Multi-site Gargamel: Optimistic Synchronization for Reliable Geo-Replicated Databases

Pierpaolo Cincilla
INRIA UPMC
pierpaolo.cincilla@lip6.fr

Sébastien Monnet
INRIA UPMC
sebastien.monnet@lip6.fr

Marc Shapiro
INRIA UPMC
http://lip6.fr/Marc.Shapiro/

ABSTRACT

Databases scale poorly in distributed configurations. This is mainly due to the cost of concurrency control and to resource contention. The alternative of centralizing writes works well only for read-intensive workloads, whereas weakening transactional properties is problematic for application developers. In a previous work we introduced Gargamel, a middleware that spreads non-conflicting update transactions to different replicas, but still provides strong transactional guarantees. We extended Gargamel to geo-replication settings. If a data-center fails, the database remains available at other locations. We minimize the synchronization cost, synchronizing optimistically replicas at distant data-centers. The evaluations of our prototype using distant Amazon data-centers show that Gargamel improves both response time and load by an order of magnitude when contention is high (highly loaded system with bounded resources), and that in the geo-replicated case, the slow-down is negligible.

1. INTRODUCTION

Databases systems have been used for decades to store and retrieve data.

Their pervasive nature makes databases fault tolerance (i.e., the ability to respond gracefully to a failure) and performance (in terms of throughput and response time) critical. Fault tolerance and performance are often addressed by replication which allows data to be stored by a group of machines. Database replication has the potential to improve both performance and availability, by allowing several transactions to proceed in parallel, at different replicas.

Database replication works well for read-only transactions, however it remains challenging in the presence of updates. Concurrency control is an expensive mechanism; it is also wasteful to execute conflicting transactions concurrently, since at least one must abort and restart.

There are several approaches to make distributed databases scalable. The concurrency control bottleneck can be alleviated by relaxing the isolation level [6, 3], relaxing the transactional ACID properties [15, 5, 1], parallelising reads [10, 11, 13], or by partial replication [14, 2].

The above-mentioned approaches families work well for some classes of application, but not for others: relaxing the isolation level introduces anomalies that can potentially break application invariants. Giving up transactional ACID properties is bug-prone and difficult to get right for application developers. Parallelising reads works well for read-dominated workloads but does not scale to write-intensive ones. Partial replication is not practical in all workloads because cross-partition transactions are not well supported, and potentially inefficient.

In previous work we introduced Gargamel [4] and we described how it scales up replicated databases efficiently to a potentially large number of replicas with full support for updates, without giving up consistency.

Gargamel retains the familiar consistency guarantees provided by commercial databases and provides strong transactional guarantees. It avoids replica synchronization after each update by moving the concurrency control system *before* the transaction execution, at the load balancer level. We described a database architecture distributed over a single site (i.e., data-center) focusing on the ability to scale and on optimal resource utilization. These goals are met, thanks to a pessimistic concurrency control. This serializes conflicting transactions ahead of time to avoid aborts and spreads non-conflicting ones.

In this paper, we present our new contribution: a multi-site architecture, using one Gargamel scheduler per site. Our approach has the potential to lower client latency by connecting to a closer replicas. This requires schedulers to synchronize in order to avoid divergence. To avoid penalizing system throughput, the schedulers synchronize optimistically, off of the critical path. Furthermore, multi-site Gargamel provides the ability to have a (or several) geographically distributed copies of both (i) the database and (ii) the scheduler's metadata. This permits to support the failure of a whole data-center.

Our contributions are the following:

- We describe how multi-site Gargamel allows several geo-replicated sites, each composed by a scheduler and a set of nodes to proceed in parallel, offering a high level of fault tolerance.

- We mitigate the high inter-site latency by synchronizing transaction execution among sites optimistically, off of the critical path.

- We show how multi-site Gargamel is suitable to lower client perceived latency by putting schedulers closer to them, to improve availability spreading schedulers in multiple geographical locations and to expand the system when the workload exceeds the capacity of a single site.

