and N data items uniformly. These data items are drawn from a pool of M data items that reside at the data server. M is purposely kept small to emulate hot data access. Each data access may be of the type read with a given read probability p_r and of the type write with a probability $p_w = 1 - p_r$. The transaction execution is *sequential*, i.e., requests for data items are generated sequentially, with each request being generated only after the previous request has been granted and some think time (for computations) has elapsed. In our model, this computation time is uniformly distributed between a given minimum and maximum values.

In the s-2pl implementation, deadlocks are detected by computing wait-for-graphs and aborting the transactions necessary to remove the deadlocks, which is the typical implementation found in commercial systems. In order to avoid the use of tunable timeouts, we use *deadlock avoidance*, that is, deadlock detection is initiated when a lock cannot be granted. However, we have also implemented *adaptive* deadlock detection

# Servers	1
# Clients	varying
# hot data items	25
Tx execution pattern	Sequential
Multiprogramming level at clients	1
# data items accessed by a transaction	1–5
Percentage of read accesses	0–100%
Network latency	1–750
Computation Time / database operation	1–3
Idle Time between transactions	2–10

Table 1: Simulation Parameters

and compared it with both fixed periodic deadlock detection and deadlock avoidance in order to assert the impact of the selected deadlock detection scheme on our experimental results. It was observed that the deadlock avoidance algorithm provides better performance than the other two schemes (Figure 1). The simulation model assumes that the computation cost at the data server to reorder the forward lists as well as computing the wait-for-graphs is the same. Table 1 summarizes all the experimental parameters and the corresponding range of values of the performance study.

After eliminating the transient phase, each simulation run terminated after 100,000 transactions were committed. For each set of parameters, the simulations were run for 5 different seed numbers in order to obtain statistically significant results. The simulation results presented in the next section have relative precision less than 1%.

5 Performance Analysis

Transaction Response Time

Figures 2 to 7 show the average transaction response time plotted against the network latency for different values of read probability – 0.0, 0.6, 0.8, 0.9, 0.95 and 1.0. The system had 50 clients accessing 25 hot data items uniformly.

For a transaction profile with no reads in it (Figure 2), g-2pl performs upto 11% better for lower propagation latencies(1-50) as seen in LANs or smaller MANs, and upto 20% better for longer latencies(>100) as in larger MANs and WANs, in terms of response times than the s-2pl scheme. The improvement is particularly high for higher propagation latencies. The behaviour of the various read optimizations is iden-

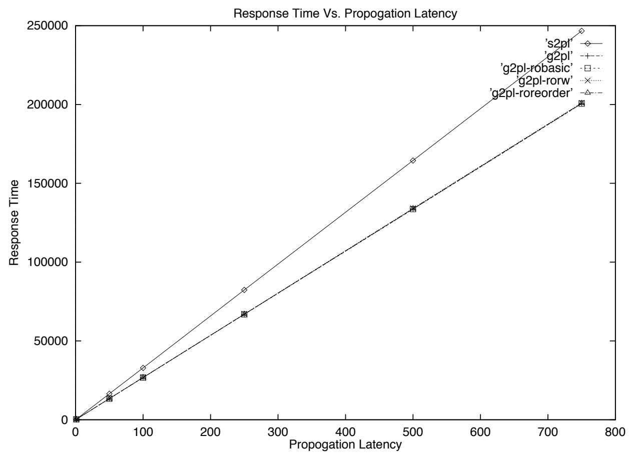

Figure 2: Response Time Vs Propagation Latency for Read Probability=0.0 (50 clients)

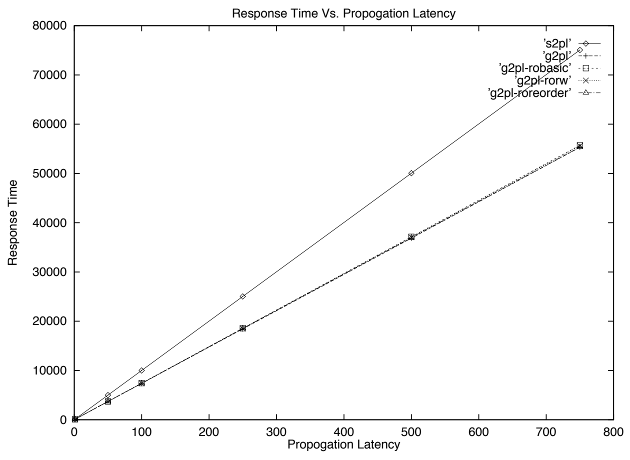

Figure 3: Response Time Vs Propagation Latency for Read Probability=0.6 (50 clients)

tical to that of g-2pl, as there are no reads for this instance.

