DOI: 10.15439/2021F46
ISSN 2300-5963 ACSIS, Vol. 25

Fed-agent – a Transparent ACID-Enabled Transactional Layer for Multidatabase Microservice Architectures

Lazar Nikolić
University of Novi Sad, Faculty of Technical Sciences
Email: lazar.nikolic@uns.ac.rs

Vladimir Dimitrieski
University of Novi Sad, Faculty of Technical Sciences
Email: dimitrieski@uns.ac.rs

***Abstract*—With the recent expansion of specialized databases and departure from the "one size fits all" paradigm, engineers might decide to use multiple databases. Each database holds a representation of a data object but offers transactions and consistency guarantees only locally. Existing solutions either require additional coding or do not provide global ACID transactions. In this paper, we present fed-agent, a transactional layer that provides global consistency and ACID transactions for single data objects within multidatabase systems. It requires no additional coding besides configuration files. We show that fed-agent scales linearly and introduces an overhead small enough for most microservice solutions.**

I. Introduction

Since the mid-2000s, the database community has shifted away from the "one size fits all" approach to database systems [1]. Many new specialized databases have emerged over the past decade, such as VoltDB [2] and Neo4j [3]. These databases are sometimes able to outperform relational databases by a large margin for their specialty workloads [4] [5]. An argument can be made that to be as performant as possible, a system should incorporate various types of databases. Data objects would be stored in different representations across multiple databases, but still logically represent a single entity. A blog post, for example, could have its content fully stored in a relational database. Its text content could be also stored in a text search engine, while viewership statistics calculated based on it could be stored in an analytical database. There needs to be a mechanism to synchronize data object representations and offer global consistency. It is important to note that databases usually cannot extend their consistency and transactional properties beyond their scope. For example, having only databases with Atomicity, Consistency, Isolation and Durability (ACID) transactions will not automatically make all distributed transactions ACID.

Traditionally, the two-phase commit (2PC) [6] protocol is used for data synchronization when high consistency is required, but at the cost of lower throughput [7]. Persistent message queues can be used instead, at the cost of less strict consistency [8], limiting the design space, increasing development costs, and harming the developer and user experience. Another usual approach is to implement Extract-Transform-Load (ETL) processes, usually as scripts that migrate data periodically between databases. ETL adds extra costs to implementing and maintaining migration scripts, which are subject to change on each schema update.

The microservice architecture is a widely used architecture for building cloud-based software solutions. Microservices expose an Application Programming Interface (API) that is used to access or manipulate data stored in databases. Each microservice knows how to transform request data into a persistent format used by a database. If most database operations are handled through API calls, the messages should contain enough information to deduce the state of the database.

In this paper, we present *fed-agent*, a transactional layer acting as a proxy that aims to provide global consistency and ACID guarantees for multidatabase systems. Other solutions either require additional coding or do not support ACID over multiple databases. Fed-agent does so for single data objects with no additional code needed besides configuration files. This way distributed transaction processing is facilitated transparently, allowing the engineers to focus on the business logic. In this paper we also prove that fed-agent can provide the above perks with a low overhead and linear scaling.

The rest of the paper is organized as follows. In Section II we present the architecture and core functionalities of fed-agent. The fed-agent evaluation results are given in Section III. In Section IV we discuss the related work, while in Section V we conclude the paper and discusses future work.

II. Architecture

Fed-agent acts as a layer that unifies read and write operations on a single data object over multiple databases. It communicates with databases through microservices built on top of them. This eliminates the need to write and maintain code that translates data between various representations, as mappers are already implemented as a part of microservices.

Fed-agent consists of multiple nodes within a single Raft [9] consensus group, with one leader and multiple followers. Only the leader can accept writes, while followers can serve read requests. In case of a leader's failure, one of the followers will become the leader so the fed-agent can continue accepting writes. We present the high-level overview of fed-agent architecture in Fig. 1. It shows a fed-agent cluster consisting of three nodes, with *fed-agent 1* being the leader. *Service 1* and *Service 2* are microservices with separate databases. A client sending writes to any service does so via *fed-agent 1,* while reads can be served by any node. For example, a read request can be routed to *fed-agent 3*.

