Fireplug: Flexible and Robust N-version Geo-Replication of Graph Databases

Ray Neiheiser*, Daniel Presser*, Luciana Rech*, Manuel Bravo‡, Luís Rodrigues‡, Miguel Correia‡
*Departamento de Informática e Estatística, Universidade Federal De Santa Catarina
‡INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—The paper describes and evaluates Fireplug, a flexible architecture to build robust geo-replicated graph databases. Fireplug can be configured to tolerate from crash to Byzantine faults, both within and across different datacenters. Furthermore, Fireplug is robust to bugs in existing graph database implementations, as it allows to combine multiple graph databases instances in a cohesive manner. Thus, Fireplug can support many different deployments, according to the performance/robustness tradeoffs imposed by the target application. Our evaluation shows that Fireplug can implement Byzantine fault tolerance in geo-replicated scenarios and still outperform the built-in replication mechanism of Neo4j, which only supports crash faults.**

***Index Terms*—Graph databases, Geo-replication, N-version programming, Byzantine faults.**

I. Introduction

Graphs offer an elegant data representation for problems that would not be easily expressed otherwise. In fact, many areas benefit from the use of graphs, including social sciences, natural sciences, engineering, and economics [1]. Unsurprisingly, the increasing number and relevance of applications using graphs as a data model spurred the development of several graph databases, which are optimized to support graph storage and query. Examples include Neo4j [2], OrientDB [3] and TAO [4]. These specialized databases have been shown to outperform classical relational databases to support graph processing: for instance, Neo4j was shown to be 3x faster than MySQL on several tasks, such as graph traversal [5].

Interestingly, some of the applications that leverage graph databases often require simultaneously strong consistency, security, and scalability [6]. For instance, Neo4j and OrientDB customers include security firms, investigation units, media companies (*Sky, Comcast, Warner*), and trade companies (*Ebay* or *Global 500 Logistics*), which use graph databases to offer real time product routing and delivery to their clients [6], [7]. Therefore, deriving solutions that can replicate graph-databases in an efficient and robust manner, is a challenge not only of technical interest but also of practical relevance.

In this paper, we describe the design and implementation of Fireplug, a flexible architecture to build robust geo-replicated transactional graph databases. Its key features are:

- Fireplug supports both intra- and inter-datacenter replication. Thus, it tolerates both errors that are common when using commodity hardware, and natural or human-caused disasters. Replication also has the potential to reduce latency by letting clients interact with the closest datacenter(s), and to enhance scalability under read-dominated workloads.
- Fireplug implements a novel Byzantine fault-tolerant deferred update replication protocol, specialized for graph databases. Tolerating Byzantine faults is fundamental as the causes for both natural and human-driven Byzantine faults are becoming more prevalent, including for instance, the increasing gate density in silicon and the threat of cyber-criminality. Furthermore, Byzantine faults can cause serious revenue loss, e.g., the Stuxnet worm [8], identified in 2010, that caused substantial delays in nuclear research at Iran. More recently, the attack on the Linux Mint distribution in 2016 [9], in which a corrupted version was uploaded to their site, compromising users that installed it.
- Fireplug supports software diversity and implements N-version programming [10] to shield the system from bugs and attacks to vulnerabilities of specific graph databases' implementations: code made by independent teams, even using different programming languages, that goes through different release procedures, is less likely to suffer from the same bugs. Graph databases are large open source projects, where it is hard to eliminate all vulnerabilities that can be exploited by attackers (via buffer overflows, query language attacks, etc.). In 2016 only, a long list of open-source software with critical security flaws has been identified [11]. Examples include vulnerabilities in the glibc Linux library, in MySQL, and in OpenJDK. Fireplug currently supports 4 graph databases: Neo4j, OrientDB, Titan, and Sparksee.
- Fireplug's architecture is flexible. The goal is to allow a variety of configurations of the system such that application fault tolerance and performance requirements are met. For instance, N-version programming can be used to make each datacenter Byzantine fault-tolerant and then just assume crash faults at the level of an entire datacenter, making inter-datacenter replication more efficient. However, if one is concerned with attacks that can compromise an entire datacenter, one can also make the inter-datacenter operation Byzantine fault-tolerant. Fireplug can also be configured to run in a single datacenter and to tolerate only crash faults; this is interesting because Fireplug outperforms the native replication scheme protocols of the graph databases, as these single master and do not benefit of the multiple replicas.

