electronics

MDPI

Article

Horizontally Scalable Implementation of a Distributed DBMS Delivering Causal Consistency via the Actor Model

Carl Camilleri, Joseph G. Vella * and Vitezslav Nezval

Computer Information Systems, Faculty of ICT, University of Malta, MSD 2080 Msida, Malta; carl.camilleri.04@um.edu.mt (C.C.); vitezslav.nezval@um.edu.mt (V.N.)
* Correspondence: joseph.g.vella@um.edu.mt

Abstract: Causal Consistency has been proven to be the strongest type of consistency that can be achieved in a fault-tolerant, distributed system. This paper describes an implementation of D-Thespis, which is an approach that employs the actor mathematical model of concurrent computation to establish a distributed middleware that enforces causal consistency on a widely used relational database management system (RDBMS). D-Thespis prioritises developer experience by encapsulating the intricacies of causal consistency behind an interface that is accessible over standard REST protocol. Here, we discuss several novel results. Firstly, we define a method that builds a causally consistent DBMS supporting elastic horizontal scalability. Secondly, we deliver a cloud-native implementation of the middleware and provide results and insights on 6804 benchmark configurations executed on our implementation while running on a public cloud infrastructure across several data centres. The evaluation concerns transaction processing performance, an evaluation of our implementation's update visibility latency, and a memory profiling exercise. The results of our evaluation show that under a transactional workload, a single-node installation of our implementation of D-Thespis is 1.5 times faster than a relational DBMS running serialisable transaction processing, while the performance of the middleware can improve by more than three times when scaled horizontally within the same data centre. Our study of the memory profile of the D-Thespis implementation shows that the system distributes its memory requirements evenly across all the available machines, as it is scaled horizontally. Finally, we also illustrate how our middleware propagates data changes across geographically-distributed infrastructures in a timely manner: our tests show that most of the effects of data change operations in one data centre are available in a remote data centre within less than 300 ms over and above the network round trip latency between the two data centres.

Keywords: causal consistency; elastic horizontal scalability; middleware; distributed DBMS; actor model; fault tolerance

Citation: Camilleri, C.; Vella, J.G.; Nezval, V. Horizontally Scalable Implementation of a Distributed DBMS Delivering Causal Consistency via the Actor Model. *Electronics* **2024**, *13*, 3367. https://doi.org/10.3390/electronics13173367

Academic Editors: Massimo Canonico and Marco Guazzone

Received: 30 June 2024
Revised: 15 July 2024
Accepted: 19 August 2024
Published: 24 August 2024

1. Introduction

The CAP theorem [1,2] demonstrates the impossibility of simultaneously achieving both partition tolerance and availability in distributed database systems that implement strong consistency (SC) [3]. SC represents the most robust form of data consistency provided by conventional distributed database management systems (DDBMSs).

The proliferation of distributed data centres (DCs) has led to broad integration of database (DB) configurations where availability and partition tolerance are prioritised over strong data consistency. This shift is driven by the need for scalability and high availability (HA) in business-grade and mission-critical software. In this context, popular database management systems (DBMSs) frequently implement eventual consistency (EC) [4]. Eventual consistency is a weak consistency model that ensures all sites (i.e., distributed partitions) of a database will, at some point, converge and store the same dataset, provided no new WRITE operations occur.

Eventual consistency is comparatively straightforward to implement in a DBMS and avoids the performance constraints associated with distributed consistency enforcement algorithms, such as Paxos [5]. These algorithms aim to achieve durability and consensus on data updates in unreliable distributed environments. However, eventual consistency relinquishes consensus and shifts the responsibilities for data consistency and safety to the application layer. This shift introduces a novel array of challenges in software development [6].

Causal Consistency (CC) [7] occupies a middle ground among consistency models in distributed systems, surpassing EC in strength while not reaching the rigidity of SC. The research has established CC as the most robust consistency achievable in distributed systems tolerant to faults [8]. The essence of CC lies in its guarantee that data element versions are only visible when all contributing operations are observable [9] by clients (i.e., data consumers). Despite its theoretical suitability for enterprise-scale applications [10], CC's widespread adoption faces several obstacles:

1. Data modelling constraints: CC DBMSs typically restrict themselves to key-value or abstract data type paradigms, limiting expressiveness.
2. Developer-unfriendly interfaces: The current CC DBMS implementations often require bespoke application design tailored to their specific semantics, failing to shield developers from distributed computing complexities [11].
3. Ecosystem incompatibility: The proprietary data formats employed by CC DBs create isolation, impeding interoperability with external data consumers.

These factors collectively contribute to the limited industrial uptake of CC, despite its theoretical advantages.

Our prior research introduced Thespis [12], a middleware solution implementing CC while utilising a relational database management system (RDBMS) for data storage. Thespis prioritises developer experience (DX) by implementing an intuitive interface via REST APIs, effectively masking the intricacies of CC. Building upon this foundation, we subsequently proposed D-Thespis [13], an enhanced iteration addressing two key aspects. First, D-Thespis achieves "elastic horizontal scalability", enabling multi-machine deployment within each DC while preserving autonomous DC configuration. This approach diverges from conventional "horizontal scalability" methods, such as partitioning, which typically mandate predetermined, uniform DBMS configurations across DCs. Second, D-Thespis refines the Thespis protocol by minimising false positives in causal dependency identification. This optimisation yields improved update visibility latency and accelerates the propagation of WRITE operation effects to clients in remote DCs.

This paper presents an implementation of the D-Thespis middleware and also describes several novel results. Firstly, for every aspect of our implementation of the D-Thespis design [13], we describe the technical choices made to achieve a cloud-native implementation of D-Thespis. Subsequently, we perform a detailed empirical analysis of our implementation, comprising a total of six thousand and forty-eight (6048) test runs against the middleware running on a public cloud infrastructure across several data centres. We show that D-Thespis can be deployed in a public cloud data centre, running on five machines, to sustain a throughput of up to 400 thousand requests per second, outperforming an RDBMS running on the same infrastructure by more than 300%. Our study of the memory profile of the D-Thespis implementation shows that the system distributes its memory requirements evenly across all the available machines, as it is scaled horizontally.

Other empirical analysis efforts focused on measuring the update visibility latency of our implementation. D-Thespis was deployed on two data centres. These data centres were interconnected such that the network round trip latency between them was measured to be 135 ms. In this configuration, the effects of the majority of data change operations in one data centre were available in the other data centre within less than 300 ms over and above the network round trip latency between the two data centres.

This paper is structured to provide a comprehensive exploration of our research. Section 2 establishes the essential terminology. Section 3 presents a concise review of the

relevant literature and outlines the fundamental architecture of Thespis. Specifically in Section 3.3, we present a summary of D-Thespis, drawing from our previous research [13]. The implementation details of D-Thespis are examined in Section 4, alongside a discussion of the requisite features for a middleware that delivers causal consistency while supporting elastic horizontal scalability. Section 5 offers an in-depth analysis of our performance evaluations. We then critically examine our findings in Section 6 before presenting our conclusions in Section 7.

2. Definitions

This section introduces the key terminology used throughout the paper.

Replica: A complete copy of a database, with each instance containing the full dataset.

Data Centre/Site: A physical location housing a database replica.

Distributed Database (DDB): A database system distributed across multiple sites.

Database Operation: An action executed by an application via the DBMS's API.

Operation Latency: The time interval between a client's request submission and receipt of the operation's result, typically measured in milliseconds.

System Throughput: The number of operations processed per unit time, often expressed as requests per second, for a given system setup (i.e., a given system implementation, configuration and infrastructure).

Data Freshness: The time required for clients connected to a remote DC to access the results of state-changing operations (e.g., WRITEs) performed in the local DC. Often, data freshness is compromised to enhance performance, as measured by system throughput and the latency of the DBMS operations.

