Fireplug: Efficient and Robust Geo-Replication of Graph Databases

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

Abstract—Although graph-databases have been assuming an increasing relevance in applications that exhibit strong dependability requirements, including tolerance to malicious faults, few works have addressed Byzantine fault tolerance in this particular context, and previous attempts suffer from lack of flexibility and poor performance. This 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 database instances in a cohesive manner. Thus, Fireplug can support many different deployments, according to the performance/robustness trade-offs imposed by the target application. Our evaluation shows that Fireplug is able implement Byzantine fault tolerance without penalty when compared to the built-in replication mechanism of Neo4j, which only supports crash faults. Additionally, performance optimizations introduced by Fireplug improve the overall performance by up to 900% in geo-replicated scenarios.

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

1 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 and 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 can 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.

Despite the facts above, and somehow surprisingly, very few works have addressed Byzantine fault tolerance in this particular context. Furthermore, as we will discuss in the related work section, the few works that have attempted to do so either offer inflexible, non-customizable, solutions, present poor performance, and are prone to attacks that exploit vulnerabilities on the underlying graph-database software (see §2). In this paper, we describe the design and implementation of Fireplug, an efficient architecture to build robust geo-replicated transactional graph databases, that aims at filling this gap.

Fireplug can be configured to tolerate faults on individual machines and/or faults that compromise (or disconnect) an entire datacenter. Further, Fireplug can be configured to tolerate different types of faults, from crash faults to arbitrary faults (also known as Byzantine faults). Tolerating Byzantine faults is key to build a dependable system 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, 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, compromised users that installed it. Similar attacks could be made on graph databases, for instance, to disable fraud detection systems.

Since a vulnerability in the codebase of a given graph database can expose all running instances to correlated faults, Fireplug supports software diversity and implements N-version programming, letting different replicas deploy different graph databases [10]. This shields Fireplug from bugs and attacks on 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. This makes attacks such as Stuxnet much harder against Fireplug that attackers would need to compromise different systems, using different vulnerabilities, to be successful. Furthermore, they would need to activate the attack simultaneously on different locations, to avoid the fault to be detected by the voting mechanisms that Fireplug uses.

From the algorithmic and system's implementation perspective, Fireplug makes the following contributions:

- Fireplug offers a common interface for multiple graph databases; this interface, that we have named GRADAM, augments the underlying implementation with multi-versioned data. Multi-versioning shields read-only transactions from aborting due to concurrent update-transactions. It is key to ensure efficient reads even when stronger guarantees are provided.
- 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 novel variations of Byzantine fault-tolerant algorithms to implement both read-only and update transactions. These variations, that we named *Graph-DUR*, are based on the deferred update approach [12], a fault tolerance technique that has proved to be effective for supporting database replication. The implementations of these variants coexist in the system, and can be selected depending on the characteristics of the underlying infrastructure and of the system workloads.
- 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 has no overhead compared to the native replication scheme protocols of the graph databases even though it provides stronger consistency and tolerates Byzantine failures, as these rely on single master algorithms and do not leverage the availability of multiple replicas.
- An open-source implementation of the resulting prototype is available on github.[1]

The system has been extensively evaluated. We have observed that Fireplug shows no overhead compared to 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.

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

2 Related Work

Fault Tolerance. Fault tolerance is a fundamental property of dependable systems [13]. In this paper we are mainly concerned with tolerating two types of faults, namely *crash* faults [14] and *Byzantine* faults [15], [16]. We say that a node fails by crashing, if it operates correctly up to a point where it stops functioning. On the opposite, a node subject to a Byzantine fault can exhibit an arbitrary behaviour. Byzantine faults may have natural causes but may also be the result of a deliberate attack by a malicious intruder.

Database Replication. Many replication techniques leverage the existence of an atomic broadcast primitive like Paxos [17]—and its variants—which is one of the most widely used tools for tolerating not only crash faults but also Byzantine faults. Not surprisingly, a number of previous works have also considered the deployment of atomic broadcast protocols across multiple datacenters [18], [19], [20]. These protocols can be used to replicate databases in multiple datacenters, at different locations [21], [22].

Among these techniques, *Deferred Update Replication* (DUR) has been shown to be particularly effective [12]. In DUR, a transaction is executed only against one local replica; such that writes are locally cached, and reads are served locally. At commit time, the transaction's write and read set are sent to other replicas for validation, by means of an atomic broadcast primitive. If no conflicts are detected, writes are atomically executed, otherwise writes are discarded, and the transaction aborted. Of particular interest to us is the work of [23], 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 [23] are twofold: our implementation considers graph semantics; and we avoid digital signatures for validating reads. Avoiding signatures in this context is relevant, not only because they are computationally expensive but, above all, because they consume additional storage. Since most objects in graph databases are very small, the overhead of maintaining signatures can increase the size of the database by an order of magnitude.

Some works have shown that it is possible to offer semantics stronger than serializability by using specialized, highly precise, clock synchronization services, namely, Spanner offers linearizability [22] and FaRM [24] offers opacity. None of these systems has been designed for graph databases or to tolerate Byzantine faults. The architecture proposed here is more general and more robust, as it avoids the use of clock synchronization to ensure safety properties, which could be a liability in the Byzantine setting.

N-version Programming Research on N-version programming, or software diversity, started in the 1970s and raised considerable interest [25]. As discussed in [25], faults are often caused by design flaws. N-version programming aims at tolerating these design faults, using a range of independently-designed software components, which has also been shown to decrease the likelihood of malicious

intrusions [26]. Unfortunately, due to the large deployment effort required to combine enough distinct implementations, N-version programming is—with a few exceptions [27]—seldom used practice.