The system has been extensively evaluated. We have observed that Fireplug outperforms the native replication mechanism of Neo4j when tolerating not only crash faults (as Neo4j) but also Byzantine faults. Additionally, our architecture shows significant performance gains when compared to more traditional architectures.

In summary, this paper's contributions are: (i) the first application of diversity and N-version programming for graph databases; (ii) *Graph-DUR*, a specialization of deferred update replication for graph databases; (iii) a flexible architecture, deployable to tolerate different fault types and settings; (iv) an open-source implementation of the resulting prototype[1].

II. Related Work

Database Replication. Many replication techniques leverage the existence of an atomic broadcast primitive like Paxos [12]—and its variants—which is one of the most widely used protocols, for tolerating not only crash faults [13] but also Byzantine faults [14], [15]. Not surprisingly, a number of previous works have also considered the deployment of atomic broadcast protocols across multiple datacenters [16]–[18].

Among these techniques, *Deferred Update Replication* (DUR) has been shown to be particularly effective [19]. In DUR, a transaction is executed only against one local replica; such that writes are locally cached, and reads are served locally. On commit time, the transaction's write and read set are sent to other replicas for validation, by means of an atomic broadcast. If no conflicts are detected, writes are atomically executed, otherwise these are discarded, and the transaction aborted. Of particular interest to us is the work of [20], that introduces an implementation of DUR able to tolerate Byzantine faults. Fireplug uses a variant of this last protocol that, unlike previous solutions, is specially designed for graph databases. The differences between our specialized DUR variant and [20] are twofold: our implementation considers graph semantics; and we avoid digital signatures for validating reads, expensive otherwise in graph workloads (§IV further elaborates on this).

N-version Programming Research on N-version programming, or software diversity, started in the 1970s and raised considerable interest [21]. As discussed in [21], faults are often caused by design flaws. N-version programming aims at tolerating these design faults, using a range of independently-designed software elements, which also has been shown to decrease the likelihood of malicious intruders [22]. Unfortunately, due to the large deployment effort required to combine enough distinct implementations, N-version programming is—with a few exceptions [23]—seldom used practice.

MITRA [24] is an example of the use of software diversity to shield the system from Byzantine faults. Although it is designed for relational databases, a similar approach may be an asset when replicating graph databases.

Graph Databases. Graph Databases [25] are database management systems that have been optimized to store, query and update graph structures. In graph databases relationships are first-class citizens on the graph data model. This is not true in other database management systems, where relations between entities have to be inferred using other abstractions such as foreign keys, making the task of querying the graph an inefficient join-intensive procedure. To avoid these limitations, graph databases store pointers in the corresponding vertices and edges. Fireplug gathers a set of features that make it unique when compared to other graph databases. First, it efficiently tolerates not only crash faults but also Byzantine faults. Many graph databases implement a variant of semi-active replication for fault tolerance, but they tolerates only crash faults; they do not consider Byzantine faults. Finally, above all, it shields the system from software vulnerabilities by relying on software diversity.

III. Fireplug Overview

Fireplug is a transactional graph database management system. It is designed to run in one or more datacenters. Each datacenter runs one or several nodes (or servers) and each node runs a (potentially different) instance of the graph database. We target a full replication scenario, where all nodes maintain a full copy of the graph. Typically, the latency among nodes residing within the same datacenter is much smaller than the latency among nodes in different datacenters. This may have an impact on the performance of Fireplug but not on its correctness, as we assume an asynchronous system model.

Fireplug offers flexibility at multiple levels. (i) Each node can be configured to run a different graph database. Currently, Fireplug supports Neo4j, OrientDB, Titan, and Sparksee. These graph databases may be oblivious to the Fireplug replication mechanisms but, still, the resulting assembly behaves as a single, serializable, database. (ii) It permits multiple fault models to coexist in a single deployment. Note that in a system where all nodes must be configured to tolerate the same type of faults, one may be forced to choose the most restrictive model, unnecessarily degrading overall system's performance.