Causal Consistency: A DDB is considered causally consistent when all causally related operations are observed in the same order across all sites [7]. Operations a and b are potentially causally related, denoted as $a \rightarrow b$, if they meet at least one of the following criteria [14]:

1. Thread of Execution: Within a process P, if P performs a before b, then $a \rightarrow b$.
2. Reads From: If operation b reads the result of a WRITE operation a, then $a \rightarrow b$.
3. Transitivity: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Causally consistent DDBs do not enforce a deterministic order for concurrent operations across their distributed nodes. Operations are classified as concurrent when they exhibit no causal dependency, implying independence in terms of data access patterns [7]. Two operations a and b are concurrent if neither precedes the other ($a \nrightarrow b$ and $b \nrightarrow a$). This characteristic permits the arbitrary replication of concurrent operations within the DDB without compromising causal consistency.

Conflict Handling: To ensure system availability even within unreliable environments (i.e., to achieve HA), as well as to perform with optimal operation latency, distributed databases must accept WRITE operations at any node without incurring the overheads of coordination and synchronisation among the rest of the nodes, particularly within the critical path [9]. Data inconsistency arises from conflicting updates across different replicas. A conflict occurs when two replicas concurrently modify the same data element. Two operations on a shared data element are considered conflicting if they produce divergent values and are causally independent (i.e., concurrent) [7]. The following is a formal definition, in the context of DDBs, of operations that are both "concurrent" as well as "conflicting":

- Let Θ_1 and Θ_2 define a WRITE operation on a data item identified by key k_x, in DC_1 and DC_2, respectively.
- Let $put(k, v)$ represent an operation that sets the value v to the data element identified by key k.
- Let $\Theta_1 = put(k_x, v_1)$.
- Let $\Theta_2 = put(k_x, v_2)$.

$\therefore \Theta_1$ and Θ_2 are concurrent and conflicting operations.

Several methods have been proposed for detecting and resolving conflicts in distributed systems [15–17]. One widely used conflict resolution strategy is the last-writer wins (LWW) approach, where the most recent update takes precedence in the event of a conflict. A database that ensures causal consistency (CC) along with conflict detection and resolution, leading to eventual convergence, is considered to provide causal+ consistency (CC+) [18].

3. Literature Review

3.1. Causally Consistent Databases

We give a brief overview of a few causally consistent implementations that are relevant to this paper.

In the COPS [18] system, the application clients are positioned within a number of servers storing a complete image of a database. A WRITE is enqueued and sent to the associate data centres, where it is deposited if all of its dependents are also found. A client-side library is responsible for tracking the dependencies of a WRITE by monitoring a context identifier. The WRITE dependencies in a given context are determined by the most recent state of all the keys that have been previously accessed, ensuring causality. Conflicts are resolved utilising a last-writer wins (LWW) technique.

COPS-GT [18] enhances the functionality of COPS by including the ability to do read-only transactions. Users can request the values of several keys instead of just one, and the DDBMS provides a snapshot of the requested keys that is consistent with the causal order of events. The COPS-GT system utilises a sequentially consistent key-value pair store, similar to the COPS system mentioned earlier. However, it differs in terms of the client library, the database, and the semantics of the READs and WRITEs. In contrast to COPS, the keys in this system are associated with a collection of versions rather than a single value. Every version is associated with a value and a collection of dependencies, represented as pairs of key and version values. This versioning allows for performing READ and WRITE operations inside a transactional context.

Bolt-on [9] refers to a specialised middleware that is built on top of Cassandra, a commercially accessible EC DB that utilises a columnar data architecture and manages object replication. Bolt-on utilises explicit causality, transferring the responsibility of tracking dependencies to the process spawning the operation. The objects are labelled with a collection of data structures, each including an identifier of the process in which they are generated, as well as an identifier that increments progressively. The WRITEs dependencies are determined by the key versions that were read in order to generate that operation. Every process possesses a collection of interests, which are the specific keys it requires to access. A resolver process is responsible for maintaining a view of these keys that is consistent with their causal relationships. This is achieved by retrieving the most recent object versions and their associated dependencies.

GentleRain [19] offers consistency and control over a distributed database that uses a key-value structure, supports many versions of data, and is both sharded and replicated. The propagation of WRITEs between DCs relies on a proprietary replication technique. The efficiency of dependency tracking lies in the small amount of meta-data stored during a WRITE operation, which includes simply a date and a server identifier. Only the data versions produced locally or versions that are sent across all data centres can be accessed by any READ operation. This ensures causality by guaranteeing that all data centres include the elements that have contributed to the formation of a version, known as its dependencies.

Wren [20] adopts a comparable strategy to Gentlerain [19] but employs hybrid logical clocks [21] to accurately date occurrences. In addition, Wren incorporates transactional concurrency control, enabling clients to execute read transactions and execute multiple WRITEs atomically.

3.2. Thespis

Thespis [12] serves as a middleware that enforces causal consistency in a database while also providing mechanisms to detect and resolve data conflicts. Thereby, Thespis fulfils the requirements of CC+. The data are maintained in a database managed by a DBMS. This DBMS provides strong consistency and a robust data model with advanced querying and reporting features, but it lacks effective horizontal scalability. Thespis, in fact, uses an RDBMS to store data. RDBMSs are commonly deployed in production enterprise-grade systems and are sought for their rich data model and efficient capabilities for the management and analysis of data sets. Therefore, while not a strict requirement for Thespis middleware, its design presumes that the primary data management system is an RDBMS and that its clients are applications that manage "objects", predominantly instances of business domain models.

Thespis addresses several goals: it facilitates the integration of CC without necessitating significant re-engineering and/or refactoring of application code, preserves data using a structure such that they remain accessible to other systems (e.g., business intelligence systems), and ensures that the performance costs of causal consistency guarantees are justified by the benefits of using a DDBMS. These goals are achieved through the integration of various methodologies, including the following:

1. The actor mathematical model for concurrent computation [22], which defines computation as a hierarchical society of "experts". These communicate by passing messages asynchronously. An actor comprises (a) a mailbox to queue incoming messages; (b) the actor's behaviour, which is the function that executes in response to the received messages; and (c) the actor's state, which is the data that the actor holds at a given time. Actors process messages one at a time and operate within the scope of actor Systems [23], allowing for hierarchical formations.
2. Command query responsibility segregation (CQRS) [24], a software architectural pattern that implements the principle of command query separation (CQS) [25] to manage distinct data models for operations that read and write data.
3. Event Sourcing (ES) [26], another software architectural pattern which defines that data modifications are recorded in terms of sequences of events. These events can be stored in an append-only event log and subsequently applied sequentially. This approach allows a system using ES to reconstruct its state at any point in time based on the recorded events.

Figure 1 presents the Thespis middleware, which provides an API supporting two actions: READ and WRITE. Both operations utilise the actor model to address the intricacies of concurrency. Firstly, through the guarantees of the actor model, the approach ensures that READ operations occur concurrently. Furthermore, the approach guarantees that WRITE operations on the same object and within the same replica occur in a defined order. The hierarchical structure of the actor systems is also leveraged to maintain a view of the underlying DB, which satisfies the requirements of causal consistency. Respectively, the writer actor and reader actor are concerned with the storage of actor states and the retrieval of business objects from the backing data store (i.e., the RDBMS). The responsibility of the replication actor handles the propagation of actor state changes across replicas. The central middleware actor system comprises a collection of actors that deliver a representation of the DB to the application. Additionally, the actor system employs a "child-per-entity" approach, creating one entity actor per type of business object (e.g., DB table), which supervises dedicated entity instance actors for each business object instance (e.g., a table's row). The entity instance actor maintains a state that consists of two components: the entity instance and the event log. By applying the events in the event log to the entity instance managed by the entity instance actor, the system obtains the most recent version of the entity that does not violate the principles of causal consistency.

