PANOPTICON: An Omniscient Lock Broker for Efficient Distributed Transactions in the Datacenter

Serafettin Tasci and Murat Demirbas
Computer Science & Engineering Department
University at Buffalo, SUNY

Abstract

For datacenter applications that require tight synchronization, transactions are commonly employed for achieving concurrency while preserving correctness. Unfortunately, distributed transactions are hard to scale due to the decentralized lock acquisition and coordination protocols they employ. In this paper, we show that it is possible to achieve scalability for distributed transactions by using a lock broker architecture, and present the design and development of such a framework, called PANOPTICON.

Panopticon achieves scalability by divorcing locks from the data items and striving to improve lock access locality. A lock can be hosted at the lock broker or at a different server than the server that hosts the corresponding data item. The lock broker mediates the access to data shared across servers by migrating the associated locks like tokens, and in the process gets to learn about the access patterns of transactions. We show that the broker can drastically improve the lock access locality and, hence, the performance of distributed transactions by employing simple rules.

Our experiments show that Panopticon performed significantly better than distributed transactions as the the number of data items and number of servers involved in transactions increase. Moreover, as the history locality (the probability of using the same objects in consecutive transactions) increase, Panopticon's lock migration strategies improved lock-access locality and resulted in significantly better performance.

1 Introduction

Concurrent execution is a big challenge for distributed systems programming and datacenter computing in particular. For embarrassingly parallel data processing applications, concurrency is boosted and scalability is achieved by adding more servers, as in MapReduce [11]. However, for applications that require tighter synchronization, such as large-scale graph processing [13, 20], distributed scientific simulations [4], and backends of large-scale web-services, boosting concurrency in an uncontrolled/uncoordinated manner wreaks safety violations. In these applications, concurrent execution needs to be coordinated to prevent latent race conditions and synchronization bugs.

Using locking primitives and employing transactions are the two most common techniques to deal with tight synchronization requirements [5, 16]. Unfortunately, locking is manual and hence is error-prone. If the developer misses to place a lock where needed, safety is violated —these kind of bugs may be latent heisenbugs that are hard to debug. On the other hand, if the developer inserts unneeded locks, the performance of the system suffers due to the unnecessary synchronizations.

Transactions provide a nice abstraction to write code, providing serializability guarantees while allowing concurrency under the hood. Since transactions are easy to use, they are employed by modern distributed systems including Google Spanner [8], Google Megastore [3], Microsoft Azure [17], and AWS RDS [2]. Unfortunately, transactions have gained a reputation for being unscalable. We identify the following two issues as main problems with the scalability of distributed transactions:

- Distributed transactions waste time in coordination. When several data items need to be locked at the same time, test-and-set approaches are inapplicable, and distributed coordination for setting locks (such as two phase locking and two phase commit) do not scale as their costs grow quickly with respect to the number of data items and servers involved in the transaction.
- There is a big latency difference between accessing a server-local item versus item that is stored across the cluster, however, modern datacenter computing systems focus on consistent hashing and load-balancing

of data to the servers and ignore locality when assigning data for storage. When locks are coupled and tied to the data, this penalizes transaction latencies severely, especially in the lock acquisition phase. Even after several repetitions of the same transaction, each time the remote lock-access costs are paid again and again.

In this paper we argue that it is possible to address the above problems and achieve a scalable transactional system. We propose a lock broker architecture, called PANOPTICON. Keeping a centralized lock broker eliminates the complexity and cost of distributed transaction solutions, and enables the lock broker to see all transaction access patterns so that the broker can improve lock access locality of transactions by migrating locks when appropriate.

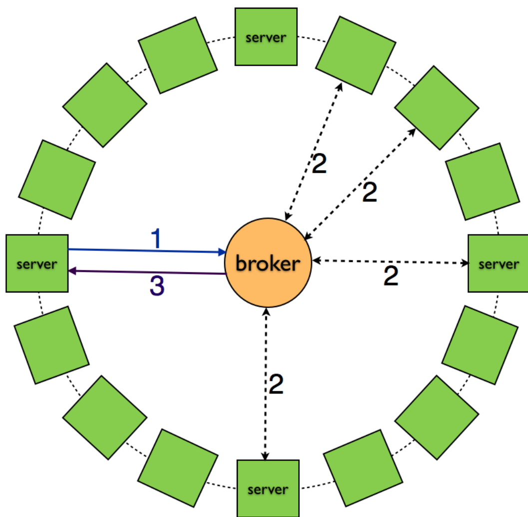

Figure 1: Panopticon lock broker

Differing from centralized lock service solutions such as Chubby [6] or Zookeeper [16], Panopticon does not keep all the locks in the broker. Dealing with large scale data, such as in Spanner [8], it is impossible to fit all the locks in the broker and have all centralized locking. And, even when that is possible, it is not desirable since it kills all opportunities for local access: None of the servers can have local transactions in that setup. Panopticon employs the broker as a cache for locks that receive across-server access. The broker migrates the locks that get accesses mostly from one server to that server to improves lock-access locality further.