A. System Components

The main software components of Fireplug are (Figure 1):

- The application front-end machines, which we designate by clients. We assume that clients receive end-user requests and run the application.
- A middleware integration layer called the *common GRAph DAtabase replication Middleware* (GRADAM). Its goal is to support the inter-operation of multiple graph databases in a cohesive environment by unifying their interface, as each database is likely to offer a slightly different interface.
- A proxy that provides a uniform interface to the graph database so that GRADAM can abstract away these details.
- A replication protocol, *Graph-DUR*, that implements deferred update replication tolerant to Byzantine faults and specifically designed for graph databases.
- As part of Graph-DUR, Fireplug implements *Hierarchical BFT-SMaRt*, an atomic broadcast abstraction implemented as a hierarchical composition of multiple instances of the BFT-SMaRt service [15].

B. System Operation

Clients connect to an instance of GRADAM typically at the closest server and coordinate the execution of transac-

[1]https://github.com/Raycoms/fireplug

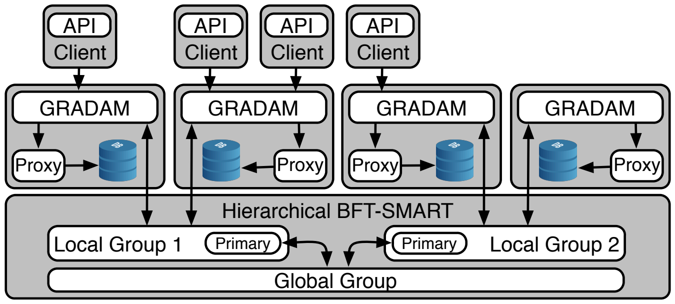

Fig. 1: Fireplug architecture.

tions that are composed of several read and write operations. Transactions are marked by the *startTransaction* and *endTransaction* delimiters. The responsibility of a GRADAM instance is twofold: (i) it bridges local clients with the local graph database instance, translating between the common interface (the one exposed to clients) and each particular graph database interface; (ii) it interacts with the remaining GRADAM instances, running on remote replicas, to ensure that all transactions that commit are serializable.

The communication among multiple GRADAM instances running at different nodes is coordinated by our own implementation of a DUR protocol. The explicit exchange of message among instances is done using an atomic broadcast abstraction. This can be configured to tolerate both crash and Byzantine faults, and to offer different qualities of service. As already mentioned, the atomic broadcast service leverages BFT-SMaRt [15], an open source library that implements Byzantine-tolerant state machine replication. Our implementation uses a hierarchical combination of multiple BFT-SMaRt groups and makes use of two broadcast services offered by BFT-SMaRt, namely a (non-ordered) reliable broadcast and a (totally ordered) atomic broadcast (later detailed in §IV-D).

IV. Graph-DUR: Replication in Fireplug

Replication in Fireplug is managed using a variant of the Byzantine-tolerant DUR proposed in [20]. We first discuss the major differences between our implementation, Graph-DUR, and [20]. Then, we describe how update and read-only transactions are processed. Finally, we detail the implementation of the atomic broadcast primitive integrated into Fireplug, a fundamental abstraction for Graph-DUR.

A. Adaptation of DUR for Graph Databases

Semantic-Awareness. Graph-DUR considers the semantics of the graph structure in an effort to reduce conflicts, a technique commonly used in both transactional memory [26], [27] and geo-replicated distributed systems [28], [29].

First, clients can potentially merge updates before pushing them to Fireplug to reduce the size of transactions; e.g., updating a vertex can be merged with its preceding vertex creation request, and deleting a vertex invalidates preceding updates or creations on the same vertex.

Second, the conflict detection algorithm has been adapted to also take the semantics of graph operations into account. Differently to relational databases—where concurrent operations over the same key trigger a conflict—there is, in graph databases, an opportunity to reduce conflicts—and enhance performance—by detecting commutativity among some operations. For instance, imagine two concurrent transactions attempting to remove the same vertex from the graph. In a transactional NoSQL database the conflict detection algorithm would abort one of the two as removing the same vertex would be considered as a conflict (both 'update' the same key). Our conflict detection algorithm considers both operations as commutative, allowing both to commit. In order to implement such conflict detection algorithm, Graph-DUR requires clients to explicitly track not only reads and writes, but also create and delete operations. Thus, commutative operations can be efficiently spotted at the certification phase and unnecessary aborts are precluded.

Signature-free Read Validation. In a Byzantine-tolerant setting a single instance of the database cannot be trusted. If the instance is faulty, it may return bogus or stale data. In [20] this is addressed by having the servers sign every data item. In the case of relational database this is valid because it is possible to sign tables or sections. Nevertheless, in our case, this is not a viable option. Graph databases store their data in fairly independent nodes and relationships. Therefore, implementing this mechanism would imply signing every node and relationship in the graph, adding a severe overhead.