WRITE operations are processed at the middleware layer. Given a WRITE operation w_e affected on entity e_1, a new version of the same entity e_2 is generated. By comparing this

new version e_2 to the previous version of the same entity e_1, it is possible to extract a set of events l that describe the transition from e_1 to e_2. The middleware is also responsible to periodically execute the snapshot process. This process modifies the state of the RDBMS to align with the updates represented by the events identified by the middleware, specifically for those events that have been received in all the replicas of the Thespis DDBMS.

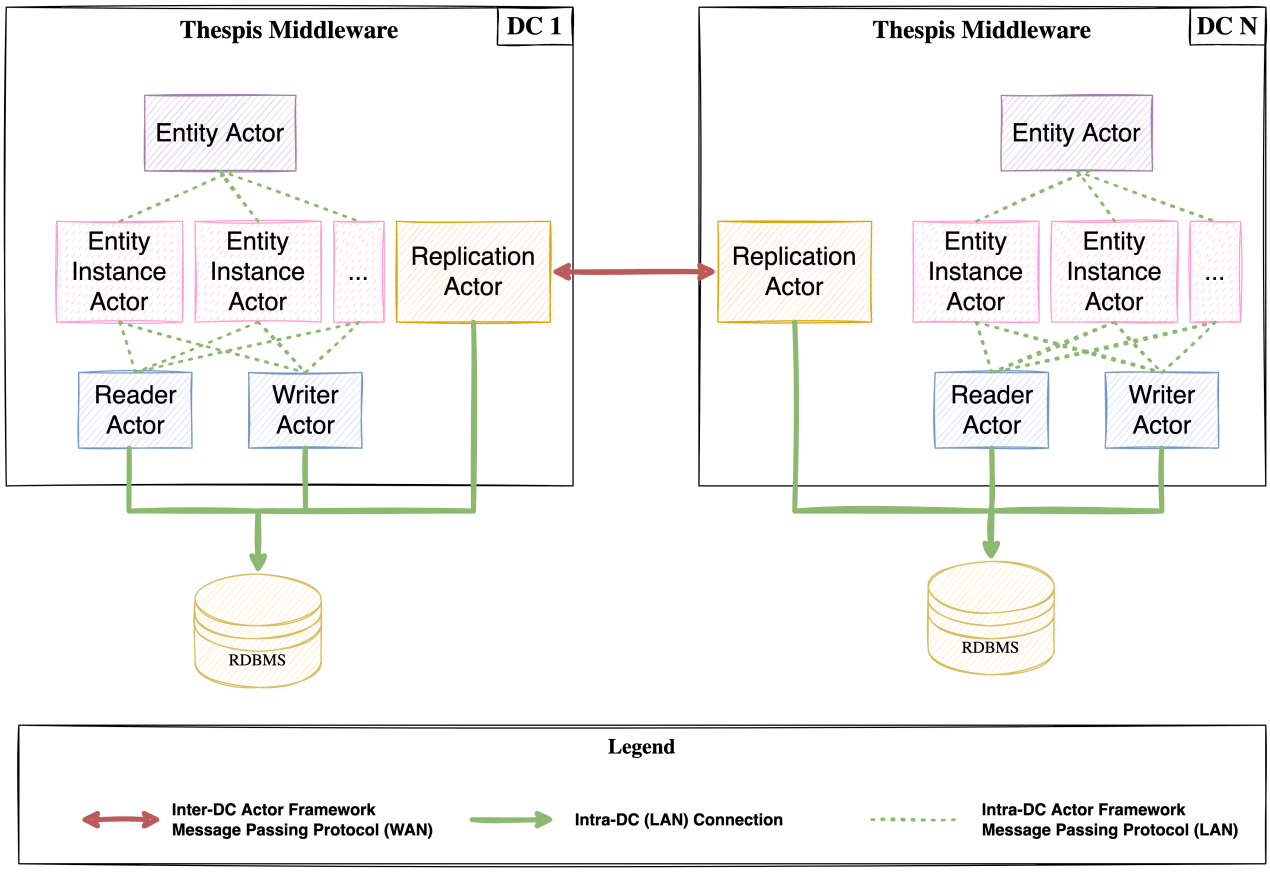

Figure 1. System Model of Thespis: A middleware delivering causal consistency over an RDBMS using the actor model.

Lastly, the system integrates a replication protocol, also designed based on the mechanics of the actor model. The replication protocol comprises two algorithms: the originating server algorithm and the remote server algorithm. Respectively, these algorithms are executed within the data centre, where a new event is generated (i.e., the DC that processes a WRITE operation), and within the data centre, which asynchronously receives the effects of the same WRITE operation. The replication protocol enforces causal consistency through the use of the stable version vector (SVV). The SVV is an M-dimensional array, with M constituting the quantity of Thespis replicas (i.e., the number of DCs). Every vector element SVV_{DC} is the latest timestamp of an event originating from the corresponding peer DC. More precisely, the entry $SVV_{DC_N}[M]$ of vector SVV_{DC} holds the most recent timestamp observed from DC M within DC N.

The findings from our assessment of an implementation of Thespis [12] demonstrate the effectiveness of this approach in attaining causal+ consistency, availability and partition tolerance, while delivering superior performance (in terms of throughput) compared to an RDBMS. Moreover, in alignment with the PACELC theorem [27], Thespis ensures causal consistency and minimises the latency of database operations in normal operating conditions, all while accommodating failures of nodes participating in the database cluster, as well as the partition of the network interconnecting the database replicas.

In the later research, we enhance Thespis to accommodate read-only transactions with ThespisTRX [28]. This extension introduces new API endpoints in Thespis to allow the retrieval of multiple entities from the backing data store (i.e., the RDBMS) over several operations while still ensuring causal consistency guarantees. We demonstrate that ThespisTRX addresses issues like time-to-check-time-to-use (TOCTOU) race conditions. Our empirical analysis reveals that the read latency in ThespisTRX is comparable to that in Thespis, thus highlighting the efficiency of our approach.

In yet another research paper [29], we tackled the issue of data integrity invariants in a distributed DBMS such as Thespis. We introduced ThespisDIIP [29], another extension to Thespis that can preserve integrity invariants for data values that adhere to the linear arithmetic inequality constraints [30] in a distributed environment. Our evaluation of ThespisDIIP [29] indicates that ThespisDIIP achieves asynchronous operation latencies that are low enough to avoid delays in the critical path of representative enterprise-grade applications.

3.3. The Design of D-Thespis

The research presented in [13] introduces a novel approach that builds upon the foundations of Thespis, enhancing it in three principal domains. The first advancement pertains to the implementation of "elastic horizontal scalability", enabling the deployment of the middleware across multiple machines within each data centre, whilst simultaneously allowing for the autonomous configuration of individual data centres. This contrasts with conventional "horizontal scalability" methods, which often rely on less flexible techniques such as partitioning, necessitating a priori definition and standardisation of DBMS configurations (e.g., partition numbers) across all the database replicas. Secondly, the proposed system aims to refine the Thespis protocol by mitigating the incidence of false positives in causal dependency identification. This improvement ameliorates update visibility latency and expedites the propagation of WRITE operation effects to clients (i.e., data consumers) in remote data centres. The third area of improvement focuses on improving performance by allowing the safe concurrency of READ and WRITE operations. The original Thespis protocol, by modelling all operations as actor calls, processes operations on any given entity within a DC sequentially. While this approach effectively manages the complexities of parallelism and concurrency for stateful entities, it inadvertently introduces unnecessary bottlenecks in handling READ and WRITE operations on individual data entities. Drawing inspiration from the previous research [31,32], D-Thespis implements transaction concurrency mechanisms without compromising correctness.