1.1 Overview of Panopticon's distributed transactions

Panopticon maintains a lock for every data item (i.e, record, key value tuple) stored in the system. By default, the lock is kept where the data is, and this is advantageous for improving lock-locality and enabling server-local transactions without the need for contacting the broker. The server contacts the lock broker only if data at other servers need to be accessed as part of a transaction.

The lock broker coordinates across-server sharing of the data in distributed transactions by migrating the corresponding locks like tokens. A server can request a lock anytime from the broker, and only from the broker. The broker gives the locks in a first-come first-serve manner. If the requested locks are at the broker, the broker responds immediately. Otherwise, the broker requests the locks from the corresponding servers first and forwards them to the requester when they are made available. The broker can request a lock back anytime from a server, and the server complies. (If the server is currently using the lock in an executing transaction, the server returns the lock after the transaction completes.)

As the centralized authority for mediating access to data, the broker learns about the access patterns of transactions at runtime and manages the migration of locks to servers in a way that improves lock access locality. The lock broker, however, is oblivious to the state of the transactions, and the servers are the ultimate transaction managers. Transactions are initiated and executed by the servers distributedly after checking that all the locks are available at the server.

When transactions involve multiple data items, this centralized lock broker solution gains an edge over the traditional distributed transaction processing. Traditional distributed transactions employ two phase locking to prevent deadlocks, which requires that the server initiating the transaction to contact the other servers for locks in increasing order of locks. Instead of contacting other servers trying to acquire locks in increasing order in a serial manner, it is more efficient to go to the broker and test/set all the locks at once. In Panopticon, the lock request is sent at once to the broker, and the broker takes care of deadlock prevention.

1.2 Contributions

Panopticon proposes a novel approach to distributed transaction processing. Panopticon achieves scalability by divorcing locks from the data items and employing lock caching at the broker and lock migration to servers to improve lock access locality. We present Panopticon's novel

lock broker architecture in Section 2. There we detail the broker operations, lock migration rules, and optimizations such as batch locking, and lazy unlocking.

We develop and build Panopticon leveraging the Hazelcast [15] platform, a popular lightweight open-source in-memory data grid for Java. Hazelcast uses traditional distributed transaction processing with decentralized two phase locking protocol. We build on Hazelcast to implement the Panopticon lock broker architecture. We present our implementation of Panopticon in Section 3 and then use this implementation to compare and contrast Panopticon's improvements over Hazelcast distributed transaction processing in Section 4.

In our experiments we identify under what conditions (e.g., how many data items per transaction, contention rate, number of servers, etc.) the Panopticon lock broker solution provides the most benefits. We also investigate tradeoffs of lock caching and migration strategies under varying workload characteristics. Using these observations, we devise efficient lock caching and migration strategies between the broker and the servers. Our experiments show that Panopticon performed significantly better than distributed transactions as the the number of data items and number of servers involved in transactions increase. Moreover, as the history locality (the probability of using the same objects in consecutive transactions) increase, Panopticon's lock migration strategies improved lock-access locality and resulted in significantly better performance.

We discuss how Panopticon can be extended to across datacenter deployments using hierarchical composition in Section 5. We compare and contrast Panopticon with other work on distributed transaction processing in Section 6.

2 Panopticon Lock Broker

2.1 Tradeoffs and lock migration

As we discussed earlier, maintaining all the locks stored at the broker is not desirable because it kills all the lock-access locality for the servers. On the other extreme, if we do not keep any locks in the broker, and make the servers keep the locks, there is a different disadvantage associated with that: If a server w needs one of the locks held by another server y, then it suffers extra delay for lock acquisition: Firstly, w sends a lock request to the broker. Then the broker will forward this request to y. Finally when the broker gets the lock from y, it forwards the lock to w.

Our solution strives to find the sweet point in this tradeoff spectrum. We notice that there can be three types of locks hosted at the master:

1. locks that receive across-server accesses,
2. locks that receive repetitive access from same server,
3. locks that receive no access for a long-time.

It is best to host *type 1* locks (locks that keep receiving across-server accesses) in the lock broker. And it is best to assign the *type 2* locks to the requesting server to avoid the overheads of repetitive requests from that server to the broker. We discuss the determination of the sweet point in the tradeoff between *type 1* and *type 2* locks next. [1]

In Panopticon, the broker gets to observe all transaction access patterns at runtime so it can differentiate between *type 1* and *type 2* locks given some rules for cut points. We use the following rule of thumb for declaring a lock to be of *type 2* and migrating that lock to a server: If two consecutive requests for a given lock l (held at the broker) comes from the same server w, then the broker migrates lock l to server w. From that point on w treats l as its local lock, the lock locality of w is improved with this, since w does not need to contact the broker for l again.

Note that this is not a permanent assignment. Later if another server y requests l, the broker migrates l back to itself, and gives y the lock. At this point, l is treated again as a *type 1* lock. When y is done with l, l is continued to be hosted at the broker (that is, until the 2-consecutive rule is satisfied and l is migrated to another server).

2.2 Transaction execution

The broker is stateless about the transactions. The broker maintains per lock information, not per transaction information. The servers are the transaction managers and they maintain the transaction information.