- We describe the system architecture, the distributed scheduling and the collision resolution algorithm and outline the fault tolerance.

MW4NG'14 December 8, 2014, Bordeaux, France

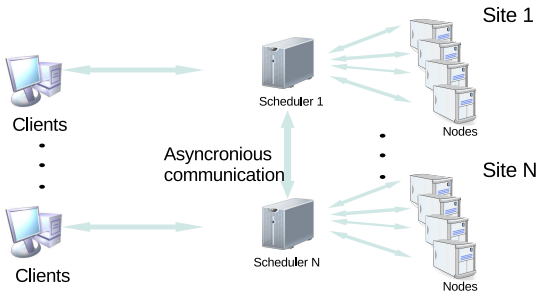


Figure 1: Multi-site system architecture

- We demonstrate the effectiveness of the multi-site speculative scheduling and the collision resolution algorithm with extensive experimentations with a prototype deployed on Amazon Elastic Compute Cloud (EC2), varying the transaction incoming rate, the database performances, and comparing against a single site deployment.

- We conclude from the experimentation that: *(i)* The optimistic scheduling approach mitigates the high inter-site latency. *(ii)* Collisions do occur, but they have only a little impact.

The paper proceeds as follows. Section 2 overviews the multi-site Gargamel architecture. The distributed scheduling and collision resolution algorithms are discussed in Section 3. We discuss fault tolerance in Section 4. The prototype and experimental results are detailed in Section 5. Finally, we conclude and consider future work in Section 6.

2. SYSTEM ARCHITECTURE

In multi-site Gargamel, as illustrated in Figure 1, there are several sites, each with its own Gargamel scheduler and a local set of nodes.

A site might be, for instance, a multicore server within a data-center, or a data-center in a cloud. The important point is that the time to deliver a message sent from one site to another, the *inter-site message latency*, is much higher than the time from one of the schedulers to its local workers. A worker is one of the process accessing the database. In our experiments each node has one worker for each CPU.

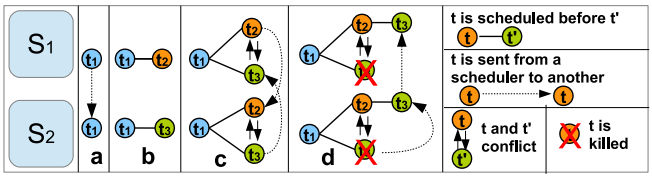
Each scheduler manages transactions execution at its nodes as described in details in [4]: within a site, conflicting transactions are serialized and non-conflicting ones are spread among replicas.

A scheduler receives transactions from local clients and sends them to its nodes for execution in parallel. Furthermore, a scheduler needs maintain and synchronize a dependencies graph, representing dependencies among transactions, with other schedulers. Synchronization between schedulers is optimistic i.e., a scheduler first sends the transaction to a worker, then synchronizes with other schedulers.

A multi-site configuration may be useful in many cases. For instance, if the client-to-scheduler message latency is high, it may be beneficial to create a site close to the client. This helps to lower the client-perceived latency. Another case is when the workload exceeds the capacity of a single site; and of course, when high availability is required and replicas should be spread in multiple geographical locations.