Fed-agent identifies an operation and the data object being read or modified based on the HTTP request's body and URL. For example, *PUT "/users/123"* implies a user object with ID "123" is being updated with data from the body. Rules for extracting the information from a request are defined declaratively in a configuration file for each endpoint.

Fed-agent uses Multi-Version Concurrency Control (MVCC) [10]. Every write request on a data object creates a new version of the data object. Reads can only see committed data object versions based on its timestamp. If fed-agent detects a microservice response contains an uncommitted data object, it will replace it with the last committed object available for the request's timestamp. MVCC implementation ensures snapshot isolation level, leaving only anomalies that happen for predicate-based operations. Since fed-agent operates only on single data objects, these anomalies cannot happen and serializable isolation level is ensured.

III. Evaluation

To evaluate fed-agent, we created a setup that illustrates a typical scenario in practice. The setup consisted of a fed-agent cluster acting as a proxy for microservices connecting to a database. The microservices are *userservice*, using *PostgreSQL 13*, *textservice*, using *Elasticsearch 7.12.1*, and *geosvc* using *Tile38* [11]. Each microservice is written in the *Go* programming language and uses only low-level database drivers and the standard HTTP library. There are two data object types: *User* and *Tweet* types. The *User* type consists of four text fields: *id*, *handle, email*, and *password*. The *Tweet* type consists of three text fields: *id*, *userHandle*, and *content*, and two double-precision numbers representing *latitude* and *longitude*. For simplicity, we store full data objects in all databases. The services offer REST API with two operations: (i) access a single data object by id and (ii) upsert a data object. All services are developed as if used in isolation and contain no code for distributed transactions. Distributed transactions and coordination are covered by fed-agent configuration files.

We use *Yahoo Cloud Serving Benchmark* (YCSB) [12] workload in our benchmarks. YCSB consists of six workloads: Workload A (50/50 read/write), Workload B (95/5 read/write), Workload C (read-only), Workload D (read latest), Workload E (top 100 records), and Workload F (read-modify-write). We do not consider Workload E since fed-agent does not currently support scans. We also add *Load* workload consisting of 100% write operations.

Overheads are measured as a difference between latencies from direct API calls to a microservice and API calls via fed-agent proxy. For write (upsert) operations, all backend

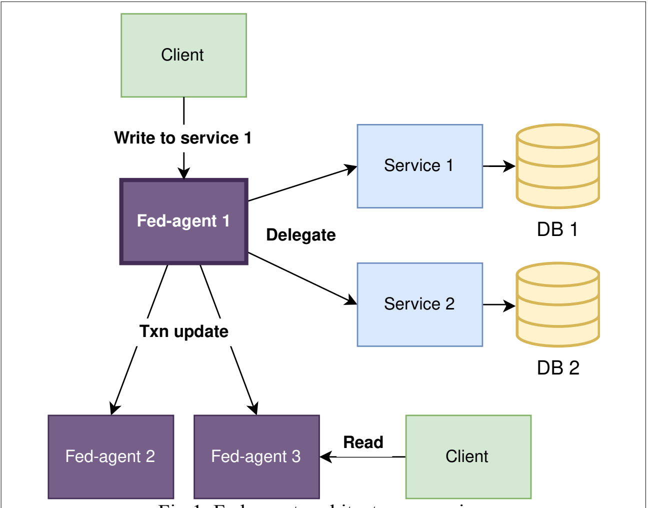

Fig 1. Fed-agent architecture overview

requests are done in parallel, so we measure the difference between the slowest microservice response and the fed-agent proxy response.