In our previous work [13] we also outline the requirements of D-Thespis, namely,

1. The data centre clock, which serves as the source of truth that provides physical timestamps within its DC;
2. The data cluster clock, which is the source of truth of the SVV within its DC;
3. A scalable event log that provides key-based multi-record lookups, key-based event queueing, durability, key-level atomicity, key-level isolation and horizontal scalability;
4. A version cache for operation concurrency, which is a cache layer installed in every DC. This layer preserves the latest causally consistent version of any data entity. The version stored in this cache layer is the one inferred by a READ operation within the same DC. This cache layer is also mutualised, such that all the nodes within a DC refer to a single cache layer. This promotes a higher cache hit ratio, i.e., the ratio of calls to the cache layer requesting data entities that are stored within the cache (cache hits) versus the total number of calls to the cache layer, including those that request data entities that are not stored in the cache (cache misses). The D-Thespis READ and WRITE operations are also tuned with cache maintenance functionality.

The D-Thespis architecture given in [13] is illustrated in Figure 2, comprising several components that depend on each other through library references, calls over the LAN (i.e., within the same DC), or calls over the internet (i.e., across different DCs). The components that are used through library referencing expose generic interfaces, such that their implementation specifics are hidden from any dependent layers. This design choice allows

multiple implementations of each library to co-exist and be used through a configuration or convention-driven setup, effectively achieving low coupling across the different layers.

The rest client API provides a lightweight facade that exposes REST APIs, enabling clients to perform read and write operations on the dataset stored in the RDBMS. This layer transforms client HTTP requests into appropriate internal data structures before forwarding them to the middleware engine.

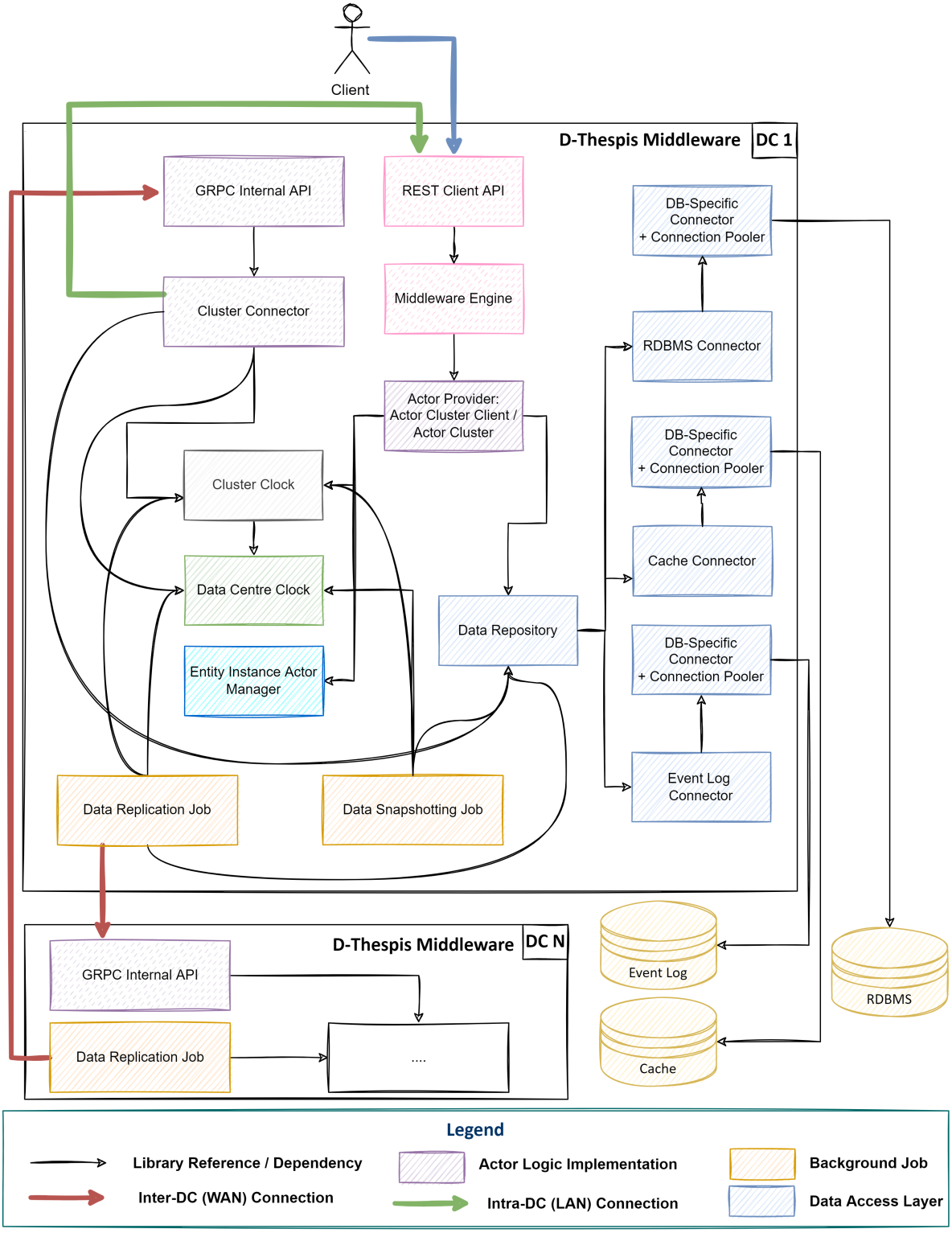

Figure 2. Conceptual Architecture of D-Thespis.

The GRPC internal API is implemented for enhanced efficiency in inter-component communication. This layer utilises the GRPC (https://grpc.io/ accessed on 24 March 2024) protocol and is exclusively used for internal network communication between various system components.

The middleware engine interacts with the actor provider module. While the middleware engine itself remains agnostic to the specifics of how the actor model is implemented, it serves as an intermediary layer, delegating requests to the actor provider module. An exception to this occurs when the actor provider module supports concurrent READ operations. In these scenarios, the middleware engine seeks to process READ requests autonomously by accessing a version of the data record requested by the client from the distributed cache. This version is guaranteed to be the most up-to-date and causally consistent version. Should this attempt fail (i.e., no cache entry exists for the requested data entity), the READ operation is then forwarded to the actor provider component.

The actor provider module encapsulates the logic necessary for execution within the context of the actor model. It consists of two components: the actor cluster client and the actor cluster. The requests received from the middleware engine are encapsulated in appropriate data structures and transmitted as messages to the actor cluster via the actor cluster client.

The actor cluster implements the core protocol of D-Thespis, using the specific semantics of its actor model framework. It also interfaces with the entity instance actor manager for algorithms that are fundamental to the system but not influenced by the particularities of the Actor model framework. This entails, for instance, the algorithm for implementing a new event to a particular version of a data entity.

The data repository is effectively a data access layer and serves as the primary interface for all other components to interact with the data storage systems. It exposes an interface through which other components can read data from the RDBMS, access the event log and interact with the distributed version cache. Access to the various data repositories is abstracted further within connectors that are specific to each database system, ensuring that the data access logic remains independent from the underlying database system.

The data centre clock is a module that enables other components to retrieve a physical clock timestamp. Various approaches can be employed to implement this module. For instance, in the server data centre clock approach, the module provides the wall clock time by interrogating the physical clock of the machine on which it is installed. Alternatively, by using the middleware data centre clock strategy, the module acts as a thin client: it retrieves a physical timestamp by interrogating an internal API offered by a component running on a designated server within the data centre.

The cluster clock is a module that serves as the entry point for reading and updating the stable version vector (SVV). Like the data centre clock, when using the server cluster clock approach, the cluster clock component manages a version of the SVV in its process memory. In contrast, when using the middleware cluster clock technique, the cluster clock interrogates a singleton service hosted on a designated machine within the local DC to retrieve or change the values of the SVV.