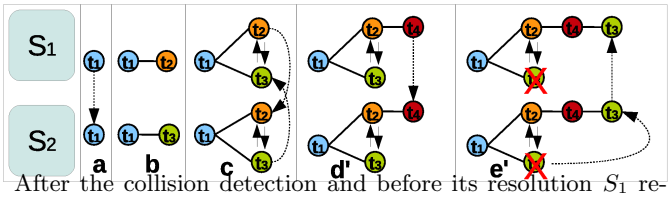
3. DISTRIBUTED SCHEDULING AND COLLISION RESOLUTION

Schedulers communicate asynchronously with one another,



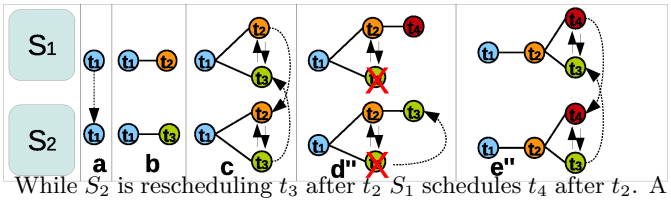
Two Sites S_1 and S_2, receive three conflicting transactions t_1, t_2, t_3. *(a)* S_1 receives t_1 and forwards it to S_2. *(b)* S_1 and S_2 receive t_2 and t_3 at the same time. *(c)* t_2 and t_3 are forwarded along with their scheduling position. S_1 and S_2 discover a collision. *(d)* S_1 and S_2 agree on the position of t_2 after t_1 and kill t_3, S_2 reschedules t_3 and forwards it to S_1.

Figure 2: Example collision.



After the collision detection and before its resolution S_1 receives t_4. *(a)* (d') S_1 receives t_4, schedules it "betting" on t_2 and forwards it to S_2. *(b)* (e') S_1 and S_2 agree on the position of t_2 after t_1 and kill t_3, S_2 reschedules t_3 and forwards it to S_1.

Figure 3: Example bet.



While S_2 is rescheduling t_3 after t_2 S_1 schedules t_4 after t_2. A new conflict arises. *(a)* (d") S_1 receives t_4 and S_2 reschedules t_3 at the same time. *(b)* (e") t_3 and t_4 are forwarded along with their scheduling position. S_1 and S_2 discover a new collision.

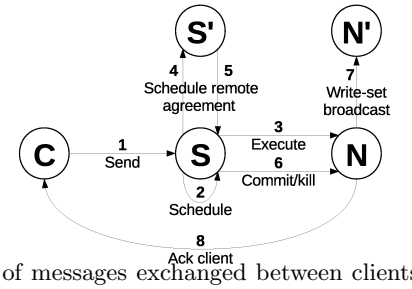
Figure 4: Example collision on bet.

and are loosely synchronized. A scheduler can receive a transaction, either from a local client (*home* transaction) or from another scheduler (*remote* transaction). First, consider a home transaction. The scheduler performs the normal scheduling algorithm described in [4]. This is done without *a priori* synchronization with the other schedulers. Once it has scheduled the remote transaction, the scheduler forwards it to all the other sites (asynchronously and reliably) along with its scheduling position. Schedulers use reliable FIFO broadcast to propagate transactions between them.

Once committed by the local database, the transaction's write-set is propagated to all other nodes (in both local and remote sites). The latter apply the write-set without re-executing the transaction [9]. Thus, although a given transaction is scheduled at all sites, it nonetheless consumes computation resource at a single site only. This avoids duplicate work, and divides the load.

3.1 Collisions

When a scheduler S receives a remote transaction from a scheduler S', it schedules it according to the single-site scheduling algorithm. When it compares its scheduling position with the position at S', it may find that the S' site has



Life-cycle of messages exchanged between clients, schedulers and nodes to execute a transaction. *(1)* The client sends a transaction to its home scheduler. *(2)* The scheduler schedules the transaction locally and *(3)* sends it for execution to a node. *(4)* The scheduler broadcasts the transaction, along with its back dependency to all other schedulers and *(5)* reaches an agreement on the position of the transaction in the dependencies graph. *(6)* The scheduler informs the node on the outcome of the agreement process (commit or kill). *(7)* In case of commit, the node broadcasts the write-set to other nodes and *(8)* acknowledges the client. In case of kill the node discards the execution.