When there are about 60% reads in the system (Figure 3), g-2pl performs 20% better than s-2pl for lesser latencies, and 25% better for longer propagation latencies. At this stage, the g-2pl read optimizations perform close to g-2pl. As there are an almost equal number of reads and writes in the system, conditions in which we get a read-only forward list, or when the forward-list consists only of trailing reads are quite infrequent. Thus, the gain obtained from the read optimizations if any, is insignificant.

For a higher read probability of 0.8 (Figure 4), g-2pl performs 20% for smaller networks and 28% for larger MANs and WANS better than s-2pl. With 80% reads in the system, the read-optimizations start showing their effects. g-2pl-robasic, g-2pl-rorw and g-2pl-roreorder show improvements of 1.3%, 1.4% and 2.4% respectively over g-2pl. As each of these optimizations adds a further condition to forward reads immediately, we see the gradual increase in performance. With these optimizations, g-2pl performs upto 30% better than s-2pl.

When we observe the results for 90% reads (Figure 5), for lower propagation latencies, g-2pl without any optimizations is slightly inferior to s-2pl. The reason is that more read requests are submitted to the server and at a faster rate that lead to more read-only deadlocks and hence more delays. For the higher propagation latencies however, g-2pl has a response time about 20% better than s-2pl. When the read-optimizations are factored in, g-2pl always performs better than s-2pl: g-2pl-robasic is 17-29.5% better than s-2pl, g-2pl-rorw is 17-29.8% better, while g-2pl-roreorder is 17-30.3% better, with lesser improvements for shorter propagation latencies, and greater improvements for longer latencies. The improved response time is particularly marked for higher propagation latencies.

Similarly when there are 95% reads (Figure 6), g-2pl performs only slightly better than s-2pl, giving maximum improvements of 8%. The addition of the read optimizations however improves performance up to 25.5%, 26%, and 26.5% for the robasic, rorw, and roreorder optimizations relative to s-2pl. The marginal difference in performance between the various read optimizations for these higher read probabilities can be explained to the fact that at higher read probabilites, read-only forward lists occur frequently enough. The conditions in which reads can be forwarded immediately by the latter two optimizations are hence relatively the same as that for robasic.

Finally, when there are only reads in the system (Figure 7), g-2pl normally suffers due to grouping of reads. Not only do we observe read-read deadlocks, but g-2pl has a response time thats up to 70% less than s-2pl. The addition of the read optimizations enables immediate dispatch of reads, and no grouping takes place. Thus, not only are the read-only deadlocks avoided, but g-2pl with the read-optimizations has response times similar to s-2pl. Thus, we see that with the read optimizations, g-2pl **always** performs better than or equal to s-2pl in terms of response time, irrespective of the read probability, getting improvements of up to 30% relative to s-2pl.

Figure 8 summarizes our results for propagation latency of 250 time units. It plots the average response time against different values of read probability. For

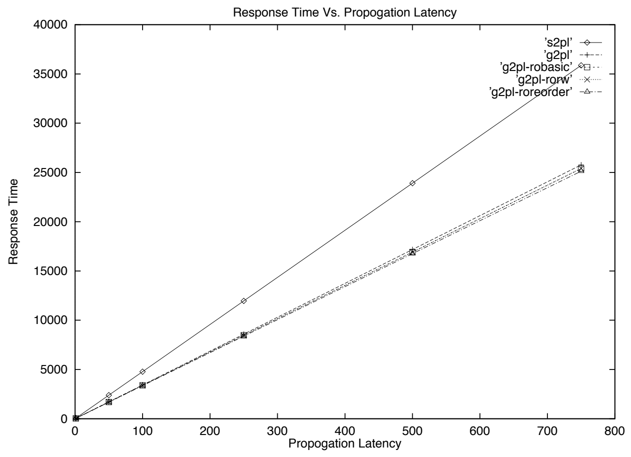

Figure 4: Response Time Vs Propagation Latency for Read Probability=0.8 (50 clients)