D-Thespis also incorporates a data replication job that runs at regular intervals. This component queries the event log to retrieve events originating from the local data centre that have yet to be propagated to other data centres in the D-Thespis cluster. Once identified, these events are aggregated into batches and transmitted to peer data centres via remote procedure calls (RPCs). Each data centre participating in the D-Thespis cluster exposes a dedicated endpoint to facilitate this inter-DC communication.

Lastly the data snapshotting job is implemented for two reasons: to optimise storage and improve system efficiency and to ensure that the underlying RDBMS is updated with the data modifications generated by WRITE operations handled by the D-Thespis middleware. Like the data replication job, this process also runs periodically, analysing the event log to identify specific events that represent stable versions of data entities, i.e., those versions for which no additional dependencies are anticipated. These events are then applied

sequentially to update the corresponding data entity's representation in the primary RDBMS. Following this consolidation, the data snapshotting job then prunes the event log, by removing the processed event data from the event log.

The system supports various deployment configurations. Figure 3 demonstrates a potential arrangement of the architecture deployed in a configuration that spans across ten nodes (servers) within a single data centre. This constitutes just one particular arrangement: other configurations can be scaled horizontally across a different number of nodes. In this type of setup, the user-facing components, specifically the REST client API, the middleware engine and the actor cluster client, function independently from the actor cluster and can therefore be served by a segregated part of the infrastructure. Requests originating from clients connected to the middleware are distributed among these nodes using conventional mechanisms suitable to load balance HTTP requests. The actor cluster can scale horizontally, utilising the distribution policies and protocols implemented by the actor model framework to allocate actors across several machines while still ensuring that only one actor representing a data item exists at any given moment. The actor cluster clients engage in the particular protocol defined by the actor model framework to locate servers hosting actor cluster nodes, which can then accept requests to the actor provider layer. In this system setup, it is presumed that all servers within the same DC maintain highly synchronised clocks. Consequently, the data centre clock layer implements the server data centre clock approach, operating independently on each node. Nonetheless, all the machines hosted in the same data centre must interrogate and manage a single instance of the SVV. Thus, each server runs a middleware cluster clock client that interrogates the same server cluster clock service via GRPC (i.e., internal) API calls. In the configuration illustrated in Figure 3, Server 10 serves as the authority for the SVV. Lastly, within this setup, the data replication job and the data snapshotting job cannot execute on more than one machine and therefore cannot be scaled horizontally: it is not possible to run concurrent instances of each of these background processes within any particular DC. Nevertheless, these background processes are independent and largely autonomous from the other system modules, allowing them to run on dedicated servers if necessary.

The D-Thespis configuration shown in Figure 4 is very different from the one illustrated in Figure 3, but it is also valid. This setup illustrates that all D-Thespis components can be hosted by a single server. In such a setup, we can use the server version of both the data centre clock and the cluster clock, which are more efficient.

The specifications of D-Thespis [13] also include the definition of the D-Thespis protocol, in the form of a series of algorithms. These include algorithms for the server cluster clock, the stable version vector, the READ and WRITE operations in D-Thespis, event ordering and the snapshot and replication operations. We omit the replicating details of these algorithms for brevity. A correctness evaluation for D-Thespis is also given [13], where we show how the definition of the D-Thespis protocol delivers monotonic read consistency, read-your-writes guarantees, monotonic write operations, write follow reads and causal consistency.

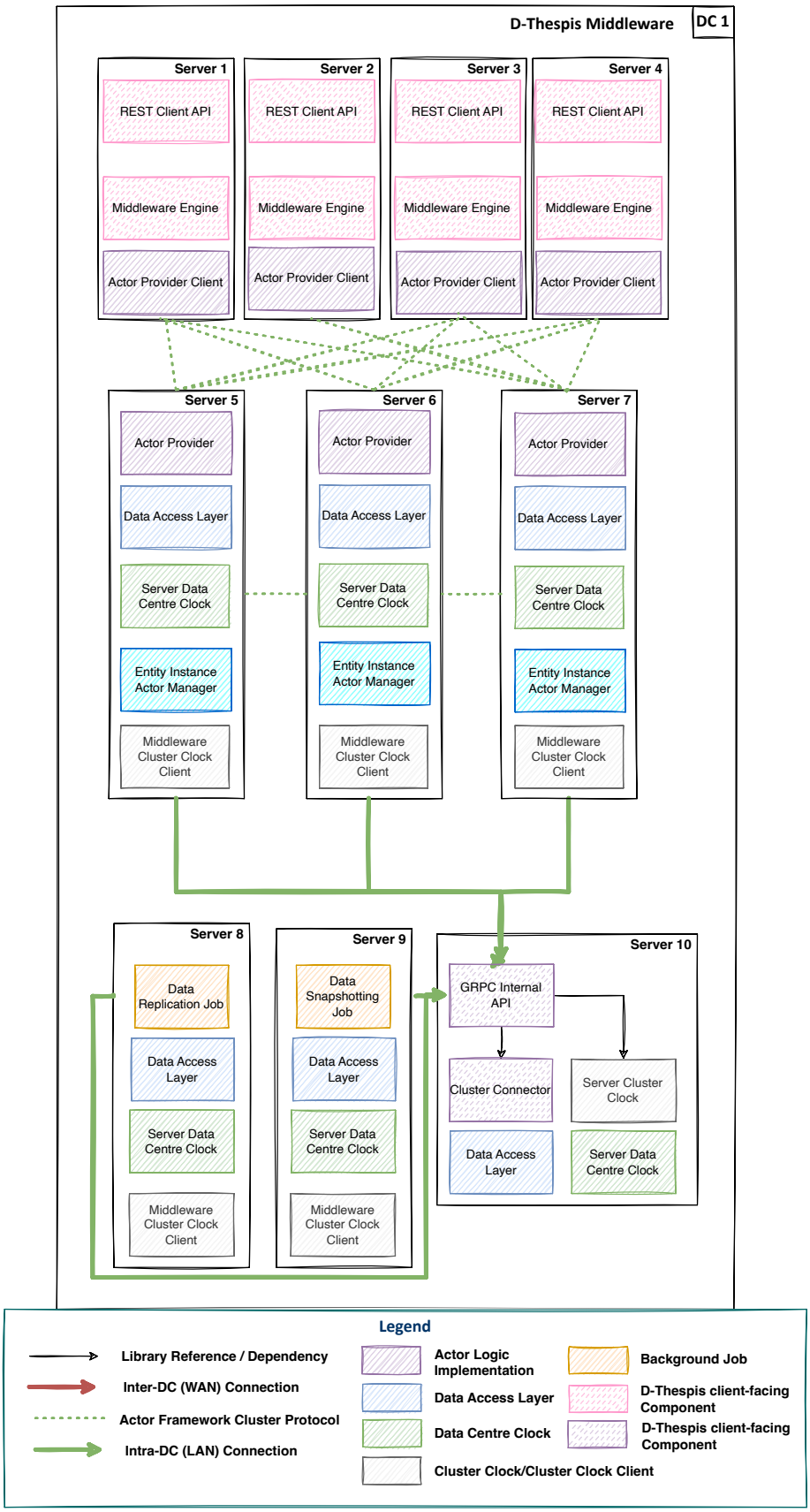

Figure 3. Conceptual Architecture of D-Thespis: Intra-Data Centre Horizontal Scalability.