We acknowledge some threats to validity of this evaluation. Because all benchmarks were running on the public cloud and not on dedicated hardware, benchmarks results are subject to factors beyond our control, such as changes of network topology, network hiccups, or hardware failure. To minimize the effect, we ran each benchmark five times and reported the average. We also set the number of operations for each benchmark to 150000. Additionally, the bare-bones microservices used in benchmarks are developed by us for this sole purpose. Microservices in practice would have more complicated logic and would be running in much more complex architectures. Running these benchmarks in such a system might yield different results.

We run benchmarks to analyze how network overhead, payload size, number of fed-agent nodes, concurrency, and contention affect fed-agent. The results are discussed below.

Network overhead. This benchmark aims to measure the network overhead created by additional messages, so we ran it with a single client thread. We ran this benchmarks on an Amazon EC2 cluster of 10 *t1.micro* nodes. One node was acting as a client. Three nodes were running a fed-agent cluster. The three services connecting to distinct databases were each running on a different node. The databases were all running on their nodes. This benchmark considers all YCSB workloads. The results are shown in Table 1 and show that for a typical web-oriented microservice solution, one can expect about 7ms overhead for writes and about 1ms overhead for reads.

Payload size scaling. For this benchmark, we measure how much payload size can affect latency. Setup is identical to the one used for network overhead. YCSB Workload A is used to measure how payload size affects both reads and writes. Results are shown in Fig. 2.a. and Fig 2.b. Read overhead shows no clear correlation with payload size: increasing the payload size seems to yield results explained as usual

TABLE I.
AVERAGE OVERHEAD IN MILLISECONDS

Type	Workload A	Workload B	Workload C	Workload D	Workload F	Load
Read overhead (ms)	0.828	0.595	1.095	1.056	0.843	N/A
Write overhead (ms)	7.354	7.749	N/A	7.383	7.289	6.522

response time fluctuations. On the other side, write is heavily affected by the payload size. Overhead increases from about 10ms to 20ms for the first 100kB, which is the largest jump. Increasing the payload size from 100kB starts adding roughly 5ms per 100kB added. The main reason behind this increase is due to writes sending Raft messages and backend HTTP requests for every client request.

Node scaling. This benchmark measures the overhead of Raft messages as the number of nodes increases. Only writes are used because reads are strictly single node, meaning that only YCSB Load workload was considered. We ran the benchmarks on an Amazon EC2 cluster consisting of 10 *t1.micro* nodes all running fed-agent. We modified fed-agent to automatically commit all transactions because delegating HTTP requests is not affected by the number of fed-agent nodes. We are also only interested in the network overhead, so only a single client thread is used. The results are shown in Fig. 2.c. Since there is no need for consensus protocol for only one node, there is a spike in overhead when a second node is added, going from 0.4ms to 8ms. Each node added beyond the first adds 0.2ms-1ms latency. There is no indication this does not stay true for exceptionally large clusters, i.e., clusters of many tens or even hundreds of nodes.

Concurrency and contention. In the concurrency benchmark, we measured how overhead scales with an increasing number of concurrent users, but no contention. We define contention as the percentage of transactions accessing *hot data* that was 20% of total data objects. In the contention benchmark, we altered contention from 0% to 100% with a constant number of concurrent users. In both benchmarks, a concurrent user was represented by a thread running the YCSB Workload A benchmark. We capped the throughput at 3000 operations/second as to not overload the system that supported 5000 operations/second. The purpose of the benchmark was to measure the overhead of a usual workload and not to push the system to its limits. We ran the benchmarks on an Amazon EC2 cluster consisting of 7 *t3.2xlarge* nodes. Three fed-agent nodes, three *usersvc* instances using *PostgreSQL 13*, and a single client instance were deployed. *PostgreSQL* isolation level was set to serializable to match that of fed-agent, so we can more accurately compare the number of aborts.

Fig. 2.d. shows how the number of concurrent users affects the overhead. For this benchmark, we ran 100% read and 100% write workloads separately. The contention rate was set to 0%. Read overhead is mostly affected by fed-agent internal locking mechanism when fetching data object versions from the version chain. There appears to be a large spike going from 0.5s to 4s somewhere between 20 and 40 threads, which then plateaus before having a sharp drop to 0.9s at 100 threads. Our initial assumption is that at this point, disk I/O becomes the bottleneck, effectively masking network overhead of fed-agent. Write overhead is affected by the same locking mechanism when inserting versions to the chain, but also by the number of messages Raft transport can support, with the latter being the bigger factor.