Figure 5: Transaction message life-cycle

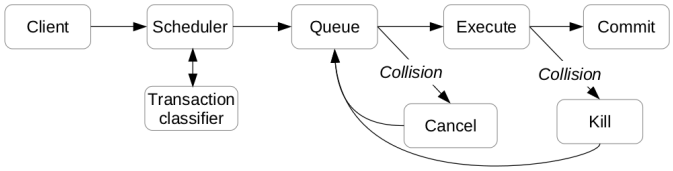


Figure 6: Transaction life-cycle

ordered the transaction differently. We call this situation a *collision*, S and S' must not both commit the transaction in different order (i.e., in a different position in the dependencies graph), otherwise they would diverge. Figures 2, 3 and 4 give some examples on how collisions happen and how they are managed.

Solving collisions requires a synchronization protocol. The obvious solution would be to synchronize *a priori*, but this could be costly, since inter-site message latency is assumed high.

Instead, Gargamel's synchronization is optimistic. It is executed in the background, off of the critical path.

As illustrated in Figure 6, a collision may occur at different times in the life-cycle of a transaction: If the collision is detected after the transaction is queued, but before it is executed, it is simply re-queued in its correct position. We call this a *cancellation*. If the collision is received after the transaction starts, the offending transaction is forcefully aborted.[1] We call this *killing* the transaction. A transaction may not commit before the collision/non-collision information is received. If this is the case, then commitment must wait. We call this situation a *stall*.

A cancelled transaction costs nothing in terms of throughput or response time. A killed transaction (either during execution or after a stall) costs lost work, this has an impact on both throughput and response time. A stall followed by a commit do not cost lost work, but this impacts throughput and response time. Our experiments show that if message latency is small compared to the transaction incoming rate, collisions most often result in cancellations. Even if the system suffers from a high collision rate, the lost work remains small.

[1]Note that this is the only case where a transaction aborts, since Gargamel has eliminated all concurrency conflicts.

3.2 Collision and Synchronization Protocol

A collision splits a chain (i.e. a sequence of dependent transactions) into incompatible branches. In the example in Figure 2*(c)*, the collision between transactions $t_2 - t_3$ splits the chain into a branch $t_1 - t_2$ and a branch $t_1 - t_3$. Those branches must not be both executed, because t_1, t_2 and t_3 mutually conflict, and should be serialized.

When this occurs, schedulers must reach agreement on which branch to *confirm* (Figures 2*(d)*). To this effect, Gargamel runs a consensus protocol between schedulers. In the presence of several branches, the protocol confirms the longest; or, if equal, the one with the first transaction with the smallest ID.

The transactions in another branch are either cancelled or killed, as explained earlier. Cancelling or killing a transaction T also cancels all local transactions that depend (directly or indirectly) on T. Thus, Gargamel does not need to run the confirmation protocol for each transaction it aborts, but only for a subset composed by one transaction for each branch.

Scheduling a transaction that conflicts with two or more colliding transactions requires *betting* on which one will win the confirmation protocol. Indeed, if a transaction is appended to conflicting branches it will be cancelled when one of the conflicting branch is cancelled. A bet consists in appending the transaction to one of the branches, hoping that the chosen branch will be confirmed. In order to maximize the probability of winning the bet, a scheduler applies the same heuristic as the collision protocol, i.e., it bets on the longest branch, or, if equal, one the one with the smallest transaction ID.

In the example in Figure 3, S_1 receives t_4, which conflicts with both t_2 and t_3. S_1 may bet either the chain $t_1 - t_2 - t_4$ or on $t_1 - t_3 - t_4$. Suppose the former (by smallest ID rule); if t_2 is confirmed, then t_4 will also be confirmed and the bet was a good one. In the worst case t_2 is cancelled and cause the cancellation of t_4 or S_1 bet on t_2 at the same time that S_2 reschedule t_3 causing a new conflict. The latter case is illustrated by Figure 4.

4. FAULT TOLERANCE

We only consider crash faults [12].We assume that nodes do not crash during the recovery process.

Multi-site Gargamel has a "natural" redundancy.This redundancy comes from the fact that each node has a full replica of the database, and each scheduler a full replica of the dependencies graph. We can use this redundancy for crash recovery.