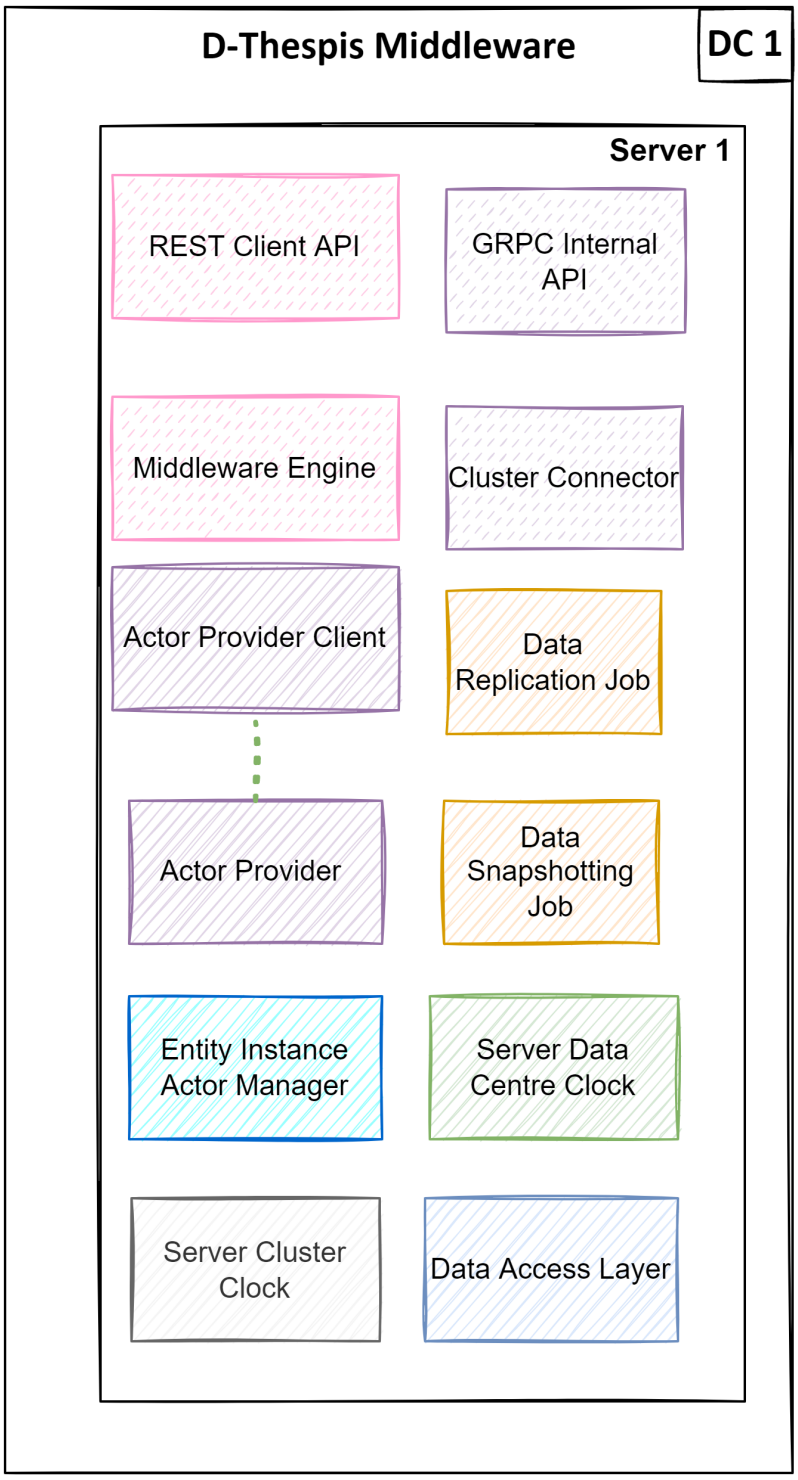

Figure 4. Conceptual Architecture of D-Thespis: Single-node configuration within a data centre.

4. D-Thespis Implementation

In this paper, we focus on an implementation of D-Thespis, along the system architecture design discussed in Section 3.3. Our implementation uses the Microsoft .NET 6.0 framework, in line with conclusions drawn from the experiments in our earlier work [33]. The implementation uses the C# language, comprising 7000 lines of code and following the concepts of a "Clean Architecture" (https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html accessed on 20 February 2024) as shown in Figure 5.

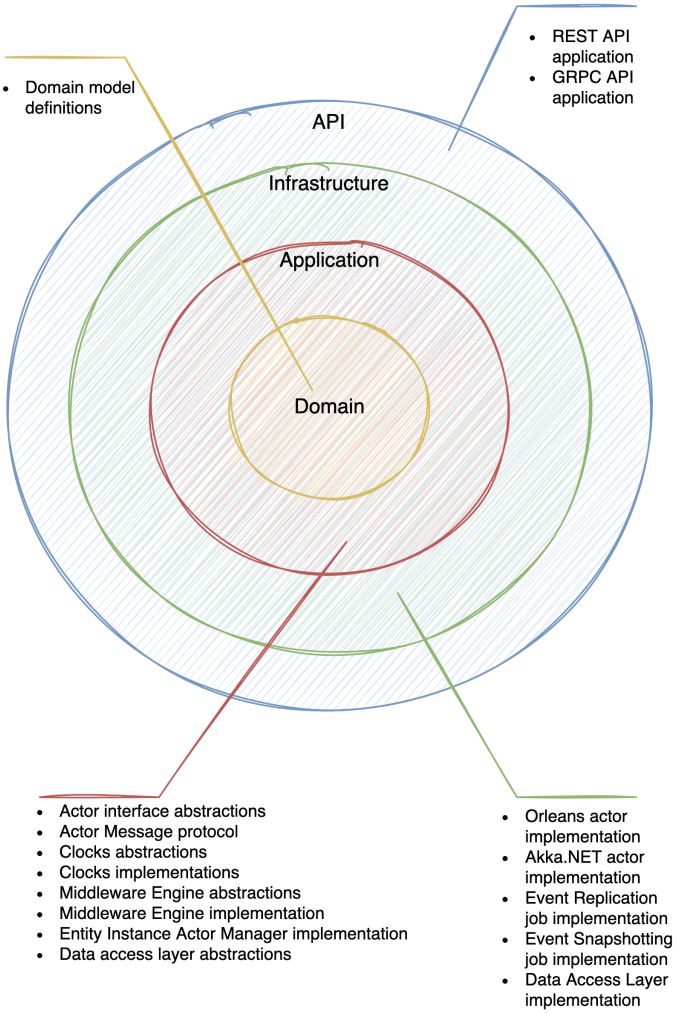

Figure 5. D-Thespis architecture based on concepts of the "Clean Architecture" https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (accessed on 20 February 2024).

Intra-layer network communication is based on GRPC https://grpc.io/ (accessed on 24 March 2024), using Google protocol buffers (Protobuf) https://developers.google.com/protocol-buffers/ (accessed on 24 March 2024) for message marshalling. The combination of GRPC and Protobuf is a modern industry standard for efficient communication [34], and the technology is also uplifted for similar purposes in the literature [35,36].

Two implementations of the actor provider layer are incorporated, one using the Akka.NET toolkit and the other the Microsoft Orleans framework. The Akka.NET actor provider implementation is tuned for single-node configuration, while the version based on Microsoft Orleans is automatically used when D-Thespis is configured in a horizontally scalable setup.

Redis https://redis.io/ (accessed on 24 March 2024) is incorporated for several purposes. Firstly, a horizontally scalable setup of D-Thespis that uses the Microsoft Orleans actor framework requires a grain directory https://docs.microsoft.com/en-us/dotnet/orleans/host/grain-directory (accessed on 24 March 2024). Briefly, the purpose of the grain directory is to track the physical location of grains, and keep track of the available silos. Microsoft Orleans supports Redis as one of its implementations of the grain directory.

Secondly, it is also uplifted for the purposes of the distributed cache layer as it answers all the requirements of a version cache for operation concurrency [13]. Specifically,

1. Key-based lookups are supported via the `GET` command, which runs efficiently in $O(1)$ complexity;

2. Key-based version storage is supported via the `SET` command, which also executes in $O(1)$ complexity;
3. Key-level atomicity and isolation are guaranteed since Redis executes all operations on a single thread [37]. Therefore, all operations are atomic and execute sequentially;
4. Horizontal scalability is supported by Redis in different configurations (e.g., master–slave and Redis cluster) and can also be built in custom logic executing against stand-alone Redis databases.