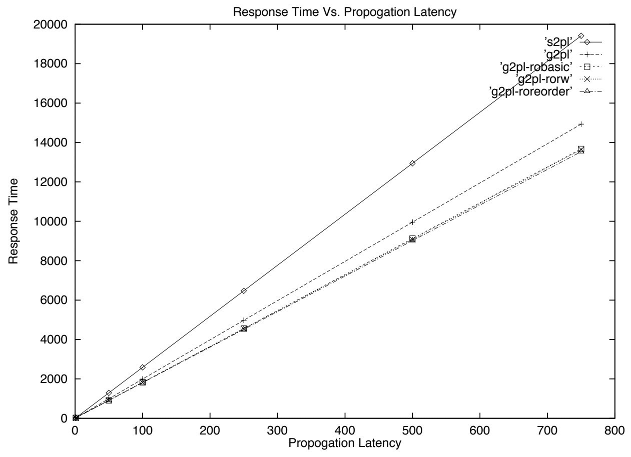

Figure 5: Response Time Vs Propagation Latency for Read Probability=0.9 (50 clients)

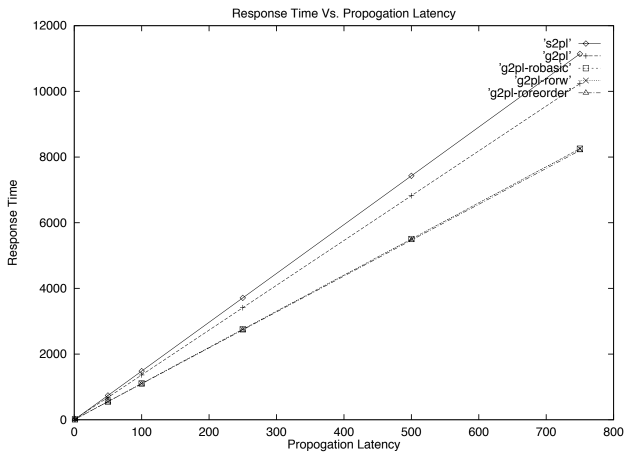

Figure 6: Response Time Vs Propagation Latency for Read Probability=0.95 (50 clients)

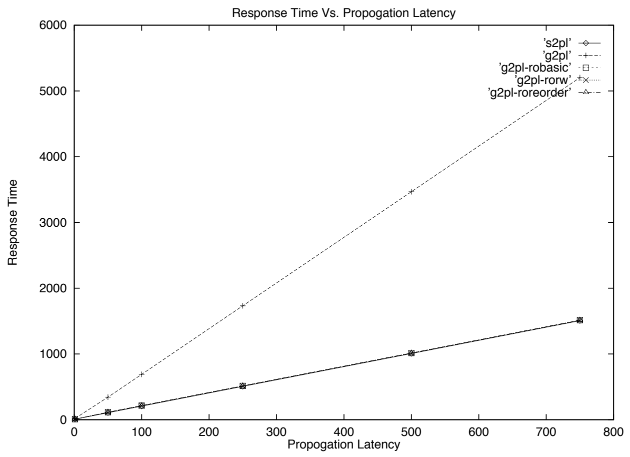

Figure 7: Response Time Vs Propagation Latency for Read Probability=1.0 (50 clients)

high data contention conditions (left part of the figure), g-2pl and its optimizations perform the best, in this example about 18%. For low data contention conditions and more specifically for read probabilities higher than 95%, the g-2pl read optimizations still perform better than s-2pl and exhibit the same performance under read-only conditions.