When a server initiates a transaction, it requests locks in batch as we explain in the next subsection. The broker coordinates assigning/delivering of the requested locks in a first come first serve basis. And when processing a lock request for a transaction, the broker assigns the locks in an increasing order to prevent deadlocks. The broker sorts the lock requests to form a total order based on the data item ID. When processing in this increasing order of locks, if the broker holds the lock, it forwards it to the requesting server. If the broker does not have the lock, it adds this server's name to the request-queue of the lock,

[1] For space-saving at the master, we can employ least recently used (LRU) policy to expel *type 3* locks (locks that have not seen an access for a long-time) back to the original host (the server that hosts the corresponding data item).

and forwards the lock requests to the server that holds the lock. When the lock becomes available, the broker will forward this lock to the server at the head of the queue maintained for that lock.

The server, as the transaction manager, is also responsible for determining when to enter the transaction. When the server checks and finds that it has gotten all the locks, it starts the transaction and *authoritatively-owns* those locks, deferring the requests for these locks until the transaction ends. Within the time frame of initiating the transaction and entering the transaction, the server may have some of the locks but it may not authoritatively-own those locks. For example, if a server requires locks of multiple data items $\{l_1, l_2, l_3, .., l_n\}$ for a transaction, it cannot authoritatively-own l_i until all data items l_j such that $l_j < l_i$ are locked in the current transaction. If the server is waiting for lock l_3 and has l_1, l_2, l_4, then l_4 is not authoritatively-owned by the server: if the broker asks l_4, the server will have to return l_4 to the broker. On the other hand, l_1, l_2 are authoritatively-owned by the server. This way of managing the locks guarantees that livelocks as well as deadlocks are avoided.

2.3 Batch locking and lazy unlocking

In Hazelcast transactions, two phase locking is employed to prevent deadlocks. Two phase locking requires that the server initiating the transaction needs to contact the other servers for locks in increasing order of locks. The server cannot contact another server until the current requested lock is acquired. Therefore, if the transaction needs to get the locks of multiple data items, this forces lock acquisition to be serial in nature and causes a significant increase in transaction time.

In Panopticon the servers are not prone to this problem. The servers can make the requests for all locks in batch and at once, because the broker takes care of deadlock prevention. The request is sent at once, but the broker processes the requested locks in the increasing order of lock IDs, and assigns the server the locks or adds the server to the wait queues, as we discussed above. By employing batch requests to the broker, Panopticon avoids incremental lock requests in Hazelcast distributed transactions and gains a big advantage in transaction execution time.

After a transaction is finished, the server needs to unlock the data items, which means returning the locks back to the broker. In this phase we propose an optimization where the server *lazy unlocks* the locks. Lazy unlocking means that the locks are released locally, but not transmitted back to broker until δ time elapses, where δ is empirically determined. Lazy unlocking provides efficiency benefits for cases when the server needs to access the same data items used in the terminated transaction immediately in the next transaction. Recall that if the same lock is accessed by the same server two times in a row, that lock is migrated from the broker to that server. And the lazy unlock provides benefits between the first and second consecutive requests of a lock by the server. Instead of returning the lock back to the broker only to request it back afterwards, the lazy unlock mechanism provides a small grace period at the server to avoid that inefficiency. This is optimization verifiably safe because if lazy-unlocked data-lock got requested, it is given back to the broker.

3 Implementation

To implement centralized locking in Hazelcast, we edited the Hazelcast source code for modifying the automated locking of variables in transactions. In this modified Hazelcast version, we introduced the *lockAll()* and *unlockAll()* methods for contacting the broker. Hazelcast does not differentiate between read-locks and write-locks and use one uniform lock for transaction items, and in Panopticon we also follow that design.

3.1 Broker actions and data structures

All instances in a Hazelcast cluster have unique IDs, and we set the instance with ID 0 (i.e., denoted as the oldest node and special node in Hazelcast) as the broker and the other instances are servers. The broker does not perform computation and responsible solely for lock management. It has a *lockTable* to keep the current owner for each data item. In addition, it has a *requestTable* that keeps track of the requests for each lock. When a lock request arrives, if the lock is already available at the broker, the broker gives the lock to the requester. If not, it inserts the requester ID to the *requestTable* and forwards a request to get the lock from the hosting server (only if a request has not been sent before). If there are multiple requests for a data item, the *requestTable* keeps a FIFO queue for each data item. The broker also has a *leaseTable* to keep track of which locks it has migrated to which servers.

3.2 Server actions and data structures

A server keeps the list of locks it own in a list called *lockList*.[2] Whenever the server needs a new lock, it first checks the *lockList* and send a lock request to the broker only if it does not find the lock in this list. In addition the server keeps a boolean *requestList* to keep track of the

list of requests for the locks it has. Since the lock exchange occurs only between the broker and a server, the servers do not need to know which server requested the lock. Whenever a request comes from the broker, they just hand over the lock to the broker if it is available.

Whenever a server exits a transaction and calls unlockAll, it checks its *requestList* and gives the requested locks to the broker. If some of the locks are not requested, the server can keep them if the broker gave a lease for the lock, in other words if the broker migrated the lock to this server.