The system can recover as long as there is at least one correct scheduler and one correct node. For correctness, once a scheduler or a node suspects a machine to be crashed, it will discard all subsequent messages from it to avoid false detection problems.

We address two different kind of failures: scheduler failure and node failure.

4.1 Scheduler Failure

The clients maintain a list of alternate schedulers. When a client suspects that a scheduler S has failed, it notifies a correct scheduler S' and sends to S' the list (*transactionList*) of transactions it has sent to S and that were not completed. Recall that schedulers use reliable First In, First Out (FIFO) broadcast to propagate transactions between them, so if a scheduler has a remote (i.e., not coming from one of its clients)

Default parameters	Value
Number of nodes	64
Number of workers per node	1
Incoming rate	50/100/150/200 t/s
Nodes and clients EC2 instances	M3.medium
Schedulers EC2 instances	M3.xlarge
Load	100,000 transactions
Warehouses (TPC-C)	10

Table 1: Experiment parameters

transaction in its local view, then eventually all correct schedulers will receive that same transaction.

A transaction t in *transactionList* can be in one of three possible states: **i) t is in the local view of S':** this means that the scheduler has crashed after step 4 of Figure 5 (transaction propagation between schedulers). This implies that the transaction has been already delivered for execution to some node (see Figure 5). In this case, S' will take no actions. The client will eventually receive the reply for that transaction. **ii) t is not in the local view of S' and is not scheduled for execution at any of the nodes of S:** this means that the scheduler has crashed before step 3 of Figure 5. The transaction is "lost" because, except for the client, none of the surviving nodes on the system knows about it. In this case, S' reschedules the transaction as a home transaction transaction. **iii) t is not in the local view of S' and it is scheduled for execution at one of the nodes of S:** this means that S crashed after step 3 and before step 4 of Figure 5. The transaction has been scheduled and sent to a node for execution, but the *shceduleRemote* message was not sent to the other schedulers. In this case, S' retrieves the transaction and its back dependency (as calculated by S) and reschedules it locally in the same position (i.e., keeping the back dependency calculated by S), and sends *shceduleRemote* to other schedulers (including itself). This recovers the execution from the point at which was interrupted.

This procedure can be repeated until there is at least one correct scheduler in the system.

4.2 Node Failure

When a Scheduler suspects a node failure, it fetches from the dependencies graph the list of transactions sent to that node for execution and checks on the survivor nodes which write-sets have not been received. It then reschedules for execution transactions that have not been received by survivor nodes. Notice that nodes send reply to clients after broadcasting the write-set to other nodes, otherwise in case of failure clients can receive more than one reply to the same transaction.

5. EVALUATION

The main Gargamel prototype components (nodes, schedulers and emulated clients) are written in Java (∼12k lines of code). They communicate through JGroups [7], a reliable multicast system. Our concurrency control and update propagation mechanism is based on group communication, and correctness depends on the properties of the communication channels.

We use an unmodified version of PostgreSQL [16], an open source relational Database Management System (DBMS) for nodes' database. The communication with the database is done through Java Database Connectivity (JDBC). Gargamel is evaluated using the TPC-C benchmark.

Default parameters	Value
Number of nodes	32
Number of workers per node	1
Incoming rate	20 to 100 t/s
Load (single-site)	10,000 transactions
Warehouses (TPC-C)	10