Thus far, we have demonstrated that g-2pl and its optimizations scale with the network latency and for all data contention conditions. In order to assert the scalability properties of g-2pl and its optimizations with repect to the number of clients, we studied how the average response time increases with the number of clients for a specific data contention conditions. Figure 9, for example, shows the average response time for propagation latency 500 units and read probabil-

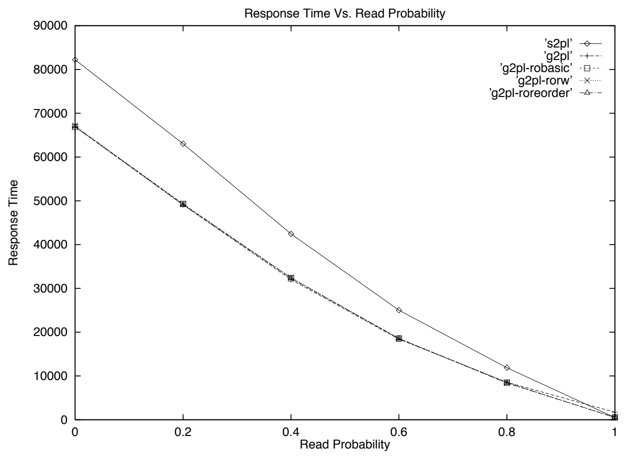

Figure 8: Response Time Vs Read Probability (50 clients, propagation latency = 250)

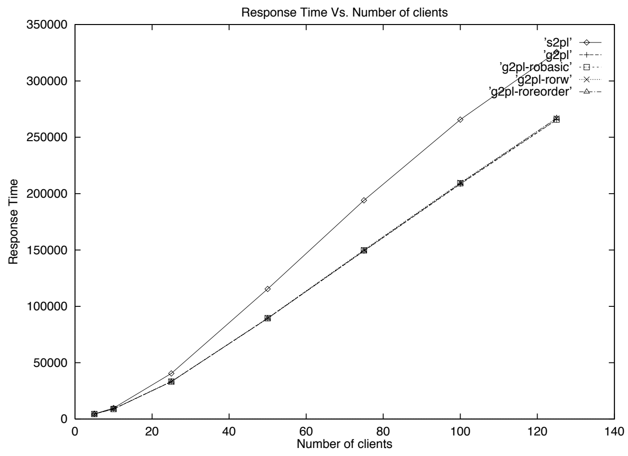

Figure 9: Response Time Vs Number of Clients (Read probability=0.25, propagation latency = 500)

ity of 0.25. As it was expected, both s-2pl and g-2pl and its optimizations grow almost linearly with the number of clients. However, it has been observed that the slope of the graphs for the g-2pl variants is always smaller than that of s-2pl which clearly indicates that the g-2pl variants also exhibit better scalability with respect to the number of clients.

System Throughput

Figure 10 compares the relative throughput of the different protocols as the number of clients is increased for a read probability of 0.75 and for propagation latency of 500. The s-2PL protocol thrashes around 8-9 clients. Although g-2pl grows slower than s-2PL, it scales much better, and its throughput keeps increasing up to 600 clients. As the number of clients increases, the grouping also increases, and the bene-

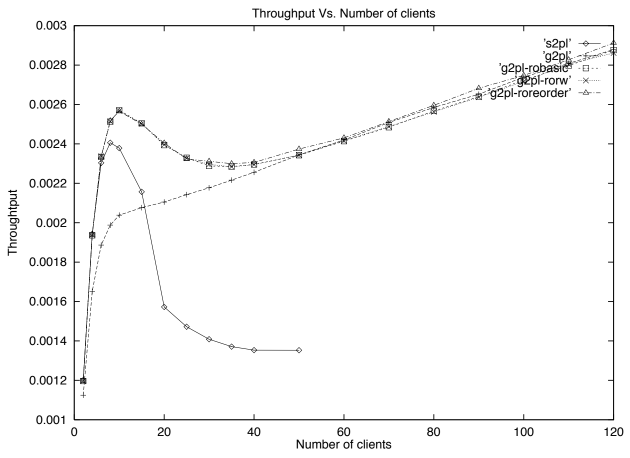

Figure 10: Throughput Vs Number of Clients (Read probability=0.75, propagation latency = 500)

fits associated with this grouping causes throughput to improve too. Similar behavior has been observed for all read probabilities except for read probability 1.0. when in g-2PL the reads are penalized due to grouping. Thus, for read probability 1.0, the roles are reversed and s-2PL exhibits better throughput up to 650 clients.

Clearly, the three g-2PL read optimizations eliminate any read penalty. For high read probabilities, they service and forward read requests immediately more often, and thus their initial behaviour tends more towards s-2pl for 30 or fewer clients. However, they exhibit better behavior than s-2PL, showing a dip around 12-13 clients and 8% higher throughput at peak. Beyond 30 clients, it seems that as the number of clients and the effects of grouping increase the behavor of the three read optimizations matches and improves slightly over g-2pl. Of these optimizations, g-2PL-roreoder seems to exhibit the best performance.