Furthermore, the applicability of Redis as a distributed cache layer is widely exploited both in the industry as well as in other works in the literature [38]. Specifically for horizontal scalability, the D-Thespis implementation does not rely on Redis out-of-the-box support but rather can connect to multiple stand-alone instances of Redis and distribute keys among them using a custom implementation of a key distribution algorithm, heavily inspired by the server selection strategy in the Redis driver from StackExchange https://github.com/StackExchange/StackExchange.Redis/ (accessed on 24 March 2024).

Lastly, Redis is also used as the event log storage layer, which is represented using one list data structure per data entity, with the list items being the events, as shown in Figure 6. In Redis, the list data structure can store up to $2^{32} - 1$ items, which are sorted in order of insertion.

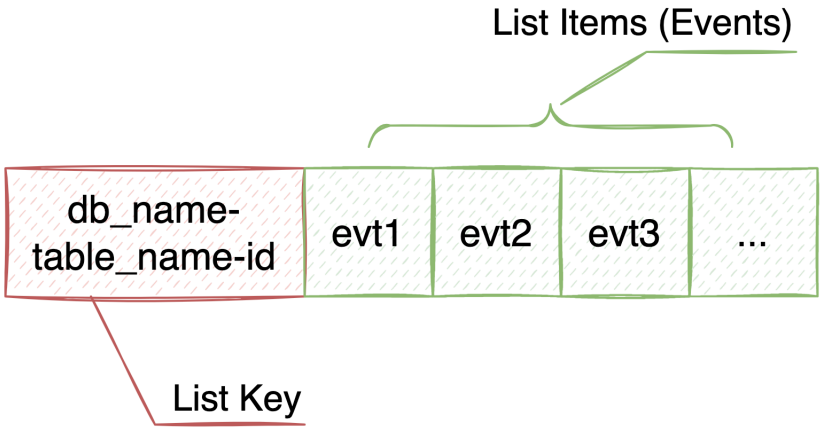

Figure 6. Event log storage using one Redis list per data entity in D-Thespis.

With this approach, Redis meets the needs of a scalable event log [13], specifically,

1. Key-based multi-record lookups are supported via the `LRANGE` command. This operation takes two positional values as arguments and returns the elements within the list that are stored at the given key and reside within the range of positions defined by the given arguments. The `LRANGE` operation has a computational complexity of $O(N)$, with N being the number of items in the list;
2. Key-based event queuing is supported by the `RPUSH` command, which adds a new item to the end of the list corresponding to the event log of a particular data entity. `RPUSH` for a single item is an operation with $O(1)$ complexity;
3. Durability is supported given that two specific configurations are set for Redis, namely, `'appendonly=yes'` and `'appendfsync=always'`;
4. Key-level atomicity and isolation are guaranteed due to the single-threaded nature of Redis;
5. Horizontal scalability can be supported by Redis as discussed for the cache layer in our definition of a version cache for operation concurrency [13].

Furthermore, by not introducing another database for event log storage, the D-Thespis implementation benefits from a reduction in overall complexity.

Despite the use of Redis for several functionalities, it should be noted that the D-Thespis approach does not rely on any feature that is specifically delivered by Redis. For

example, the Redis cluster feature answers the horizontal scalability requirements of the distributed cache layer and even introduces elements of high-availability and promotes resilience. However, the approach of using custom-built key-placement logic decouples the implementation from dependencies on features that are highly specific to Redis. Although this reduces support for the high-availability features of the Redis Cluster module, more importance is given to ensure that the D-Thespis approach can generalise for any other technology that can deliver the requirements of the distributed cache layer. Therefore, other implementations could swap out Redis for other technologies and even opt for a more complex (but perhaps more tuned) setup of different technologies for cache and event log storage.

5. D-Thespis Middleware Performance Evaluation

The implementation of D-Thespis was rigorously benchmarked. A number of systems-under-test (SUTs) were used, namely,

- `DB_1S_1DC`: An application that exposes the same REST API as D-Thespis, executing operations directly on the relational DBMS (`DB`), running on a single server (`1S`) in one data centre (`1DC`).
- `DB_NS_1DC`: The same as `DB_1S_1DC` but with the REST API running on multiple (`N`) servers (`NS`), while the relational DBMS runs on a single server.
- `DT_1S_1DC_O`: D-Thespis (`DT`) deployed on a single server (`1S`) within a single data centre (`1DC`) using the Microsoft Orleans actor framework (`O`), enabling concurrent READ operations over cached data and using the server cluster clock implementation.
- `DT_NS_1DC_O`: D-Thespis (`DT`) deployed on multiple (`N`) servers (`NS`) within a single data centre (`1DC`) using the Microsoft Orleans actor framework (`O`), enabling concurrent READ operations over cached data and using the middleware cluster clock implementation.
- `DT_1S_1DC_A`: D-Thespis (`DT`) deployed on a single server (`1S`) within a single data centre (`1DC`) using the Akka.NET actor toolkit (`A`) with caching disabled and using the server cluster clock implementation.
- `DT_NS_2DC_O`: D-Thespis (`DT`) deployed across two data centres (`2DC`) with each data centre running the same configuration as `DT_NS_1DC_O`.

We designed our benchmarks to specifically answer the following questions:

Q1: What is the impact of the workload profile (i.e., read/write ratio) on the performance of D-Thespis under every configuration?

Q2: Does the dataset size have any bearing on the performance of D-Thespis under every configuration?

Q3: How does the performance of all D-Thespis configurations change in relation to the number of requests submitted?

Q4: How does the performance of D-Thespis under every configuration compare to the `DB_1S_1DC` and `DB_NS_1DC` SUTs, under all the different workload conditions?

Q5: For `DT_NS_1DC_O`, what is the effect of horizontally scaling D-Thespis within a single data centre?

Q6: What is the update visibility latency of D-Thespis? Thus, for `DT_NS_2DC_O`, how long does it take for an event to be visible in a remote data centre?

Q7: How can the memory footprint of D-Thespis be characterised?

5.1. Benchmark Hardware Infrastructure

Benchmarks were executed on a hardware infrastructure deployed in the Google Cloud Platform (GCP), as illustrated in Figure 7. The hardware infrastructure like the one used in our earlier work for actor model framework micro-benchmarks [33], was, in fact, configured with the same hardware specifications as illustrated in Table 1. Briefly, the infrastructure exploits two types of cloud services.

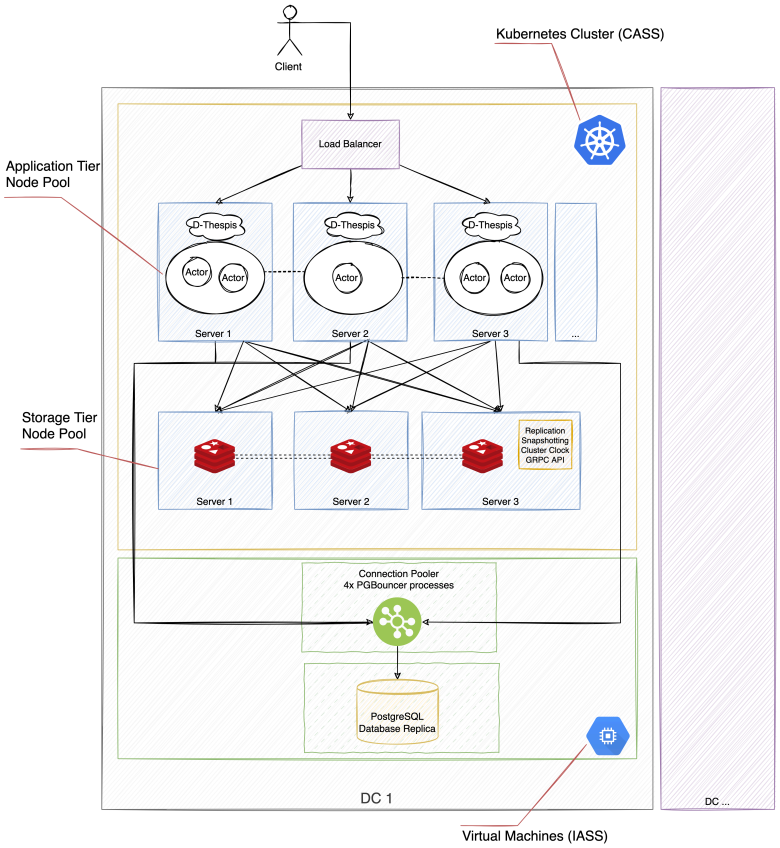

Figure 7. D-Thespis Performance Evaluation Infrastructure.