3.3 Communication and messaging

Messaging between the servers and the broker is done via ITopic publish-subscribe mechanism in Hazelcast. We use $n + 1$ topics where n is the number of servers. For server-to-broker communication we used a shared channel called *toBroker*. However broker-to-server communication uses separate channels for every $server_i$ to ensure messages are published only to relevant servers. Each server and the broker runs a message listener thread to continuously listen for incoming messages on the registered ITopic. Whenever a new message arrives, message is parsed and handled based on the message type.

There are three types of messages in Panopticon: A *request* message is used to send a lock request to the broker or server holding the lock. Of course a server cannot send request messages to other servers directly; the broker does it on behalf of the server. A *reply* message is sent, to submit the lock to the requester when the lock becomes available. (Note that these messages might only be sent between a server-broker pair, never a server-server pair.) Finally, a *lease* message is used by broker to give/cancel leases of locks. For this purpose broker keeps a consecutive request list called *conseqList* that holds the consecutive requests to data items along with the id of the requesting server.

Finally, Panopticon does not assume reliable channels and is robust against message loss. To handle message losses, Panopticon employs message reply timeouts. If a lock request message is not answered in a specified time, the server resends the lock request. Side effects of resending are avoided since the lock request and reply messages are idempotent.

[2]Note that these data structures are Lists, compared to the Table data structures in the broker, because for these locks, the other party is clear and unique the broker. On the other hand, the broker needs to keep the other party for a lock explicitly in its table because the other party can be any of the servers.

4 Experiments and Evaluation

4.1 Setup

To evaluate the performance of Panopticon, we performed experiments on AWS EC2 using medium Linux instances, which have two EC2 compute units and 3.75 GB of RAM each. [3]In our experiments, one instance is designated as the broker, and the remaining instances are servers. In all the experiments, and the transaction time is provided to be an average of 1000 such transactions.

We compare Panopticon with the decentralized two phase locking based Hazelcast transactions. In our basic Panopticon implementation, to save messages, we made the broker batch lock-replies and grant them together in one message to the server. Panopticon-L denotes the basic Panopticon framework with lazy unlocking optimization that we described in Section 2.3. We also performed experiments with Panopticon-S, which improves Panopticon with read staging. In Panopticon-S the broker sends lock-replies to the server as locks become available. And when a server receives a lock, it reads the data item immediately to stage a copy of the data-item at its cache even before it enters the transaction (by receiving all the locks). Panopticon-SL denotes the Panopticon-S version with lazy unlocking optimization.

We measure the effect of many parameters in Panopticon. Pr_Hist is the probability of using the same objects in consecutive transactions. For example $Pr_Hist = 0.7$ means that if a transaction uses 100 shared objects, 70 of them will also be accessed by the previous transaction from the same server. N_conseq is used to determine the time for giving a lease. If the same server makes N consecutive requests to an object without any other server requesting it in the meanwhile, then the lease of the object is given to the server and the server can keep it until another request comes. Otherwise, it immediately returns the lock to the broker after its transaction finishes.

In the experiments, we selected N_conseq as 2, Pr_Hist as 0.9, the total number of data items as 1024, and the number of data items requested in transaction as 16, if not stated otherwise.

[3]We used medium instances to show that the lock broker is able to scale and perform better than Hazelcast distributed transactions on modest hardware. We predict that using AWS large and xlarge instances would benefit Panopticon more, as they would provide parallel send and receive opportunities at the broker. Hazelcast distributed transactions, on the other hand, would still need to serialize send and receive due to two-phase locking even when parallel send and receives are available.

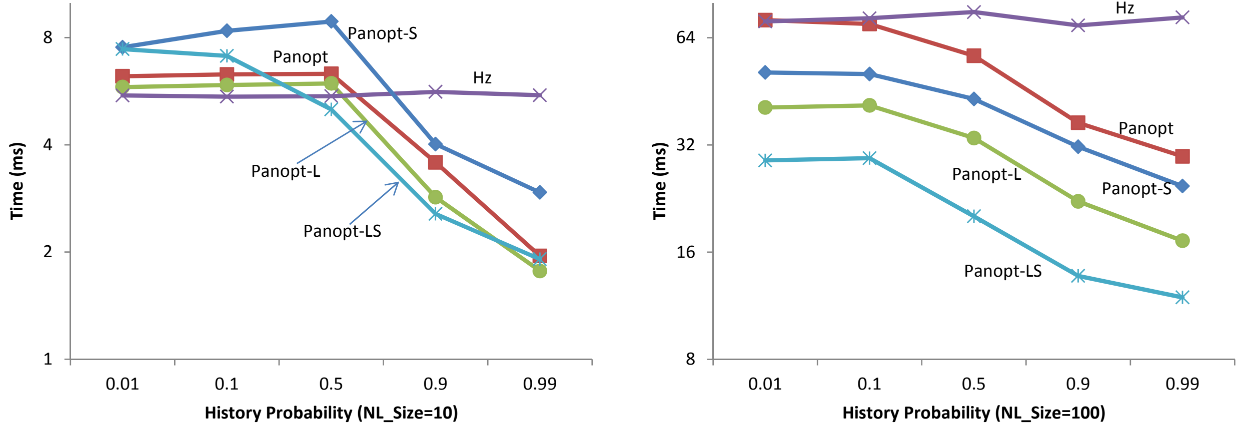

Figure 2: Change in time as the probability of selecting the same items in consecutive transactions increases