Therefore, we validate read-only transactions by comparing the contents of different instances. In order to decrease the complexity and delay caused by global validation, we propose several optimizations in IV-C.

B. Update Transactions

Update transactions are executed following the DUR algorithm: optimistically against a single replica (typically the closest one) and validated in parallel by all replicas in total order, to ensure serializability. As a result of our algorithm, all correct replicas certify the same sequence of transactions, in the same order. Thus, all correct replicas will reach a consistent decision regarding the commit or abort of the transaction.

Algorithm 1 depicts the node-side (or server-side) protocol. A client first selects a node. Then, it requests the identifier of the current snapshot of the database. To ensure serializability, every time a new value is read, its timestamp is checked to see if the transaction is reading from the same snapshot. If not, this means that another update transaction has been committed, and the transaction will be aborted. Writes, deletions, and creations are cached at the client. When the transaction is ready to commit, the transaction is sent for certification. A totally ordered broadcast is used, to ensure that all servers certify the transactions in the same order. If the transaction passes the certification, it is committed by all nodes; otherwise it is aborted and all updates discarded.

The main differences to classical DUR are, as noted above, the need to keep separate read, write, create, and delete sets at the client, and the need to use a conflict detection algorithm that takes the semantics of the graph operations into account.

Algorithm 1 Server code

```
1: $global\_ts = 0$                                  ▷ Initiate global snapshot Id
2: function GET_SETVER_TS
3:     return global_ts
4: function SERVER_READ(oid)
5:     (oid, v, ts) ← RETRIEVE(oid)
6: function VALIDATE_READSET(read_set)
7:     for ∀ (oid, v, ts) ∈ read_set do
8:         (oid, v', ts') ← RETRIEVE(oid)
9:         if $v \neq v'$ or $ts \neq t'$ then
10:            return abort
11: function VALIDATE_UPDATESET(ts, w_set, d_set, c_set)
12:     for ∀ (oid, v) ∈ c_set ∪ d_set ∪ w_set do
13:         (oid, v', ts') ← RETRIEVE(oid)
14:         if $ts < t'$ then
15:             return abort
16: function COMMIT(ts, $t.RS$, $t.WS$, $t.DS$, $t.CS$) ▷ called in total order
17:     result ← VALIDATE_READSET($t.RS$)
18:     if result = abort then
19:         return abort
20:     result ← VALIDATE_UPDATESET(ts,$t.WS$, $t.DS$, $t.CS$)
21:     if result = abort then
22:         return abort
23:     global_ts ← global_ts+1
24:     APPLY_UPDATES(global_ts, $t.WS$, $t.DS$, $t.CS$)
```

C. *Read-only Transactions*

To improve performance, read-only transactions are executed differently from classical DUR. Fireplug supports read-only transactions with 2 different resilience levels: *locally-safe* reads, that cross check the results using two replicas of the same datacenter and; *globally-safe* reads, that cross check the results using nodes of different datacenters (globally-safe reads should be used if there is the threat of an entire datacenter becoming compromised). Read transactions are always first executed optimistically in a single replica and then sent—through the non-ordered channel—to $f + 1$ replicas for cross validation. If the cross validation fails, read transactions are re-executed in pessimistic mode, similarly update transactions. Although, this may cause additional overhead in the worst case, given that workloads in graph databases are typically read-dominated [4], in most of the cases, the validation will succeed without requiring the execution of a totally ordered broadcast. This brings significant performance gains in the most frequent case.

Note that locally-safe reads do not guarantee reading the newest data committed globally. Therefore, using this primitive, Fireplug guarantees snapshot isolation [30] instead of serializability. Nevertheless, this is still a useful and powerful criterion, the default in major database engines such as Oracle or Microsoft SQL Server, which brings significant performance improvements (as shown in §V-B).

D. *Hierarchical Atomic Broadcast*

Graph-DUR requires the execution of an atomic (totally) ordered broadcast primitive across multiple datacenters. We have implemented this primitive as a hierarchical composition of multiple instances of the BFT-SMaRt service.

The general architecture of the implementation is depicted in Figure 2. In each datacenter, we setup an atomic broadcast group that coordinates all replicas that reside in that datacenter.