MITRA [28] 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 [29] 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.

Graph Processing. There is a recent surge of interest in large scale graph processing, starting with Google' s Pregel [30]. The objective is to process large graphs in parallel, similarly to what MapReduce does for unstructured data. Apache Giraph is an open source graph processing system that tolerates node crashes like Pregel [31]. There are several others that also tolerate crashes, such as PowerGraph [32] and Trinity [33]. Greft, instead, tolerates Byzantine faults [34]. Nevertheless, none of these systems uses replication for fault tolerance. They use checkpointing mechanisms that periodically save the state of processes in persistent storage, allowing future recovery in case of failure.

Wide Area BFT. Our algorithms are based on atomic broadcast protocols for the Byzantine fault-tolerant model, also known as BFT protocols [16]. A number of algorithms have been proposed to implement BFT in the wide-area and our work is inspired by these previous results. Most notably, Steward [20] proposes an hierarchical BFT protocol, where a node from each local group is elected to represent the group in a global group. As it will be explained later, we use the same general approach but a different implementation: while Steward requires members from the local group to sign all actions of the primary, to achieve better performance in the common case, we let the primary proceed alone, and use our knowledge about the application semantics to validate its actions before applying any updates. Recent work in the blockchain area has also started to address the wide-area performance of consensus. However, most of these works, such as [35], use dissemination strategies based on gossip, that fail to guarantee deterministic termination.

Summary of Related Work. The importance of graph databases has been growing and they are now used in applications that have strong reliability requirements. A framework that can provide Byzantine fault tolerance in a scalable, flexible, and efficient manner is needed. Many of the ingredients required to build such frameworks have been previous explored in the literature. However, they need to be adapted to operate efficiently with graph databases, and need to be integrated with each other to build a coherent whole.

3 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 significantly 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. Naturally, the cost of coordinating replicas across multiple datacenters (in our case, using atomic broadcast) may not be negligible (we discuss these costs in the evaluation section). Still, experience with large-scale geo-replicated systems, such as Google's Megastore [21] and Spanner [22] has shown that several application are willing to pay these costs to ensure liveness in face of an outage that affects an entire datacenter.

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 the overall system performance.

Note that although Fireplug has been designed to leverage software diversity at the level of the graph database, in the current prototype, the Fireplug middleware itself runs a single implementation. In theory, one could also run multiple implementations of our libraries in different nodes, but the development effort to derive such versions is out of the scope of our research project.

3.1 System Components

The main software components of Fireplug, depicted in Figure 1, are the following:

- 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. The proxy augments the underlying database to support multi-versioned data.

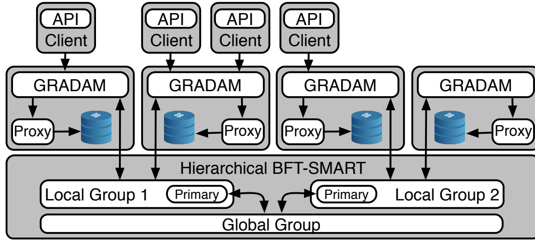


Fig. 1: Fireplug architecture.

- Several variants of a new replication protocol, *Graph-DUR*, that implements deferred update replication tolerant to Byzantine faults and specifically designed for graph databases.
- As part of its replication protocol, Fireplug implements *Hierarchical BFT-SMaRt*, an atomic broadcast abstraction implemented as a hierarchical composition of multiple instances of the BFT-SMaRt service [16]. An instance of BFT-SMaRt that has members exclusively from the same datacenter is denoted a local group. An instance of BFT-SMaRt that has members from different datacenters is denoted a global group (our architecture uses a single global group).

3.2 Proxy

Due to the software diversity support by Fireplug, we defined a common interface (access class) to be used by the Fireplug clients. The interface defines the methods that each individual access class has to implement in order to ensure interoperability. The *start* and *terminate* methods are required to start a new instance of the database and terminate it. The *beginTransaction* and *endTransaction* methods delimit a transaction. We support the most fundamental operations, namely *create*, *read*, *update*, and *delete*, to ensure that the interface can be implemented by a wide range of graph databases. Each of these operations has its corresponding method in the common interface. These methods accept, as a parameter, either a vertice or an edge object. Fireplug requires an instance of GRADAM to be implemented for each different graph database. We do not see this as an impairment for adoption, because GRADAM is relatively easy to implement. In our prototype, the implementation of GRADAM for the different databases required the following number of lines of code: 509 for Titan, 673 for Sparksee, 441 for OrientDB, and 738 for Neo4j.

3.3 System Operation

Clients connect to an instance of GRADAM, typically at the closest server, and coordinate the execution of transactions 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 threefold: (i) it bridges local clients with the local graph database replica, translating between the common interface (the one exposed to clients) and each particular graph database interface; (ii) it augments the underlying databases with multi-versioning data; (iii) 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 DUR protocol. For some operations, namely to implement update-transactions, the explicit exchange of messages 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 [16], 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 §4.2).

4 Graph-DUR: Replication in Fireplug

Replication in Fireplug is managed using several variants of the Byzantine-tolerant DUR proposed in [23]. We first discuss the major differences between our variants (that we have named Graph-DUR) and [23]. 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.

4.1 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 [36], [37] and geo-replicated distributed systems [38], [39].