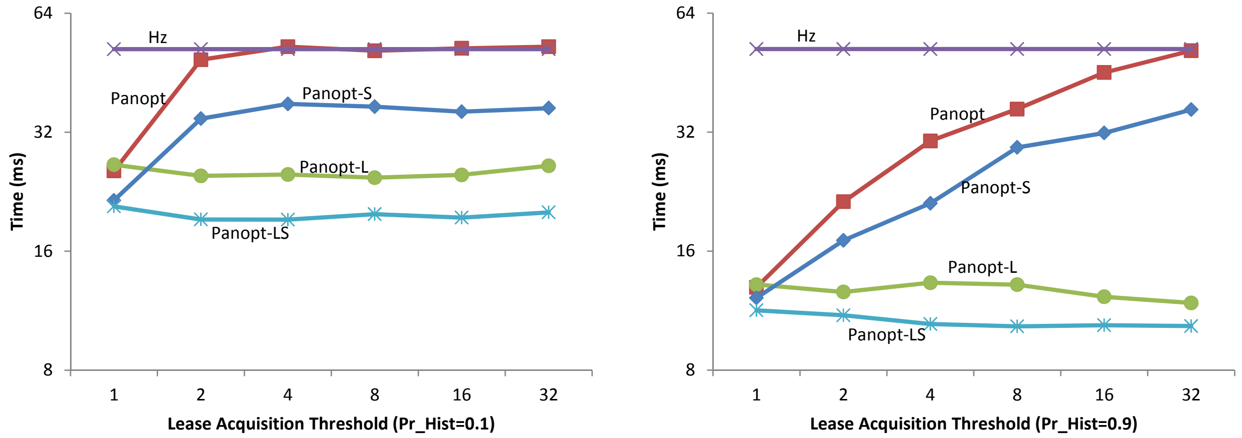

Figure 3: Change in time as the required number of consecutive requests for lease acquisition increases

4.2 Experiment results with 4 servers

In our first set of experiments, we measured the effect of increase in the probability of selecting the same items in consecutive transactions. Note that Hazelcast does not exploit the history of accesses. Therefore the results remain stable in Hazelcast regardless of the history probability. Figure 2 shows that when the number of objects in a transaction is low (i.e. 10), decentralized locking performs slightly better than Panopticon in low degrees of consequent access. On the other hand, if the history probability is high, Panopticon acquires the edge and the gap between the methods increases as the probability increases. When the number of objects in a transaction is high (i.e. 100), this difference becomes more significant.

In our second set of experiments, we evaluated the effect of N_conseq, the required number of consecutive requests for lease acquisition. If this number equals 1, servers will always keep the locks they acquired unless they receive a new request for this lock. If this number is very high, servers will probably never send enough consecutive requests to acquire the lease of the lock. As a result, locks will always be returned to the broker after a transaction completes in a server.

In Figure 3 we see the effect of this number for history probabilities of 0.1 and 0.9. When the probability is 0.1, the chance of getting a lease vanishes quickly causing Panopticon to perform similar to Hazelcast when $N_conseq \geq 2$. On the other hand, when $Pr_Hist = 0.9$, servers will continue to access the same objects for more consecutive transactions. As a result, even if the N_conseq increases, a significant amount of leasing occurs even if N_conseq increases.

Note that when $N_conseq = 1$ Panopticon performs

very similar to Panopticon-L since both methods will cache every object after the transaction completes. In addition we can observe that Panopticon-L is robust to N_conseq due to two advantages: Firstly, even if it does not have the lease, by waiting until the next transaction starts, it can avoid unnecessary return of locks to broker. Secondly, in lazy unlocking writes in a transaction are written to the local cache instead of the distributed object store. As a result, transaction duration becomes shorter decreasing the time required by other servers for getting the object locks that were being used by this worker.

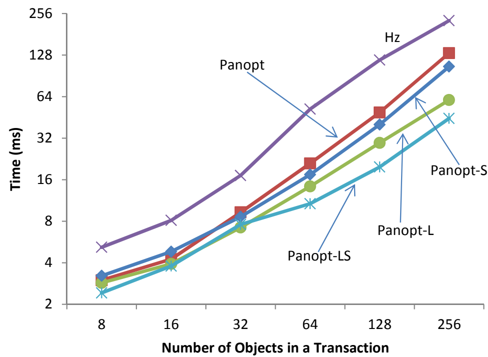

Figure 4: Comparison of Panopticon with decentralized locking as the number of locked items in a transaction changes

Next, we varied the number of data items locked in a transaction to test the effects of contention. Naturally, as the number of locked items increase, the duration of the transactions and also the contention for locks increase in all methods. However, as Figure 4 shows, Panopticon scales better than decentralized locking as the number of locked objects increase due to its batch locking and non-busy lock holding capabilities. In addition, in all experiments we have consistently observed that lazy unlocking improves the performance of Panopticon. This improvement becomes more evident as the number of locked objects in a transaction increases.