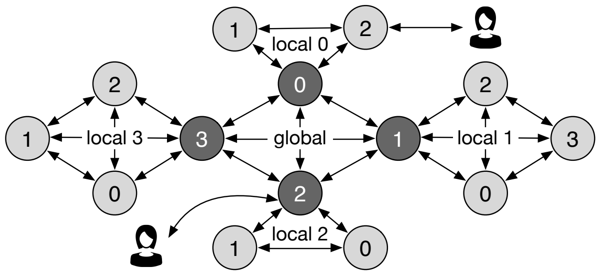

Fig. 2: Hierarchical architecture.

Then, one replica from each datacenter is elected to participate in an inter-datacenter atomic broadcast group—we refer to each of these nodes as *primary*. Thus, instead of coordinating all replicas in a single large atomic broadcast group—an approach that we refer as *flat*, coordination is achieved by executing a sequence of actions on the smaller intra-datacenter and inter-datacenter atomic broadcast groups. We denote the inter-datacenter group simply as the *global* group and the internal group in datacenter i as $local_i$. For liveness, we assume that the system is augmented with an unreliable failure detector that can trigger the change of a faulty or stalled leader.

The global group can be configured to tolerate both crash or Byzantine faults, requiring $2f + 1$ or $3f + 1$ participants respectively. Note that if the global group is configured to tolerate Byzantine faults, and less than $3f + 1$ datacenters are available, datacenters may have to participate in the global group with more than one node. Local groups can also be either crash- or Byzantine-tolerant, offering extra flexibility when configuring the system to improve performance. If the global group is configured to only tolerate crashes, local groups will not be able to tolerate Byzantine faults.

1) General Structure of the Protocol: The protocol operates as follows. First, the client forwards an update transaction t to the global group. In turn, the global group decides on an order for t by relying on the underlaying atomic broadcast abstraction. If the nodes decide to commit the transaction after checking for conflicts, each primary broadcasts t, together with the assigned timestamp, to its local group. A receiving node, only applies t if all transactions with a timestamp smaller than t's timestamp have already been applied.

If the global group is configured to tolerate Byzantine faults, there is an extra step in the protocol. Once the global group decides to commit a transaction t, each primary signs the decision and broadcast (through the non-ordered channel) to all the other primaries. Once a primary p has received $f + 1$ signatures for t, this is broadcasted to the nodes in n's local group, together with the t's assigned timestamp. A receiving node in a local group, only applies t, if its signed by $f + 1$ nodes of the global group and all transactions with a timestamp smaller than t's timestamp have already been applied.

2) Crash Faults: When a node crashes or is suspected, if that node is not a primary, no special action is performed since it is transparently handled by the atomic broadcast principle. But, if the node is a primary (member of the global group), corrective measures need to be performed. First a new primary is elected from the local group of the faulty node. If the old

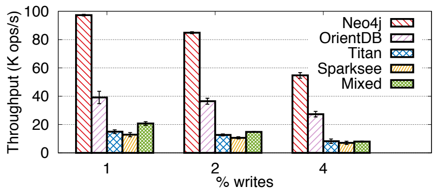

Fig. 3: N-version programming.

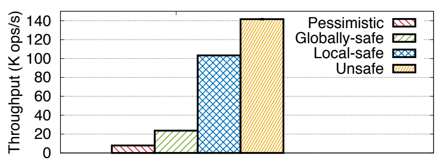

Fig. 4: Read modes.

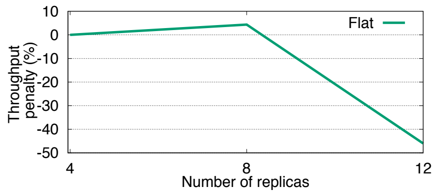

Fig. 5: Hierarchical vs. flat.

primary is still active (and was just slower), it will remove itself from the global group. If the old primary crashed, the new primary of the local group pro-actively joins the global group, obtains the delivery log of the global group, and reliably broadcasts to the local group all messages that have not been propagated to the local group by the faulty leader.