6 Conclusions

Most global Internet systems have a pressing need for scalability. Since the demand for services is extremely bursty, it is hard to dimension the capacity of servers, network and storage elements needed for a specific service. Scalable protocols that are able to effectively hide the network latency and other processing delays will go a long way to mitigate the problems faced by various service providers. Towards this end, several data shipping access protocols have been proposed including our *group two-phase locking* (g-2PL) protocol.

While the g-2PL protocol outperformed other comparable protocols under high data contentions, its performance was comparatively poorer when the data contention was low, that is, in predominantly read-only situations. Since a majority of transactions on the Internet fall in the read category, it is clear from a practical standpoint that any scalable data sharing protocol must address this scenario well. With this motivation, in this paper, we present three read optimizations to the original g-2PL protocol that enables it to exhibit scalable performance under the entire range of data contention levels, at different network latencies and with a larger number of clients. The results of our experiments confirmed that the g-2PL protocol with its read optimizations consistently outperforms the s-2PL protocol, while adding only a minor level of complexity to the system. The performance gains in response time are quite large, at about 30% under the server-based two phase locking protocol.

It seems that the cooperative aspect of g-2PL, in conjunction with its superior performance over other data shipping protocols, can be used as a core data access protocol in Internet peer-to-peer (P2P) systems. We are currently extending this work for such P2P environments.

Acknowledgments

This work has been partially supported by NSF awards NCR-9624125, NCR-9702389 and IIS-9812532.

References

[1] S. Banerjee and P. K. Chrysanthis, "Network Latency Optimizations in Distributed Database Systems," *Proc. of the Intl. Conf. on Data Engineering*, pp. 532–540, 1998.

[2] S. Banerjee and P. K. Chrysanthis, "Group Two-Phase Locking: A Scalable Data Sharing Protocol," *IEICE Transactions on Information Systems*, Vol. E82-D, No. 1, 1999.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[4] M. Carey and M. Livny, "Conflict detection tradeoffs for replicated data," *ACM Transactions on Database Systems*, vol. 16, no. 4, pp. 703–746, 1991.

[5] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques For Main Memory Database Systems," *Proc. of the ACM SIGMOD Intl. Conf. On Management of Data*, pp. 1–8, 1984.

[6] K. P. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, pp. 624–633, 1976.

[7] M. J. Franklin and M. Carey, "Client-Server Caching Revisited," *Distributed Object Management* (T. Ozsu, U. Dayal, and P. Valduriez, eds.), pp. 57–78, Morgan Kaufmann Publishers, 1993.

[8] M. J. Franklin, M. Carey, and M. Linvy, "Transactional client-server cache consistency: Alternatives and performance," *ACM Transactoions on Database Systems*, 1997.

[9] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt, "Crash Recovery in Client-Server EXODUS," *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 165–174, 1992.

[10] D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *IEEE Database Engineering*, vol. 8, 1985.

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a distributed File system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[12] L. Kleinrock, "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Mag.*, vol. 30, pp. 36–40, 1992.

[13] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communication of the ACM*, vol. 34, no. 10, pp. 34–48, 1991.

[14] C. Mohan and I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment," *Proc. of the 17th Intl. Conf. on Very Large Databases*, 1991.

[15] T. Nguyen and V. Shrinivasan, "Accessing Relational Databases from the World Wide Web," *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 529–539, 1996.

[16] S. Venkataraman, J. F. Naughton, and M. Livny, "Remote Load-Sensitive Caching for Multi-Server Database Systems," *Proc. of the Intl. Conf. on Data Engineering*, 1998.

[17] Y. Wang and L. Rowe., "Cache Consistency and Concurrency Control in a Client/server DBMS Architecture," *Proc. of the ACM SIGMOD Intl.*

Conf. on Management of Data, pp. 367–376, 1991.

[18] K. Wilkinson and M. A. Neimat, "Maintaining Consistency of Client-Cached data," *Proc. of the 16th Intl. Conf. on Very Large Databases*, pp. 122–134, 1990.