Fig. 2.e. shows how contention rate affects the overhead, while Fig 2.f. shows how it affects the number of aborts. We set the number of threads to 40 for the benchmark. The overhead and the number of aborts appear largely unaffected by contention. The number of aborts is almost ten times higher than the baseline, which we attribute to distributed transactions simply being slower and causing more conflicts. The

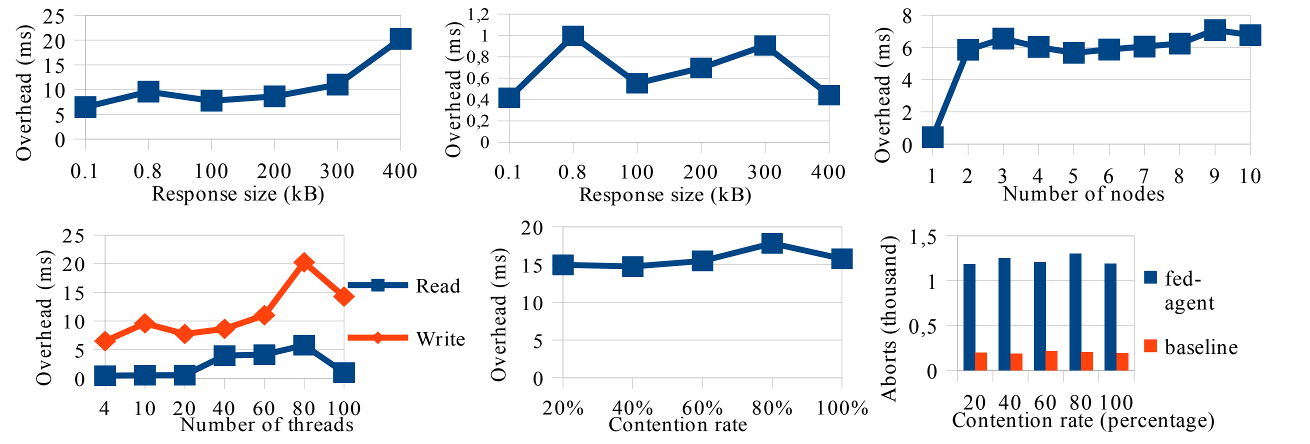

Fig 2. Benchmark results, left to right, top to bottom: a) payload size write scaling, b) payload size read scaling, c) write scaling with the number of nodes, d) concurrency scaling , e) contention scaling, f) aborts in relation to contention

total number of aborts is 1000-1200, which is small compared to the total number of operations, which is 150000.

Summary. The expected overhead for most use cases is 7-10ms for writes and 1ms for reads. For single-node fed-agent deployments, overhead for both reads and writes is less than 1ms. The highest measured overhead is 30ms for payload sizes of 400kB. This is an extreme scenario because most REST API payload sizes are 1-2kB [13]. In our opinion, fed-agent's overhead should be acceptable in microservice architectures with usual response times that are at least an order of magnitude higher, usually tens of milliseconds or longer. Fed-agent scales linearly: additional nodes add no overhead for reads and add 0.2ms-1ms overhead for writes.

IV. Related work

Distributed transactions in multidatabase systems are a well-known topic that can be traced back to the 1980s [6] [14]. It has seen some recent development, particularly in the domain of distributed databases and microservices.