3) Tolerating Byzantine Nodes: There are two cases to consider. First, if a primary behaves correctly locally but wrongly globally. This will be detected by other primaries; e.g., wrong signing or wrong behavior when executing the underlaying atomic broadcast primitive, which will notify the local group to which the faulty primary belongs. In turn, the local group will elect a new primary that eventually will join the global group. Second, when a the faulty primary behaves correctly at the global group but fails to propagate the updates to its local group. In this case, replicas of a given datacenter could become stalled for a long period. In fact, this behavior could only be detected by a client doing globally-safe reads or writes. To speed up the detection of this faulty behavior, the client proxy waits for $f+1$ replies from each datacenter; if those replies are missing it issues an accusation against the primary of that datacenter that is sent to all members of the corresponding local group. As in the first scenario, this will trigger a leader change in the local group.

4) Tolerating Byzantine Datacenters: Fireplug can also be configured to tolerate faults at the level of an entire datacenter, as long as the deployment includes at least $3f+1$ datacenters or the datacenter offering multiple replicas to the global cluster is not faulty. In fact, the protocol described in the previous section tolerates the failure of f entire datacenters or of f machines in one of the datacenters.

V. Evaluation

In this section, we present the results of an experimental study of Fireplug. We answer the following questions:

- How well does Fireplug perform in terms of throughput?
- How do the different read-only protocols perform?
- What are the performance advantages of the hierarchical atomic broadcast when compared to a flat solution?

Our workloads are read-dominated, mimicking worloads based on real usage of graph databases (such as Facebook's TAO [4]). The dataset has been synthetically generated using GMark [31] and has 100,000 nodes and approximately 230,000 relations. Each experiment has been executed three times in order to avoid statistical errors, and the results presented are the mean of the throughput per second observed in all servers during a 5 minutes execution interval. We discarded the first and the last minute of each experiment, to avoid effects of the warm-up and cool-down periods. Experiments were carried out using the Grid'5000 testbed [32].

A. Throughput Experiments

We compare the performance of Fireplug using different graph databases and "mixed" (all 4 databases together), when varying the read/write ratio. We reach up to 4% of writes, which is already 20 times the ratio observed by the Facebook TAO's team [4]. The goal is to understand the impact of writes in Fireplug and how it behaves when multiple graph databases coexist. We used a total of 4 replicas in a single datacenter, tolerating one Byzatine fault.

As Figure 3 shows, Neo4j exhibits the best performance compared to all other configurations. Nevertheless, the mixed execution is, as expected, in between other configurations: better than the two worst. This is not a surprise since we expect the slowest graph database to slow down the system significantly. Nevertheless, it exhibits slightly better throughput than the slowest because read-only transactions do not usually need to contact all replicas. Interestingly, when running the mixed one, we noticed that balancing the load among the replicas—as each performs differently—is of paramount importance to achieve reasonable performance. The problem is that, if care is not taken, one can overload a slow replica, leading all other replicas to a severe slowdown. In this set of experiments, we have manually tuned the load among replicas based on the maximum performance each can handle.

B. Read-only Transactions

In this second experiment, we compare the performance of our two optimistic read-only modes (locally-safe and globally-safe) with a pessimistic mode, which executes read-only transactions as update transactions by relying on the more expensive atomic broadcast abstraction. As a baseline (bar named *unsafe*), we also plot the throughput that a read protocol that simply reads from the local replica, with no validation afterwards, would achieve. Although such a read protocol may return bogus values, as there is no cross-replica validation, it serves as a throughput upper-bound.

We deploy Fireplug with 16 replicas, evenly distributed among 4 datacenters. In this setting, Fireplug tolerates one Byzantine node per datacenter, and the entire failure of one datacenter. Our results (Figure 4) shows that the pessimistic approach exhibits the worst performance. The globally-safe mode already shows a significant improvement by handling 3× more operations per second, still providing the same guarantees. Interestingly, the locally-safe mode gets fairly close to the unsafe mode. Therefore, if applications can tolerate reading slightly stale data, embracing the locally-safe mode can bring significant performance improvements.

C. Hierarchical vs. Flat

In this last experiment we compare our hierarchical architecture with a flat architecture in which there is a unique atomic broadcast group. For this, we vary the number of replicas from 4 to 12 and see the impact in throughput. Figure 5 shows the throughput penalty incurred by the flat architecture when compared to an equivalent experiment using the hierarchical architecture that Fireplug includes. As one can observed, the flat behaves very similarly to the hierarchical up to 8 replicas. When we increase to 12, there is a severe throughput penalty of 45%. We ran the flat version for more than 12 replicas, but the system was crashing, indicating that it would be difficult to manage more than 12 replicas in a single group.