First, clients can potentially merge updates before pushing them to Fireplug to reduce the number of operations that need to be managed by the underlying databases. For instance, 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, in particular, we avoid flagging conflicts when the operations are known to be commutative. 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 transactions, given that both transactions “update” the same key. Our conflict detection algorithm takes into account that both operations commute, 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 [23] this is addressed by having the servers sign every data item. In the case of relational databases this is valid because it

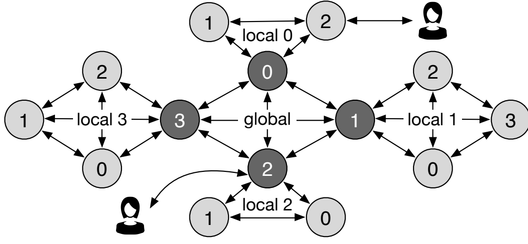


Fig. 2: Hierarchical architecture.

is possible to sign tables or sections. Nevertheless, when using graph databases, this is not a viable option. Different 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. In fact, depending on the size of the objects, adding signatures can increase the size of the database and the correlated bandwidth cost by an order of magnitude. The tradeoff is that we need to 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 §4.4.

4.2 Hierarchical Atomic Broadcast

Some variants of Graph-DUR require 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 BFT-SMaRt.

Specification. We assume a set of n processes, $\Pi = \{p_1, \ldots, p_n\}$ which may fail by crashing, or by behaving maliciously. If a process does not fail, we say it is a *correct* process, and otherwise it is a *faulty* process.

Like other atomic broadcast primitives, our hierarchical atomic broadcast is defined by the HAB-broadcast(m) operation, where m is the message to be broadcast. The HAB-broadcast(m) operation may trigger the HAB-deliver(m) event at a process. The hierarchical atomic broadcast primitive satisfies the following properties:

Validity. If a process p_i delivers a message m, some process p_j has broadcast m.

Integrity. Every process delivers a message at most once.

Agreement. If a correct process p_i delivers a message m, then all correct processes eventually deliver m.

Total order. If two correct processes p_i and p_j deliver two messages m and m', then p_j delivers m before m' iff p_i delivers m before m'.

Instantiation. 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. 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 primary [40].

The global group can be configured to tolerate f crash or f 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, then 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. The right configuration is scenario-specific, and depends both on the user requirements and on the estimated power of the adversary.

Operation. Hierarchical Atomic Broadcast is implemented as follows. Clients initiate a HAB-broadcast by sending messages directly to the global group. These messages are then totally ordered by BFT-SMaRt (where consensus is used to assign a sequence number to each message) and delivered to all members of that group. Differently from the standard BFT-SMaRt implementation, in the HAB-broadcast variant, the message is delivered together with the signed votes from a quorum of members of the global group. This quorum of signatures certifies the sequence number that has been assigned to the message (by the global group). Then, each member of the global group broadcasts the message (along with the corresponding set of signatures) to all members of the local group. For this purpose, it uses the more efficient unordered Byzantine broadcast service of BFT-SMaRt. This ensures that if a correct member of the local group receives the message, all members do receive the message. Messages are delivered to the remaining members of each local group in the order defined by the global group. Later in the evaluation section we show that this strategy is much more scalable than trying to perform atomic broadcast on a single group that contains all replicas.

4.3 Update Transactions

Fireplug implements two different variations of the Graph-DUR protocol. Both variants execute transactions against a single replica and certify concurrent transactions in total order. However, each variant uses a different communication pattern and a different set of replicas involved in the certification. We have called these variants Hierarchical-Broadcast Global-Certification (HB) and Hierarchical-Certification Global-Apply (HC) respectively. Each of these variants represents a different trade-off between CPU and Network resource consumption. We explain these variants in the following subsections.

4.3.1 *Client Algorithm*

The client algorithm is the same for both variants of Graph-DUR and is presented in Algorithm 1. The clients maintains a read-set ($t.RS$), a write-set ($t.WS$), a create-set ($t.CS$), and a delete-set ($t.DS$), where it stores all the objects that have been, respectively, read, updated, created, and deleted in the

Algorithm 1 Client code

```
1: function BEGINTRANSACTION(t, transactionClass)            ▷ Start Transaction
2:     t.RS ← ∅                                          ▷ Initialize read-set
3:     t.WS ← ∅                                          ▷ Initialize write-set
4:     t.CS ← ∅                                          ▷ Initialize create-set
5:     t.DS ← ∅                                          ▷ Initialize delete-set
6:     t.server ← best_closest_server(transactionClass) ▷ Pick a server
7:     t.ts ← ⊥                                          ▷ Transaction timestamp
8: end function
9: function READ(t,oid)                                  ▷ Read operation
10:    if (oid, v, ts)∉ (t.RS ∪ t.WS ∪ t.CS ∪ t.DS) then
11:        (oid,v,ts)← t.server.SERVER_READ(oid)
12:        t.Rs ← (oid, v, ts)
13:        if t.ts = ⊥ then
14:            t.ts ← ts
15:        else if t.ts < ts then
16:            abort
17:        end if
18:    end if
19:    return v
20: end function
21: function CREATE(t,oid)
22:    t.CS ← (oid)                                      ▷ Save in create-set
23: end function
24: function DELETE(t,oid)
25:    t.DS ← (oid)                                      ▷ Save in delete-set
26: end function
27: function WRITE(t,oid, v)
28:    t.WS ← (oidi, v)                                  ▷ Save in write-set
29: end function
30: function ENDTRANSACTION(t)
31:    if HB-mode on then
32:        HAB-BROADCAST(hb-validation, t, ts, t.RS, t.WS, t.DS, t.CS) to all servers.
33:    else                                              ▷ HC mode on
34:        ABCAST(hc-validation, t, ts, t.RS, t.WS, t.DS, t.CS) to servers in global group.
35:    end if
36:    wait for identical (t, outcome) from f + 1 servers
37:    return outcome
38: end function
```

Algorithm 2 Certification Procedure

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

course of the transaction. Updates, deletions, and creations, which are cached at the client, will only reach the servers on commit. To make sure clients read the changes made during the transaction, on each read the result will be matched with the write-set and then updated accordingly. The client starts by selecting a node against which it will execute the update transaction optimistically.