4.3 Scalability results

To evaluate the scalability of Panopticon, we keep the number of objects in a transaction constant and measure the total time it takes for 1000 transactions with an increasing number of servers. Each transaction reads and writes 64 objects from a pool of 1000 total objects shared across the cluster. Note that since the total number of objects is constant, the lock contention among servers increases significantly as more servers are employed.

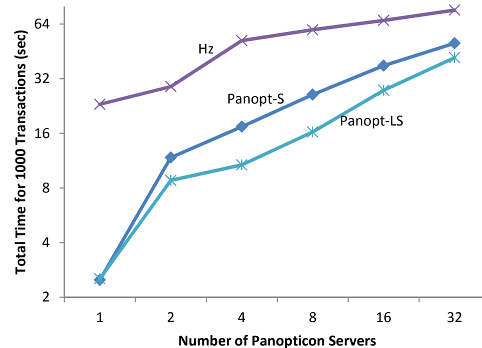

Figure 5: Scalability of Panopticon as the number of servers in the cluster increases

Figure 5 shows that Panopticon performs remarkably well when the contention is low thanks to the data and lock caching mechanisms in Panopticon. When the number of servers increases, Panopticon scales linearly preserving its edge over decentralized locking all the time. In this figure, we did not include Panopticon and Panopticon-L since they are consistently outperformed by their counterparts, Panopticon-S and Panopticon-LS, that use read staging.

In the current version of Panopticon, we manage broker-server communication in the cluster via ITopic publish-subscribe mechanism in Hazelcast. After analyzing Panopticon logs, we noticed that ITopic becomes the communication bottleneck as the number of messages in the system increases with the increasing number of servers. In the next version, we plan to employ more scalable communication primitives between Panopticon servers and broker.

5 Discussion

Here we discuss some of our design decisions and some extensions/improvements to Panopticon.

5.1 Scalability and extension to multiple brokers

The single lock broker can scale well in Panopticon, since it is not contacted for heavy-weight operations and does not keep track of transaction state information. As for the space requirements are concerned, the broker is also light-weight. The broker does not host all the locks; it just hosts the locks that receive across-server accesses. Moreover, the broker needs to keep location pointers only for

locks that are not in their default hashed home. If a lock is hosted at its original hash function home, where the data is also hosted, then the broker does not need to remember the lock location since it can use the hash function to find it.

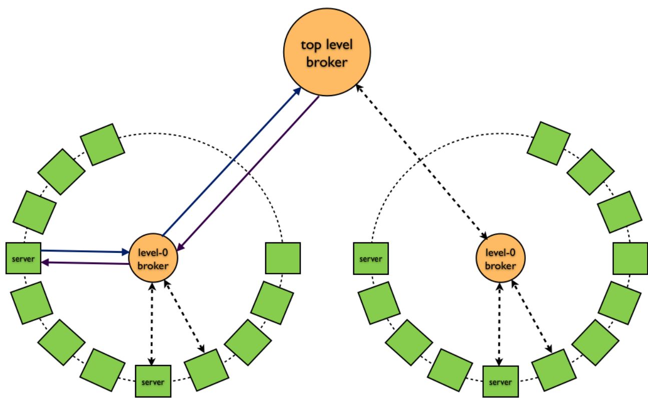

Figure 6: Hierarchical composition of Panopticon lock brokers

In order to give more scalability and avoid bottlenecks when using extremely large number of servers and locks, we can employ a hierarchical composition of the brokers. For this we have k level-0 lock brokers each overseeing a cluster of servers and a top-level lock broker overseeing these k lock brokers. In this scheme each level-0 broker is responsible for brokering the locks for a nonoverlapping range of the key-value space (rather the locks corresponding to the key of the data), and will not honor requests for keys/locks outside their range. The level-0 brokers do not need to coordinate with each other directly; they will coordinate with the top-level broker using the same Panopticon protocol that manages coordination of the servers with a broker. In this case, the level-0 brokers take on the role of servers for the top-level broker, and take on the role of tracking information for across cluster (i.e., across level-0 broker boundary) transactions.

Using hierarchical composition extends scalability of Panopticon. With such a setup, we can make Panopticon manage extremely large lock spaces which may not fit into the memory of a single broker. Moreover, using hierarchical composition of brokers at different datacenters, the Panopticon system can provide a partial answer to the across-datacenter/WAN transactions problem. Providing an efficient and complete system for across-datacenter transactions remains part of our future work.

5.2 Timeless leases and dealing with partitions

Panopticon does not employ timed leases. A lock is owned by either the broker or is migrated to a server, and if it is the latter, the broker can request the lock back anytime. Thus, timed leases are not needed for the functionality of Panopticon. But would timed leases be required in the presence of partitions?

We find that, even without timed leases, we are able to devise simple ways for dealing with availability and consistency to the face of partitions by using broker and server timeouts. For this, we exploit the broker-based architecture of Panopticon.