D. Fireplug vs. Neo4j

Finally, we have also compared Fireplug with the native replication mechanism of Neo4j (we omit the graphical representation for lack of space). In our experiments, we were not able to scale Neo4j to more than 8 replicas. Neo4j uses a single-master replication scheme. We suspect that having a single master is the cause of their scalability problems. Interestingly, even with 8 replicas, Neo4j throughput was already significantly lower than the one exhibit by Fireplug (50% throughput penalty).

VI. Conclusion

We have presented Fireplug, a flexible architecture to build robust geo-replicated transactional graph-databases. Fireplug combines in a novel way ideas from N-version programming, a hierarchical Byzantine-tolerant state-machine replication, and a deferred update transactional protocol specialized for graph databases to build a cohesive, flexible replicated graph database that can be configured to tolerate different threats. Our hierarchical architecture shows better performance and scalability when compared to more traditionally used architectures such as flat and single-master. Furthermore, our evaluation shows that the optimistic read modes bring significant performance gains by slightly weakening consistency.

Future work includes the integration of self-adjusting mechanisms such as a load-balancer to dynamically adjust the load depending on the instantiated databases. We also plan on studying whether our techniques could be applied under partial replication, a more scalable setting.

Acknowledgments: This paper is dedicated to the memory of our dear colleague Prof. Lau Cheuk Lung who motivated us to pursue this work. This work was supported by the FCT via projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013; and by the CNPq/Brasil via project 401364/2014-3.

References

[1] M. Newman, "The structure and function of complex networks," *Siam Review*, vol. 45, no. 2, pp. 167–256, 2003.
[2] Neo4j, "The graph foundation for the enterprise," 2016. [Online]. Available: https://neo4j.com/
[3] OrientDB. (2016) Main page. [Online]. Available: http://orientdb.com/
[4] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," ser. ATC'13.
[5] C. Vicknair and et al., "A comparison of a graph database and a relational database: A data provenance perspective," ser. ACM SE'10.
[6] Neo4j, "Our customers," 2016. [Online]. Available: https://neo4j.com/customers/
[7] OrientDB. (2016) Customers. [Online]. Available: http://orientdb.com/customers/
[8] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.
[9] L. Mint. (2017) Beware of hacked ISOs. [Online]. Available: http://blog.linuxmint.com/?p=2994
[10] S. Brilliant, J. Knight, and N. Levenson, "The consistent comparison problem in n-version software," *ACM SIGSOFT Software Engeneering Notes*, vol. 12, no. 1, pp. 29–34, jan 1987.
[11] W. source. (2017) Top open source security vulnerabilitiess. [Online]. Available: https://www.whitesourcesoftware.com/whitesource-blog/open-source-security-vulnerability/
[12] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.
[13] L. Lamport and M. Masa, "Cheap Paxos," ser. DSN'04.
[14] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
[15] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMART," ser. DSN'14.
[16] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring Paxos," ser. DSN'12.
[17] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling Byzantine fault-tolerant replication to wide area networks," ser. DSN'06.
[18] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Customizable fault tolerance for wide-area replication," ser. SRDS'07.
[19] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," ser. DSN'12.
[20] F. Pedone and N. Schiper, "Byzantine fault-tolerant deferred update replication," *Journal of the Brazilian Computer Society*, vol. 18, no. 1, pp. 3–18, 2012.
[21] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, Dec 1985.
[22] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Os diversity for intrusion tolerance: Myth or reality?" ser. DSN'11.
[23] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, Oct 2007.
[24] A. F. Luiz, L. C. Lung, and M. Correia, "MITRA: Byzantine fault-tolerant middleware for transaction processing on replicated databases," *SIGMOD Rec.*, vol. 43, no. 1, pp. 32–38, May 2014.
[25] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.
[26] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," ser. PPoPP'08.
[27] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira, "Type-aware transactions for faster concurrent code," ser. EuroSys'16.
[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," ser. SSS'11.
[29] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," ser. SOSP'11.
[30] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," ser. SIGMOD'95.
[31] G. Bagan, A. Bonifati, R. Ciucanu, G. H. . Fletcher, A. Lemay, and N. Advokaat, "gMark: Schema-driven generation of graphs and queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 856–869, April 2017.
[32] Grid'5000, "Grid'5000, a scientific instrument [...]," https://www.grid5000.fr/, 2017.