The client also keeps a variable $t.ts$, that stores the snapshot used for performing reads; all reads are consistent with this snapshot. This variable is initialized with the timestamp of the first object read by the transaction (line 14). Additionally, on each read, the timestamp of the object in the database has to be compared with the transactions *snapshotId* to guarantee the isolation property of the transaction. If the timestamp corresponds to a transaction which has been committed after the start of the current transaction, the transaction will be aborted (line 16). Since GRADAM supports multi-versioning, the database will retrieve a version matching the transactions *snapshotId* to avoid the abort.

From the client's perspective, the only difference between running transaction on HB or HC mode is how the validation of the transaction is requested when it attempts to commit (line 31). In HB mode, the validation request is sent to all replicas using the hierarchical atomic broadcast primitive described before. In HC mode the validation request is sent just to the members of the global group, using the default atomic broadcast primitive of BFT-SMaRt.

4.3.2 Certification Algorithm

The certification algorithm is presented in Algorithm 2. This algorithm can be executed by any server, as long as certification requests are processed in total order. Each node and relationship contain, additionally to the timestamp, the hash of the node itself. This will be used to check whether the server tries to send outdated, wrong or corrupted data to the client. The certification check, for a transaction ts, may be broken down into two fundamental steps:

- The validation procedures check if the transaction does not violate serializability. For that, we check that no transaction ts' with an update set that overlaps with ts update set or with ts read set has committed after ts started (lines 8–15 and 16–23);
- The validation procedures checks that the values that have been returned to the client by the delegate replica match the values stored locally. For this purpose, the procedures check if all hashes $h' \in v' \in t.rs$ are equal to their corresponding values and hashes $h \in v$ in the database.

If the transaction passes both checks it can be committed. Otherwise it needs to be aborted.

4.3.3 Hierarchical-Broadcast Global-Certification

The Hierarchical-Broadcast Global-Certification (HB) variant of Graph-DUR is the one closer to the classical DUR. In this variant, update transactions are executed following the DUR algorithm: optimistically against a single replica (typically the closest one) and validated in parallel by all

Algorithm 3 Hierarchical-Broadcast Global-Certification (HB)

```
1: function HAB-DELIVER(hc-validation, ts, t.RS, t.WS, t.DS, t.CS)
   from client c                          ▷ called in total order by all replicas
2:     (result, global_ts) ← VALIDATE_TRANSACTION(ts, t.RS, t.WS,
   t.DS, t.CS)
3:     send result to client c
4:     if result = commit then
5:         APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
6:     end if
7: end function
```

replicas in total order, to ensure serializability. As a result of this 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.

The pseudo-code is given in Algorithm 3. When the validation request is received in total order from the hierarchical broadcast protocol, all nodes apply the validation procedure described before and inform the client of the outcome. Additionally, if the transaction commits, the update set is applied to the local database. HB requires a single execution of the totally ordered broadcast algorithm at commit time. This is used to ensure that all replicas can certify the transaction in exactly the same order. This ensures that the certification, that is deterministic, yields exactly the same results at all correct replicas. The drawback of this approach is that it suffers from redundant CPU consumption, as all replicas perform essentially the same work every time an update transaction attempts to commit.

4.3.4 Hierarchical-Certification Global-Apply

The Hierarchical-Certification Global-Apply (HC) variant of Graph-DUR aims at trading processing time for communication. In this variant, a transaction is just certified by the nodes in the global group. These nodes produce a signed proof of the transaction outcome. Then, the updates associated with a transaction are broadcast together with the proof that the associated transaction was serializable and can be committed, and applied to the local datastore by all nodes without the need of going through local certification.

The pseudo code is given in Algorithm 4. In detail, when a client finishes a transaction containing update requests, it broadcasts the transaction to the global group only. The transaction is delivered in total order to the members of the global group that perform the certification (line 2), sign the result, and broadcast the result to the global group, using the unordered Byzantine tolerant primitive (line 3). Additionally, the servers send the result to the client (line 4). If the certification fails the transaction is aborted and no further steps are required. If the transaction commits, each member of the global group collects $f+1$ identical commit messages (line 6). If no quorum can be achieved, the transaction is aborted. If a quorum is achieved, the update set is applied locally and sent to all members of the local group (lines 8 and 9). Finally, when the values are received by the members of the local group, each member checks if the $f+1$ commit-votes are valid and then applies the update set in the order defined by *global_ts* (line 15).