While there is no global agreement on a partition in a general distributed system setup (some nodes may detect a partition while others do not), the Panopticon setup simplifies the partition detection and handling significantly. Detection is very simple and there are no dissenting opinions on it, since it is dictated by the broker. Furthermore, unlike a general distributed system setup which is prone to arbitrary shaped partitioning, the partitions in the Panopticon are always simple and uniform. There are only 2 cases for a partition: 1) the broker is in the partition, and 2) the broker is not in the partition. We call the partition with the broker as the main partition. If the broker is not in the partition, this means the server is partitioned away as an isolated single node and we call this as a single partition.

While we do not include fault-tolerance to partitioning in the current version of Panopticon, we plan to implement it in the next version using this simple idea. If the broker requests a lock from a server and cannot contact the server (because of the partition or server being down), the broker creates that lock and starts hosting it. Transactions continue using replicated (backup) copies of that data item that fall in the main partition. A partitioned server can also detect this when it cannot reach the broker. On discovering this, the isolated/partitioned server stops using the locks. Note that for this solution, we do not need timeout per lock, rather timeout per server.

5.3 Further optimizations at the broker

Incorporating priorities. It is undesirable to have a transaction that is tried several times without success, while other transactions are executing without hick ups. After the transaction times out and aborts the first time, we would like to assign this transaction a priority so it can complete. In Panopticon, we can design a simple solution to maintain/track priorities of the transactions. If the transaction has waited and aborted earlier, we can increase its priority next time. Then we can keep prior-

ity ordered queues for lock-request/waiting at the broker. This ensures that, if a transaction rolls-back due to waiting last time, its chances of succeeding increases significantly next time. We can also this to prioritize certain critical transactions over non-critical transactions.

Congestion control. Since the broker gets to observe every across-server transaction request, it can notice when contention is increasing by just monitoring its *requestTable* entry queues. In future work, we will consider how to use this information to take corrective actions.

Transaction reordering for breaking long dependency chains. When an arriving transaction t starts waiting on a lock from another executing or pending transaction, this creates a dependency. Another arriving transaction in turn starts waiting on a lock from t, we call this a transaction dependency chain. The broker can detect long transaction dependency chains as they form. The broker can then reorder (or in the worst case abort) some transactions to break the dependency chains, and improve the completion times of the transactions, and the overall system performance.

Proactive locking. While serialization is usually thought as an on-demand and reactive process, it can in fact be proactive and anticipatory. By anticipating beforehand and by eliminating a server request for a lock (albeit in a fraction of the cases), latency can be reduced and throughput can be improved. By adopting the proactive serialization philosophy, the lock broker can anticipate a future request and migrate locks to some servers proactively so those servers can save time.

The challenge here is to figure out how proactive serialization can be achieved accurately enough to get a pay off. One approach to achieve proactive serialization is to employ presignaling. The servers can anticipate which locks they will be needing in the next transactions and ask for those locks speculatively. The broker can choose to honor some of these speculative-requests and achieve proactive serialization to improve performance. Another approach to achieve proactive serialization can be to do a static analysis of servers' programs and identify certain patterns/sequence of accesses. When the broker observes one of the identified patterns playing out, it can give the locks in advance to the respective servers. Finally, for proactive serialization it is possible to employ simple machine learning at the broker. The broker can learn the patterns of requests for locks over time and then can start distributing these locks speculatively before they are requested. In the next version of Panopticon we will consider incorporating simple machine learning based rules at the broker.

6 Related Work

6.1 Lock services

Google Chubby [5] is a centralized lock service that provides an interface similar to a distributed file system with advisory locks. Chubby depends on manual locking from the developers and is prone to the disadvantages of locking approaches. To keep the load light, Chubby provides coarse-grained locks instead of finer-grained locks. This locking scheme is more appropriate for loosely-coupled distributed systems: The Google File System [14] and BigTable [7] use Chubby as a lock service. ZooKeeper [16] is an opensource clone of Chubby.

The lock token idea has been employed in the distributed filesystems domain [21, 22]. GPFS [22] employs a centralized global lock manager in conjunction with local lock managers in each file system node, where the global lock manager can lease locks to local lock managers.

Differing from centralized lock servers, the Panopticon lock broker does not maintain all the locks, and rather it is a cache for locks that receive across-server access requests. Differing from previous work on lock brokers, Panopticon lock broker supports distributed transactions on multiple objects.

6.2 Transaction processing

Single-key transactional support. Since distributed transactions are costly and fail to satisfy the scalability requirements of web applications, several system designs have sacrificed the ability to support distributed transactions in lieu of supporting single key/object transactions in a statically partitioned setup. ElasTraS [9] provides ACID guarantees for transactions that are limited to a single object and single partition.

Limited multi-key transactional support. Since several applications requires collaboration, scalable and consistent multi-key access is critical for them. Google Megastore [3] and Megastore defines "entity groups" to partition the distributed datastore and provides ACID semantics to multi-key transactions that are confined within a predefined entity group. Megastore still has a limit of "a few writes per second per entity group" because higher write rates will cause even worse performance due to the conflicts and retries of the multiple leaders of the Paxos protocol employed for performing transactions. Many

applications in Google used Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. Examples of well-known Google applications that used Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine.