ReTSO [15] has an architecture similar to fed-agent and serves as a global transaction tracker. Unlike ReTSO, fed-agent is an integrated solution that does not use any other components. GRIT [16] optimistically executes transactions, capturing write and read sets and then asynchronously applying them. It is not stated whether database service handling the write sets are an existing part of a microservice, or a component specifically developed for GRIT. Calvin [7] is conceptually similar to fed-agent but uses databases as backends instead of microservices. Deuteronomy [17] separates transactions from databases allowing it to execute them on multidatabase systems. Typhon [18] offers snapshot isolation level of access to single keys when accessed via Cerberus protocol, but not transparently. Dey et al [19] provides a Java client library for tracking transaction meta-data. The system heavily relies on test-and-set operations, limiting the choice of databases that can be used. None of the listed systems can automatically and transparently facilitate distributed transactions, but instead require using a client library.

V. Conclusion and future work

In this paper, we presented fed-agent, a transactional layer that provides ACID capabilities on single data objects over multidatabase microservice architectures. Each microservice can be developed to use a different database, regardless of consistency levels or ACID properties of the database. Fed-agent provides transparent transactions triggered automatically whenever an operation matching the provided configuration is detected. Microservices can be developed in isolation and require no code that implements distributed transactions. This brings the focus of engineers away from coordination into business logic. Our experiments show low overhead and linear scalability for a typical microservice setup.

There are several areas in which fed-agent can be improved in the future. First, we can allow users to declaratively define mappings between request types instead of being forced to use identical requests for all microservices. Second, the system can be split into shared-nothing partitions. Third, multi-object open transactions can be supported by introducing operations akin to SQL *BEGIN*, *COMMIT,* and *ABORT*. Fourth, operations reading ranges of data objects (scans) should also be supported as it is one of the common workloads described by the YCSB.

References

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, “The end of an architectural era: it's time for a complete rewrite,” Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker, pp. 463–489, 2018.

[2] VoltDB, 10-May-2021. [Online]. Available: https://www.voltdb.com/. [Accessed: 23-May-2021].

[3] Neo4j Graph Database Platform, 13-May-2021. [Online]. Available: https://neo4j.com/. [Accessed: 23-May-2021].

[4] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, “The BigDAWG polystore system and architecture,” 2016 IEEE High Performance Extreme Computing Conference (HPEC), 2016.

[5] P. Bakkum and K. Skadron, “Accelerating SQL database operations on a GPU with CUDA,” Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units - GPGPU '10, 2010.

[6] C. Mohan, B. Lindsay, and R. Obermarck, “Transaction management in the R* distributed database management system,” ACM Transactions on Database Systems, vol. 11, no. 4, pp. 378–396, 1986.

[7] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin,” Proceedings of the 2012 international conference on Management of Data - SIGMOD '12, 2012.

[8] W. Vogels, “Eventually consistent,” Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009.

[9] D. Ongaro, and J. Ousterhout, „In search of an understandable consensus algorithm”. In 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14) , pp. 305-319, 2014.

[10] P. A. Bernstein and N. Goodman, “Multiversion concurrency control—theory and algorithms,” ACM Transactions on Database Systems, vol. 8, no. 4, pp. 465–483, 1983.

[11] Tile38. [Online]. Available: https://tile38.com/. [Accessed: 23-May-2021].

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10, 2010..

[13] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, “REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices,” Lecture Notes in Computer Science, pp. 21–39, 2016.

[14] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, “Overview of multidatabase transaction management,” CASCON First Decade High Impact Papers on - CASCON '10, 2010.

[15] F. Junqueira, B. Reed, and M. Yabandeh, “Lock-free transactional support for large-scale storage systems,” 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W), 2011.

[16] G. Zhang, K. Ren, J.-S. Ahn, and S. Ben-Romdhane, “GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases,” 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019. Conference on Data Engineering (ICDE) (pp. 2024-2027). IEEE.

[17] Levandoski, J. Justin, D. Lomet, M. Mokbel and K. Zhao. “Deuteronomy: Transaction Support for Cloud Data.” CIDR (2011).

[18] V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi, “Typhon: Consistency Semantics for Multi-Representation Data Processing,” 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017.

[19] A. Dey, A. Fekete, and U. Rohm, “Scalable distributed transactions across heterogeneous stores,” 2015 IEEE 31st International Conference on Data Engineering, 2015.