Table 2: Experiment parameters for in-disk database

5.1 Multi-Site Gargamel Evaluation

In this set of experiments we compare multi-site Gargamel against single site Gargamel on a in-memory database. We measure the client-perceived latency (i.e. the time elapsed between the time the client sends the transaction and the time it receives the corresponding commit message). Figure 7 compares the case of all transactions being serialised at a single node to Round-Robin and single-site Gargamel (both using and not using certification) in terms of transaction latency for different incoming rates. Figure 8 shows the latency perceived by clients in Ireland and Oregon in the single site and multi-site configuration varying the incoming rate. The latency perceived by the client in Ireland on the single site setting is an order of magnitude lower than the latency experienced by all other clients. This is because in single site Gargamel transactions coming from the Ireland EC2 region are executed in the local data-center and do not need to synchronize remotely. In the other hand, transactions coming from clients located in Oregon in the single site Gargamel case show a much higher latency because they suffer for the high latency between the west of Europe and the west of the United States. Interestingly, the latency observed for the client located in Oregon in the single site configuration, are similar to the ones observed in the multi-site configuration, a little bit higher than multi site Gargamel clients in Oregon and a little bit lower than multi-site Gargamel clients in Ireland. This is because from whatever client multi-site Gargamel receives a transaction, it synchronizes with the other scheduler, paying the price of a transatlantic round-trip message. In multi-site Gargamel, clients in Oregon have a lower latency than clients in Ireland because in order to simplify the implementation, our prototype elects a scheduler do be the leader of the agreement protocol. The leader resolves conflicts in case of collisions. In our experiments the leader is the Oregon scheduler, giving a small advantage to clients connected to that scheduler.

In average, multi-site Gargamel does not show any improvement or overhead over single-site Gargamel for transactions that come from distant clients. This is because at low incoming rate, as in this experiment configuration, transactions are executed as soon as they arrive, so multi site Gargamel does not form long execution chains and the agreement latency cannot overlap the time the transaction spend waiting for the execution of previous transactions in its chain. The optimistic scheduling benefit comes from the fact that the agreement on the transaction execution order between sites proceeds in parallel with the transaction execution.

5.2 Impact of Database Performance

We evaluate the impact of transaction execution latency running the same experiments of Section 5 in a less efficient database. The next set of experiments is based on a in-disk PostgreSQL database. Points in the following plots are produced with a single run of 10k transactions. We reduced the number of transactions of each experiments from 100k to 10k to keep constant the total "running time" i.e. the time it