Note that, given that $f+1$ votes to commit the transaction are enough to certify that the transaction validation

Algorithm 4 Hierarchical-Certification Global-Apply (HC)

```
1: function ABCAST-DELIVER(hb-validation, ts, t.RS, t.WS, t.DS,
   t.CS) from client c   ▷ called in total order by replicas in the global
   group
2:     (result, global_ts)← VALIDATE_TRANSACTION(ts, t.RS, t.WS,
   t.DS, t.CS)
3:     RBCAST(hc-result, ts, result) to servers in global group.
4:     send result to client c
5:     if result = commit then
6:         result-set ← collect f+1 RBCAST-DELIVER(hc-result, ts, com-
   mit) from other servers in the global group
7:         if |result-set| ≥ f+1 then
8:             APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
9:             RBCAST(global-apply, ts, result-set, global_ts, t.WS,
   t.DS, t.CS) to all server of my local group
10:        end if
11:    end if
12: end function
13: function ABCAST-DELIVER(global-apply, ts, result-set, global_ts,
   t.WS, t.DS, t.CS)
14:    if result-set is valid then
15:        APPLY_UPDATES(global_ts, t.WS, t.DS, t.CS)
16:    end if
17: end function
```

succeeded, it is not strictly required that all members of the global group certify the transaction. Therefore, in the most common case (i.e, when no Byzantine faults occur) it is possible to load balance the work of certifying the transactions among the members of the global group. It is clear that HC, even with this optimization, requires significantly more communication than HB, given that $f+1$ votes need to be collected before the update set can be applied. However, instead of requiring all nodes to validate the transaction, with HC only a quorum of nodes need to perform this task. In scenarios where capacity of nodes is very heterogeneous, HC can bring advantages since it is possible to offload the validation work to the more powerful machines.

4.4 Read-only Transactions

To improve performance, read-only transactions are executed differently from classical DUR. Fireplug supports read-only transactions with several different resilience levels. One dimension of resilience is concerned with the geographical scope of the fault: *locally-safe* reads, that cross check the results using two nodes 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). The other dimension of resilience is concerned with the database instance: *version-unaware* reads do not care about the database instance when validating reads and *version-aware* reads always validate read transactions using different graph databases (in order to resist to vulnerabilities that compromise all instances of the same graph database).

Furthermore, Fireplug considers 3 different implementations of each resilience level, namely: *pessimistic* read-only transactions (PR), *optimistic latency-driven* read-only transactions (OR-L), and *optimistic throughput-driven* read-only transactions (OR-T). We describe the three different implementations below:

- Pessimistic read-only transactions are executed as these were update transactions by broadcasting them, in

total order, to all replicas in the global cluster. Thus, pessimistic read-only transactions are as expensive as update transactions. Nevertheless, this implementation has some advantages when compared to the other two implementations described below. First, it guarantees that all replicas have a state consistent with the local replica when validating the read transaction. This ensures that, if the local replica is correct, the validation of the read transaction always succeeds. Second, even if the local replica is faulty, the fault is always detected.
- Optimistic latency-driven also broadcasts the read-only transaction to all replicas, but use the unordered broadcast primitive instead. Therefore, when remote nodes attempt to validate the transaction, they may not have an up-to-date state. Nodes that are up-to-date (i.e., that have already installed the snapshot that was used for the client when executing the read transaction), can validate the transaction and send the result back to the client. The client can continue as soon as it received f additional validation confirmations from the desired set of replicas (the exact set depends on the desired resilience). This ensures a quicker validation, although it still requires global communication.
- Finally, optimistic throughput-driven reads is the least expensive implementation. The read transaction is broadcast to f other nodes with the desired resilience properties using point-to-point communication. This avoids global communication. Unfortunately, if the selected targets are not up-to-date, the client may be forced to select additional targets, which may increase the latency of reads.

In both optimistic executions if the read transaction returns an abort, read transactions are re-executed in pessimistic mode since the client cannot be sure if he read wrong or just outdated data. 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.

4.5 Configuring Graph-DUR

As described before, Graph-DUR must be configured differently depending on the type of failures one wants to tolerate. In this section, we discuss how Graph-DUR should be configured to tolerate three types of failures: crash faults, Byzantine nodes, and Byzantine datacenters. We also briefly discuss how Graph-DUR can detect these failures and how to recover from them.

4.5.1 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 protocol. 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 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 primary.

4.5.2 Tolerating Byzantine Nodes

A Byzantine node can be a non-primary or a primary. If it is a non-primary, any misbehaviour will be detected by a client, that will not be able to validate the result of the transaction executed against the faulty node. The client simply selects another server (the client may also report the fault, such that the server is eventually replaced). If the faulty node is a primary we need to consider two scenarios, namely, the faulty node can misbehave in the global group or in the local group (or in both). We discuss these two scenarios below.

If a faulty primary misbehaves in the global group in a manner that prevents the global group to make progress (for instance, it it is the global group's primary), it will be suspected by the correct nodes in the global group, forcing a view change. The correct members of the global group will then report this fact to the affected local group, such that a new primary is elected. A faulty primary may also behave correctly in the global group but fail to forward transactions to its local group. To detect this scenario, clients expect a confirmation that a write transaction has been applied from $f+1$ members of each local group. Failure to collect such quorum from a given local group will also lead to the corresponding primary being suspected and to the election of a new primary.

4.5.3 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.

5 Evaluation

In this section, we present the results of an experimental study of Fireplug. We answer the following questions: What are the differences in performance among the two update protocols (HB and HC) and how these compared with classical DUR based on a flat group? What is the impact of multi-versioning? How do the different read-only protocols perform? How well does Fireplug perform in terms of throughput when compared with the native replication schemes of the underlying graph databases?

The evauation shows not only the absolute performance of Fireplug but also its overhead with regard to different baselines: §5.1 compares the performance against non-hierarchical protocol, § 5.5 compares with the native replication scheme of Neo4j, and § 5.3 shows the overhead of the multi-version features.

In the experiments we use five different workloads: Read-only (100% Reads), Read-heavy (0.2% Writes), Conflict-prone (5-20% Writes), Balanced (25-75% Writes), and Write-only (100% Writes).

All workloads are based on real usage of graph databases (such as Facebook's TAO [4]) just varying the different write levels. The dataset has been synthetically generated using gMark, a well-founded approach for schema-driven generation of graphs and query workloads [41]. It represents a social-network and has 100,000 vertices and approximately 230,000 edges. Each experiment was executed three times in order to avoid statistical errors, and the results presented are the means of the throughput (operations per second) observed on all servers during a 6 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.