Limited wider transactional support. Relaxing the static entity groups restriction, Gstore [10] allows dynamic group formation. Key grouping requires a two-phase locking protocol which is a costly protocol, and Gstore prohibits transactions across these formed groups. To provide transactions over a distributed key-value store, Scalaris [23] employs Paxos. Similarly, CloudTPS [26] employs two-phase commit protocol to implement transactions over a distributed key-value store. CloudTPS makes the assumption that applications access only a few partitions in any of their transactions.

Sinfonia [1] provides multi-key transactional support by limiting the allowed operations in a transaction to support only a small subset of compare, and conditional read/write operations on the memory nodes. These "mini-transactions" tradeoff expressivity of transactions with improved performance. The "ordering transactions with prediction" paper [12] proposes a similar architecture to Sinfonia, but address transactions that can have conflicts. Instead of using locks, they use OCC transactions, and suggests a prediction based ordering of them in advance (making reservations at the Object Managers), in order to reduce abort rates of transactions.

General distributed transactions. Recently a number of systems attempted to provide general unrestricted transactions. H-Store [18] partitions the database in to disjoint subsets that are assigned to a single-threaded execution engine assigned to one core on a node. H-Store's scalability relies on careful data partitioning across executor nodes, such that most transactions access only one executor node. Deuteronomy [19] introduces a distributed database architecture that emphasizes decoupling of transactional component from the data component. Calvin [25] employs a deterministic ordering guarantee to reduce the prohibitive costs associated with distributed transactions.

Spanner [8] is Google's multiversion distributed database that allows distributed transactions. Spanner employs Paxos at coordinators and two-phase commit across coordinators and uses accurate timekeeping with tightly synchronized atomic clocks as a means to improve the performance of distributed transactions. The coordinators manage and coordinate locks for data items by maintaining lock lists. Since many coordinators need to be coordinated for serialization of distributed transactions, two phase commit is employed for coordinating the coordinators. While distributed coordinator transactions using two phase commit inevitably take their toll, the Spanner team believes "*it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions*". Applications that use Spanner, such as Google's F1 advertising backend [24], can specify which datacenters contain which bits of data so that frequently read data can be located near users to reduce write latency.

Panopticon as compared to previous work. Most of the systems described above rely on some form of limitation on transactions that allows for an acceptable performance. Panopticon keeps things simple with a lock broker architecture, and eschews costly protocols for distributed coordination. As a result, Panopticon does not limit transactions and allows arbitrary multi-key/object transactions. Also different from these existing work, Panopticon divorces locks from the data items in an effort to improve lock access locality. Finally, different from the systems described above, Panopticon learns the access pattern of transactions on-the-fly and adaptively migrates locks and data items in order to improve access/lock locality in the system.

7 Conclusion

Panopticon achieves scalability by divorcing locks from the data items and striving to improve lock access locality. A lock can be hosted at the lock broker or at a different server than the server that hosts the corresponding data item. The lock broker mediates the access to data shared across servers by migrating the associated locks like tokens, and in the process gets to learn about the access patterns of transactions. We showed that the broker can drastically improve the lock access locality and, hence, the performance of distributed transactions by employing simple rules.

We implemented Panopticon leveraging the Hazelcast in-memory data grid platform. Our experiments demonstrated Panopticon's improvements over Hazelcast's distributed transactions. The lock broker architecture performed significantly better as the number of data items and number of servers involved in transactions increase. This is because it is more efficient to go to the broker and test/set all the locks at once, instead of contacting other servers trying to acquire locks in increasing order in a serial manner. Also as the history locality (the probability of using the same objects in consecutive transactions) increase, Panopticon's lock migration strategies im-

proved lock-access locality and resulted in significantly better performance.

Panopticon can be employed for large-scale coordination-intensive applications including web-services, financial applications, online graph applications, social network graph processing, and distributed storage and databases.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[2] Amazon web services.". `http://aws.amazon.com/rds`.

[3] J. Baker, C. Bond, J. Corbett, JJ Furman, A. Khorlin, J. Larson, JM Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. *CIDR*, pages 223–234, 2011.

[4] S. Braun. A cloud-resolving simulation of hurricane bob (1991): Storm structure and eyewall buoyancy. *Mon. Wea. Rev.*, 130(6):15731592, 2002.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[6] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[7] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] J. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

[9] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2009.

[10] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, page 13, 2004.

[12] I. Eyal, K. Birman, I. Keidar, and R. van Renesse. Ordering transactions with prediction in distributed object stores. *LADIS*, 2013.

[13] Facebook graph search. `http://www.facebook.com/about/graphsearch/`.

[14] S. Ghemawat, H. Gobioff, and S-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37/5, pages 29–43. ACM, 2003.

[15] Hazelcast, in-memory data grid. `http://www.hazelcast.com/`.

[16] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[17] R. Jennings. *Cloud Computing with the Windows Azure Platform*. Wrox, 2010.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[19] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[21] A. Mohindra and M. Devarakonda. Distributed token management in calypso file system. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 290–297, 1994.

[22] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[23] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[24] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *VLDB*, 6(11):1068–1079, 2013.

[25] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[26] W. Zhou, G. Pierre, and C-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, pages 525–539, 2012.