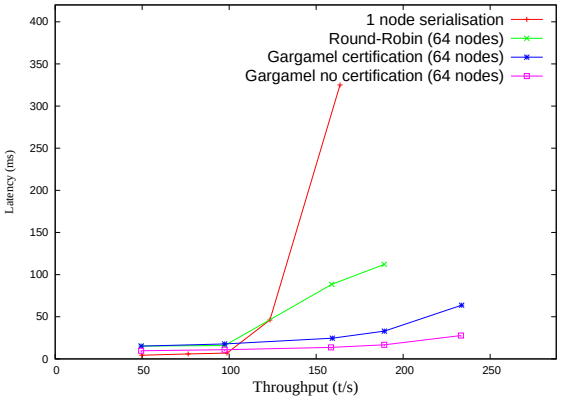


Figure 7: Single-site comparison

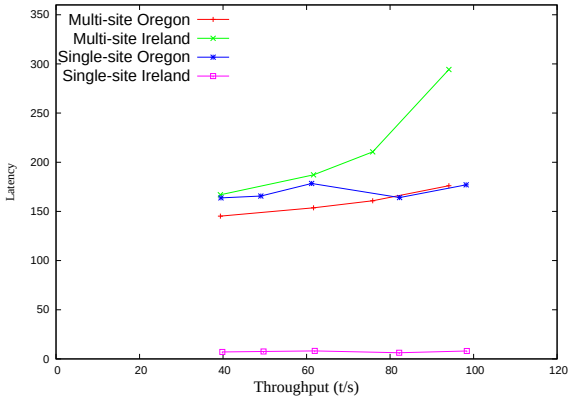


Figure 8: single site VS multi-site Gargamel

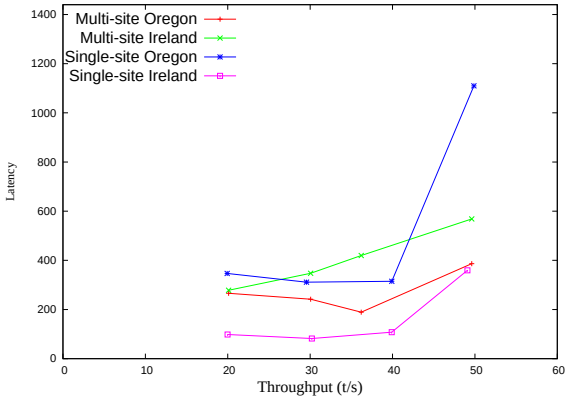


Figure 9: Single site Gargamel VS multi-site Gargamel slow database

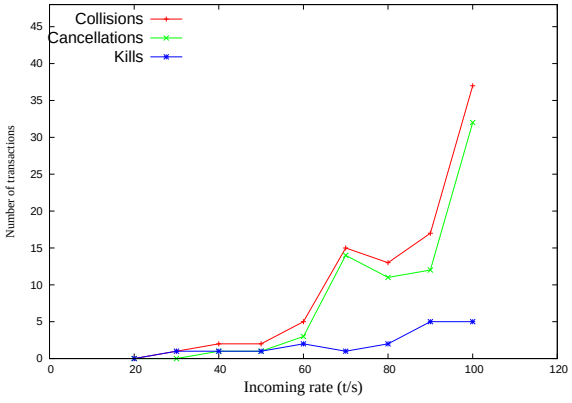


Figure 10: Impact of collisions

takes for the system to execute all the transactions. Unless differently specified, we use the parameters from Table 2.

The in-disk database is more than ten times slower than the well-tuned in-memory database.

In the experiments presented by Figure 9 we compare how multi-site and single site Gargamel perform when using a slow in-disk database. As for the experiments illustrated by Figure 8, we measure the latency perceived by different clients located in different regions as the incoming rate, and consequently the throughput, increases. The main difference with executions using an in-memory database is that even the latency of clients in Ireland in single site Gargamel experience a high latency, and the difference with other clients is smaller than for the in-memory database. This is due to the long transaction execution time (between 50 and 90 millisecond). In this experiment the incoming rate is low, so transactions are executed as soon as they arrive in the system. They do not form chains and the higher execution time is not enough to show the benefits of the optimistic scheduling approach.

5.3 Impact of Collisions

The optimistic scheduling is interesting when Gargamel forms transactions chains. To show this, we have to saturate the system. At high incoming rates Gargamel organizes transactions in chains according to their conflict relations. Multi-site Gargamel schedules transactions at each site according to the local view, then synchronizes schedulers optimistically. In this way multi-site Gargamel masks the agreement delay (because the agreement is performed in parallel with the execution of the transactions in the chain), but collisions on the transaction execution order at different sites are possible. In order to evaluate the impact of collisions and the effectiveness of the optimistic approach we saturate Gargamel schedulers pushing the incoming rate up to 100 transactions per second in the in-disk database.

Figure 10 shows the number of transactions that collide, and among colliding transactions how many are cancelled and how many are killed. We recall that a transaction is cancelled when it is rescheduled before its execution starts, and is killed when its execution is started and have to be aborted in the database. The key difference is that cancellations do not cause waste of work because the transaction is not executed at any replica but just rescheduled at a different position while kills involve executing and aborting a transaction at a node, wasting resources. As showed in Figure 10, when the incoming rate is low, at 30 transactions per second, there are few collisions. Those collisions result in a kill of the corresponding transaction, because transactions are executed as soon as they arrive