The experiments were carried out in the Grid'5000 testbed [42]. Each database replica runs on a separate dedicated machine with 64GB RAM and eight 2.0GHz cores. Client instances were started on the same physical machines as the server they connect to. Some scenarios use multiple data centers: scenarios with 8 servers use two data centers, each with 4 servers each (2 representatives in the global group) and scenarios with 12 servers use 4 data centers each with 3 servers (one representative in the global group). Because the latency among Grid'5000 clusters is very small, we emulated larger geo-replicated scenarios by using `netem` [43] to add a 20ms delay in the links that connect machines from different datacenters.

5.1 Update Transactions: HB Flat vs HB Hieraquical

To compare the performance of the different protocols used to process update transaction we use a write-only workload and assess what is the maximum throughput achievable with each protocol in different scenarios. We set the percentage of writes to 100%, to assess the load writes impose on the system in both configurations. We gradually increased the number of clients until the throughput reached a plateau, to capture the maximum capacity of that configuration: for the flat configuration the max-throughput was achieved by using 5 to 60 clients depending on the number of servers; for the hierarchical configuration the number of clients ranged from 30 to 60. The results are depicted in Figure 3, showing the difference in percent of both approaches for different numbers of nodes. As it can be observed, the advantage of Fireplug increases significantly with an increasing number of servers up to 900% with 12 servers. This is due to the fact that the message complexity of the algorithms grows quadratically with the number of members in each group. We chose to display the difference in performance in percent to highlight the increasing difference between both approaches with an increasing number of servers.

5.2 Update Transactions: HB vs HC

We now compare HB with Hierarchical-Certification Global-Apply (HC). As discussed in § 4.3, HB is expected to have an advantage over HC in most execution cases since the primaries do not have to calculate extra signatures and also do not have to exchange them before sending them to the local replicas. Nevertheless, HB puts more load on the local (non-primary) replicas than HC, given that with HB local replicas also have to execute and verify the transactions. Following our analysis, the time to execute and verify a transaction takes, on average, more than twice the time it takes to do the verification only. This means that with HB local replicas are subject to approximately twice the load than local replicas running HC (for each write transaction).

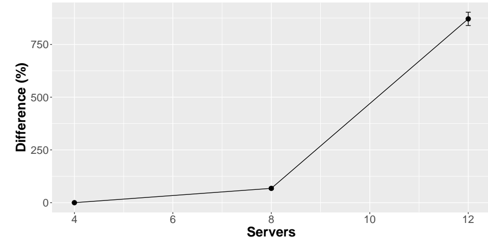


Fig. 3: Comparison between flat and Fireplug's throughput.

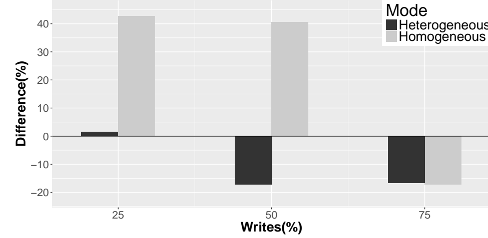


Fig. 4: Comparison between HC and HB's throughput.

From the discussion above, one can reason as follows. If the system is not overloaded, HB is expected to perform better, because it requires a smaller number of communication and processing steps. If the system is overloaded and machines are homogeneous, the performance of both HC and HB will fall. However, it is possible to sustain more load by upgrading only machines in the global cluster and switching from HB to HC, given that HC alleviates the burden on the local replicas. Our experimental results, confirm this analysis.

Figure 4 shows the performance of HB vs HC in two scenarios: a scenario where all machines have high capacity (homogeneous) and a scenario where only the machines in the global cluster have high capacity, and the local replicas have lower capacity (half the capacity of primaries). With 25% writes, the local replicas are not overloaded in any of the scenarios, thus HB performs much better than HC (almost 40% better). However, with 50% writes, in the scenario where local replicas are less powerful, they can no longer keep-up with the load, thus HC outperforms HB (by almost 20%). Finally, with 75% writes, even when all machines have high capacity, HC performs better. In this case, the amount of writes saturates even the more powerful machines but, since local replicas have less work in HC, they have some free cycles to serve read transactions improving the overall throughput. In summary, HB should be used in steady state (we assume the system is provisioned to avoid overload in normal operation) and HC can be used to mitigate the effects of workload peaks. A fully discussion of self-adaptation mechanisms for Fireplug is out of the scope of this paper. We just point out that we also implemented sensors on each replica that monitor the size of the request queues and that allow to detect when nodes are overloaded.

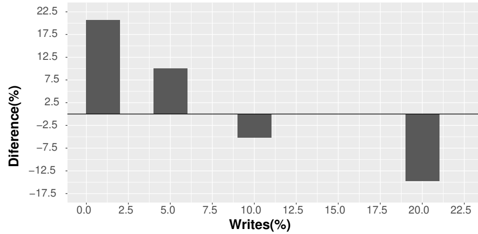


Fig. 5: Single- and multi-versioned database throughput.

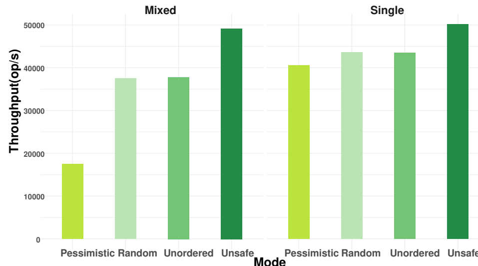


Fig. 6: Throughput for read-modes with 4 servers

5.3 Impact of Multi-Versioning Features

The graph databases we use do not natively support multi-versioning. However, our middleware enhances these graph databases with the multi-versioning features. We now study the impact of these extensions. On one hand, these features introduce additional complexity in the GRADAM layer, thus they have the potential to slowdown the performance of Fireplug in contention-free scenarios. On the other hand, multi-versioning shields read-only transaction from aborts, so it may offer improvements in scenarios where contention exists. In particular, read transactions can read from a consistent snapshot, defined at the time the transaction starts, even if other transactions update the graph concurrently. This allows read transactions to commit in scenarios where a single-version database would cause them to abort.

We assess this by running our hierarchical framework with a total of 8 replicas with increasing write-load (1-20%) in two different settings: With multi-versioning enabled and without multi-versioning. Additionally, we query only 10% of the dataset to increase contention. The results are presented in Figure 5. It can be seen that, with very little writes multi-versioning adds an overhead of over 20% to the system. However, with an increasing percentage of writes the advantages of multi-versioning start to compensate this overhead. In the case of 10% of writes multi-versioning already increases the performance of the system by 5%. Increasing the writes 20% further increases the advantage of multi-versioning to 15%, as expected.

5.4 Read-only Transactions

In order to show the differences between the proposed read-modes, we performed experiments using both a single group and an hierarchical setup with 4 datacenters, each one with 4 servers.

For the single group scenario, we experimented two different deployment scenarios: a deployment that uses multiple implementations of the database (namely Neo4j and OrientDB) where the average latency among servers was 20ms (dubbed ”mixed/wide-area” setup) and a deployment with a single database (Neo4j) and an average latency among servers of 1ms (dubbed ”mixed/local-area” setup). In both cases we executed 100% reads. The results of these experiments are depicted in Figure 6. As expected, the unsafe execution shows the highest throughput of all four alternatives and shows no significant difference in the single or mixed database setup. Also, as expected the pessimistic (ordered) verification shows the worst performance and greatly varies between the single database and mixed database setup. This happens since in the single database setup we only use the fastest database implementations and, therefore, the performance bottleneck is still in the network, while in the mixed setup the performance bottleneck shifts to the slowest database as it has to verify all messages as well. For the same reason the unordered and random access executions show a slightly improved performance in the single database setup. Interestingly, both show almost identical performance, as the unordered execution has the advantage of having only to wait for $f+1$ equal responses, while the random verification eventually hits a slower database and has to wait for its response.

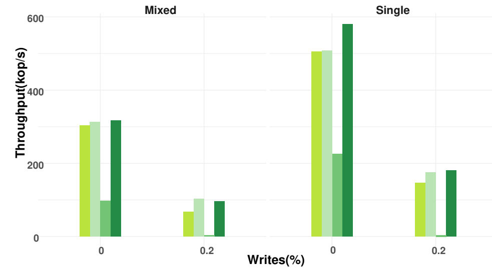


Fig. 7: Throughput for read-modes with 12 servers

For the hierarchical scenario with 12 servers, depicted in Figure 7, we set the latency among members of the local groups to 1ms and the latency among members of the global group to 20ms. As before, we used scenarios with a single database implementation (Neo4j) and with multiple implementations (Neo4j and OrientDB). Finally, we subjected both scenarios to a read-only workload and to a read-heavy workload. The results follow the trend observed in the single group case. As expected the unsafe execution shows a better performance in almost all cases. Only in the mixed setup with 0.2% the pessimistic execution shows a small disadvantage since OrientDB is unable to handle as many write requests early in the execution and ends up falling behind significantly. Additionally, as depicted, the overall performance depends heavily on the database setup (single vs mixed) and on the write levels (0% vs 0.2%). We considered very low write levels since the focus of this experiment was to measure the the performance of the system in a read-dominated setting similar to values observed by Facebook in [44].

The pessimistic (ordered) execution performance drops heavily when introducing writes into the system as expected, since reads are now queued behind the writes which leads to a strong performance toll. Additionally, in this setup the difference between the local and global verification is

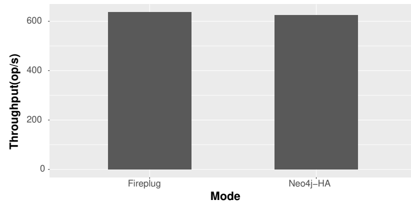


Fig. 8: Neo4j-HA vs Fireplug

stronger than in Figure 6 since in the previous experiment the global cluster received only requests from the one local cluster and not from multiple local clusters, but in the current execution, the global cluster has to handle requests from multiple datacenters which results in a bottleneck.

5.5 Fireplug against Native Replication

Finally, we compare the performance of Fireplug against the best performing native replication system of all databases we used. In particular we compared a deployed of a single group running 4 replicas of Neo4j (Neo4j-HA) and a similar cluster running an homogeneous deployment of Fireplug. We compared the Byzantine-tolerant version of Fireplug. Since the treatment of write-transactions is the main difference between Neo4j-HA and Fireplug we executed the experiments with 100% writes.

Results are depicted in Figure 8. As shown in the bar graph, Fireplug shows a 2% performance advantage over Neo4j considering 100% writes even though Fireplug has been configured to tolerate Byzantine failures and Neo4j only tolerates crash faults. Additionally, due to the total order and deferred update replication, Fireplug provides serializability while Neo4j provides only snapshot isolation. This happens since Neo4j relies on a single master replication scheme and has a complicated locking mechanism which results in a lot of transactions running into deadlocks which then have to be aborted. Fireplug avoids this issue by executing the transactions in total order, aborting only when conflicts arise, which has a lower impact on the system performance.