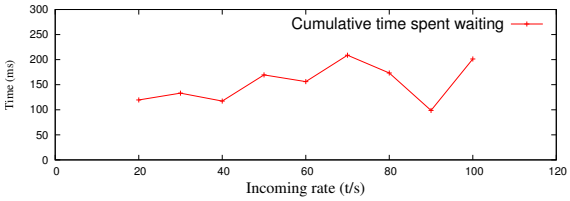


Figure 11: Cumulative time spent waiting for agreement

and there is no time to reschedule them. Until 60 transactions per second, collisions are very rare and roughly half of them cause the transaction to abort (kill) and the other half cause a simple rescheduling (cancellation). When the incoming rate increases, the number of collision increases as well and the effectiveness of the optimistic protocol became visible: most of the collisions cause a simple cancellation and transactions are rescheduled before being executed. At an incoming rate of 80 transactions per second and more, the number of collision increases rapidly (from 10 to more than 30), but the number of collisions that cause lost work, remains stable and low. Even if multi site Gargamel experience some amount of collisions, the lost work remains low, and the degradation of the resource consumption optimality is modest.

The optimistic scheduling imposes to synchronize schedulers to agree on the transaction position in the dependencies graph. Transactions cannot be committed in the database until the agreement on their position in the chain is reached. If a transaction execution finishes before the schedulers have agreed on its position, the transaction wait for the agreement process to finish before being commit or killed. In the next experiment we have measured, for increasing incoming rates, the cumulative time spent by transactions waiting for the outcome of the agreement. Figure 11 shows the cumulative time spent by transaction waiting for the outcome of the agreement as the incoming rate increases. The cumulative waiting time is between 100 and 200 milliseconds for all the runs. The workload of a run is composed by 10k transactions, consequently in average the transaction commitment is delayed by a time between 0.001 and 0.002 milliseconds. Considering that in this setting (slow in-disk database) the average transaction execution time is between 60 and 90 milliseconds, the delay on the commit time is negligible. The delay is so small because most of the transactions do not wait at all: at the time the transaction is executed the agreement protocol is done and the transaction can commit right away; and as showed before, cases in which the transaction cannot commit and should be killed are rare.

6. CONCLUSION

A transaction reordering mechanism is proposed by Pedone et al. [8]. Their system reorder transaction commitment order to reduce aborts. Differently from Gargamel they don't avoid aborts, but they alleviate their impact.

Multi-site Gargamel allows several geo-replicated sites, each composed by a scheduler and a set of nodes, to proceed in parallel. Each site receives transactions from local clients and executes them at local nodes. Synchronization among sites on the execution order is done optimistically, off of the critical path. Multi-site Gargamel is suitable to lower client perceived latency by putting schedulers closer to them, to improve availability spreading schedulers in multiple geographical locations and to expand the system when the workload exceeds the capacity of a single site.

We have described the system architecture, the distributed scheduling and collision resolution algorithm and outlined the fault tolerance.

We evaluated its performances and benefits using a prototype running on top multiple Amazon data-centers.

7. REFERENCES

[1] Memcachedb http://memcachedb.org/.

[2] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí. Sipre: A partial database replication protocol with si replicas. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 2181–2185, New York, NY, USA, 2008. ACM.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[4] P. Cincilla, S. Monnet, and M. Shapiro. Gargamel: boosting DBMS performance by parallelising write transactions. In *Parallel and Dist. Sys. (ICPADS)*, pages 572–579, Singapore, Dec. 2012.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[6] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. *Reliable Distributed Systems, IEEE Symposium on*, 0:73–84, 2005.

[7] A. Montresor, R. Davoli, and O. Babaoğlu. Middleware for dependable network services in partitionable distributed systems. *SIGOPS Oper. Syst. Rev.*, 35(1):73–96, Jan. 2001.

[8] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Symposium on Reliable Distributed Systems*, pages 175–182, 1997.

[9] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[10] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *In Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, pages 155–174, 2004.

[11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 17–30, New York, NY, USA, 2011. ACM.

[12] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[13] J. Sobel. Scaling out. Engineering @ Facebook Notes `https://www.facebook.com/note.php?note_id=23844338919`, Aug. 2008.

[14] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, NCA '01, pages 298–309, Washington, DC, USA, 2001. IEEE Computer Society.

[15] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[16] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *ACM Sigmod Record*, volume 15, pages 340–355. ACM, 1986.