6 Conclusion

We 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 traditional architectures such as flat and single-master. Our evaluation also 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 UIDB/ 50021/ 2020; 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," in *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose (CA), USA, 2013, pp. 49–60.

[5] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proceedings of the 48th Annual Southeast Regional Conference*, Oxford, Mississippi, 2010, pp. 42:1–42:6.

[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. Leveson, "The consistent comparison problem in n-version software," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1481–1485, Nov. 1989.

[11] White source. (2017) Top open source security vulnerabilities. [Online]. Available: https://www.whitesourcesoftware.com/whitesource-blog/open-source-security-vulnerability/

[12] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Boston (MA), USA, 2012, pp. 1–12.

[13] J. Laprie, "Dependability: A unifying concept for reliable computing and fault-tolerance," in *Dependability of Resilient Computers*, T. Anderson, Ed. BSP Professional Books, 1989.

[14] L. Lamport and M. Massa, "Cheap paxos," in *Proceedings of the 34th IEEE/IFIP International Conference on Dependable Systems and Networks*, Florence, Italy, 2004, pp. 307–314.

[15] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[16] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Atlanta (GA), USA, June 2014, pp. 355–362.

[17] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[18] P. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proceedings of the 4snd IEEE/IFIP International Conference on Dependable Systems and Networks*, Boston (MA), USA, Jun. 2012.

[19] 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," in *Proceedings of the 36th IEEE/IFIP International Conference on Dependable Systems and Networks*, Philadelphia (PA), USA, 2006.

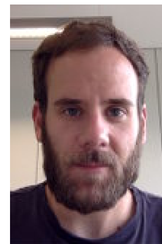
[20] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, Jan. 2010.

[21] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Fifth Biennial Conference on Innovative Data Systems Research*, Asilomar (CA), USA, Jan. 2011.

[22] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013.
[23] 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.
[24] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro, "Fast general distributed transactions with opacity," in *International Conference on Management of Data*, Amsterdam, The Netherlands, Jun. 2019.
[25] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, Dec 1985.
[26] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Os diversity for intrusion tolerance: Myth or reality?" in *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, Hong Kong, China, Jun. 2011, pp. 383–394.
[27] 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.
[28] A. Luiz, L. Lung, and M. Correia, "MITRA: Byzantine fault-tolerant middleware for transaction processing on replicated databases," *SIGMOD Record*, vol. 43, no. 1, pp. 32–38, May 2014.
[29] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.
[30] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, Indiana, USA, 2010, pp. 135–146.
[31] Apache. (2017) Apache Giraph. [Online]. Available: http://giraph.apache.org/
[32] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *Proceedings of the 10th Symposium on Operating System Design and Implementation*, Hollywood (CA), USA, 2012.
[33] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 International Conference on Management of Data*, New Yor (NY), USA, 2013, pp. 505–516.
[34] D. Presser, L. C. Lung, and M. Correia, "Greft: Arbitrary fault-tolerant distributed graph processing," in *Proceedings of the 4th IEEE International Congress on Big Data*, New York (NY), USA, 2015, pp. 452–459.
[35] G. Danezis and D. Hrycyszyn, "Blockmania: from block dags to consensus," University College London, Tech. Rep., 2018.
[36] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City (UT), USA, 2008, pp. 207–216.
[37] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira, "Type-aware transactions for faster concurrent code," in *Proceedings of the 11th European Conference on Computer Systems*, London, United Kingdom, 2016.
[38] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, Grenoble, France, 2011, pp. 386–400.
[39] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, Portugal, 2011, pp. 385–400.
[40] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[41] 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.
[42] Grid'5000, "Grid'5000, a scientific instrument designed to support experiment-driven research," https://www.grid5000.fr/, 2017.
[43] Netem, "Network emulator," 2019. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
[44] Facebook. (2016) Tao: The power of the graph. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/tao-the-power-of-the-graph/10151525983993920/

Ray Neiheiser has earned his masters degree in Computer Science from the Federal University of Santa Catarina (UFSC) and he is now a PhD student at the same university. His area of interest is the construction of reliable distributed systems. In the context of this, he specifically focuses on Byzantine fault-tolerant consensus and distributed ledger technology.

Luciana Rech is an Associate Professor at the Informatics and Statistics Department (INE) of the Federal University of Santa Catarina (UFSC) and a member of the Distributed Systems Research Laboratory (LAPESD). Graduated from University of Cruz Alta in Computer Science, with Master's degree in Computer Science from Federal University of Santa Catarina and Ph.D. in Electrical Engineering from the same university.



Manuel Bravo is a post-doctoral researcher at the IMDEA Software Institute in Madrid, Spain. He has a double-degree PhD in Computer Science from the Universidade de Lisboa and the Université Catholique de Louvain. His interest is in the design and implementation of distributed systems. Specifically, he is interested in understanding replication and consistency in such systems.

Luís Rodrigues is a professor at Instituto Superior Técnico, Universidade de Lisboa and a researcher at INESC-ID. His research interests lie in the area of reliable distributed systems. He is co-author of more than 150 papers and 3 textbooks on these topics. He is a senior member of the IEEE and of the ACM.

Miguel Correia is is associate professor with habilitation at Instituto Superior Técnico, Universidade de Lisboa, and senior researcher at INESC-ID. He has more than 150 publications and is Senior Member of the IEEE. His research focuses on cybersecurity and dependability (aka fault tolerance), typically in distributed systems, in the context of different applications (blockchain, cloud, mobile).