Table 1. Benchmark Infrastructure Specifications.

Role	Kubernetes Cluster Nodes	Database Server	Load Server	PGBouncer Server
Operating System	Container-Optimized OS	Debian Linux 9.13 (stretch)	Debian Linux 10 (buster)	Debian Linux 11 (bullseye)
CPU	8x Xeon @ 2.80 GHz	8x Xeon @ 2.80 GHz	60x Xeon @ 2.80 GHz	8x Xeon @ 2.80 GHz
RAM	32 GB	32 GB	240 GB	32 GB
Disk	10 GB SSD	100 GB SSD	10 GB Spindle	60 GB SSD

Firstly, a Kubernetes (GKE) cluster was configured with two node pools (i.e., group of servers). The application tier node pool was configured to host most of the D-Thespis components, whilst the storage tier node pool hosts instances of Redis. One server in the storage tier also runs the singleton components of D-Thespis, i.e., the event replication job and the event snapshotting job, and it also exposes the GRPC API for internal communication across the D-Thespis cluster. When running in a horizontally scaled configuration (i.e., `DT_NS_1DC_0` and `DT_NS_2DC_0`), the same server in the storage tier also hosts the cluster clock component and, therefore, serves as the location where the stable version vector resides. In all configurations, it is assumed that the servers in the GKE cluster are equipped with highly synchronised clocks. All components are configured as Kubernetes statefulsets, thus the GKE cluster guarantees the most the number of expected replicas execute at any point in time (e.g., only a maximum of one replica of the singleton components executes at any time). During all tests, three servers are deployed constantly in the storage tier pool. Having three servers running Redis instances was found to be sufficient for the experiments at hand in that the benchmark workloads did not saturate the Redis instances.

Secondly, three compute instances (virtual machines) were deployed in the Infrastructure-as-a-Service (IaaS) mode, as follows:

1. The database server hosts an instance of PostgreSQL;
2. The PGBouncer Server runs four instances of PGBouncer, configured with the option `SO_REUSEPORT` such that all instances share the same TCP port;
3. The load server from which the requests to the D-Thespis API dictated by the benchmark workload for each execution are generated.

5.2. System under Test Configuration

Every SUT consisted of a different configuration, while some components were commonly used by most SUTs. The next few sections describe the configurations in more detail.

5.2.1. Compute Engine Instances

The servers deployed in IaaS mode were used in the same manner for all SUTs. All the benchmarks were executed against the same relational DBMS, using the same connection pooler, while all benchmark workloads originated from the same server.

5.2.2. Redis for Event Log and Cache Storage

Except for `DB_1S_1DC`, all SUTs deployed a statefulset of three replicas, each running a Redis instance having access to a 300 GB SSD disk. These served as the event log storage repository and, for SUTs allowing concurrent reads and with the cache enabled, also as the cache repository. A YAML configuration was used to deploy these instances. One detail of note in this configuration is that Redis is executed with the configuration parameter `'appendfsync=always'`. This is required to guarantee the durability requirements of the event log storage but in practice is overkill for the cache repository, which does not have strict durability requirements. For these benchmarks, the same Redis instances were used for event log storage and cache storage to make efficient use of the available hardware infrastructure.

5.2.3. Redis for Grain Directory

For the SUTs running Microsoft Orleans (i.e., all SUTs except `DB_1S_1DC` and `DT_1S_1DC_A`), an additional statefulset scaled to one replica running a Redis instance was deployed with a 1 GB SSD disk. This instance served as the grain directory for the Microsoft Orleans cluster.

5.2.4. D-Thespis with Microsoft Orleans

A configuration was developed and used to deploy an SUT running Microsoft Orleans. This configuration assimilates to the one given in Figure 3, and comprises three separate statefulset workloads in the GKE cluster:

1. `d-thespis-client` includes the REST API, the middleware engine and the actor provider client components
2. `d-thespis` includes the actor provider and related components
3. `d-thespis-internal` includes the internal API exposed over GRPC (for message packing with Protobuf), the server cluster clock, the data replication job and the event snapshotting job.

Across the different configurations of SUTs running in this configuration, the `d-thespis-client` and `d-thespis` were scaled as necessary, while the `d-thespis--internal` workload was always configured to run a single instance.

5.2.5. D-Thespis with Akka.NET

Another specific configuration for SUT `DT_1S_1DC_A` was developed, like the one given in Figure 4, where all the modules of D-Thespis are hosted on a single node in the GKE cluster.

5.2.6. REST API over Relational DBMS

Finally, one last configuration of the SUT `DB_1S_1DC` was developed. This is the most simple and configures a scalable statefulset workload that exposes the D-Thespis REST API but connects directly to the relational DBMS.

5.3. Benchmarking Tool and Dataset

As in our previous works [12,33,39], the Yahoo! Cloud serving benchmark (YCSB) tool was used for these benchmarks. However, in this case, a customised version of YCSB https://github.com/carlcamilleri/YCSB (accessed on 20 February 2024) was adopted. This version was forked from the official YCSB repository https://github.com/brianfrankcooper/YCSB (accessed on 20 February 2024) and includes two main customisations:

1. A D-Thespis client that has knowledge of the D-Thespis REST API and can send READ and WRITE queries in the correct format. The D-Thespis client can also be configured with a list of URLs such that the workload is distributed across these endpoints, thus, allowing a workload to exercise multiple servers without necessarily relying on load balancing equipment;
2. An extension of the BasicDB module of YCSB to support asynchronous requests and, therefore, allow YCSB clients to make better use of application thread pools where the target database supports asynchronous requests. In the case of the D-Thespis client, the benchmark uses this interface to perform asynchronous HTTP requests.

The benchmarks were executed over the YCSB dataset, with the relational DBMS seeded with the same dataset of three million records described in our earlier work [33]. Out of these, one million records were randomly selected to be considered for benchmark workloads.

5.4. Performance Benchmarks: Variations and Procedure

The first set of benchmarks targeted all the possible configurations across the following dimensions:

1. The type of SUT: D-Thespis with the Microsoft Orleans framework (i.e., `DT_NS_1DC_O`) and the REST API that executes queries directly on the RDBMS (i.e., `DB_NS_1DC`).
2. The number of servers, i.e., N in the SUT type, with N varied for values 1, 2, 3 and 5. For example, for the specific SUT `DB_3S_1DC`, the benchmark workload is handled by three instances of the REST API application, running on three servers in the application tier of the GKE cluster. Conversely, for the specific SUT `DT_3S_1DC_O`, three servers in the application tier of the GKE cluster host the `d-thespis-client` workload, whilst another three servers in the application tier of the GKE cluster host the `d-thespis` workload.
3. The type of workload, with each type performing a different ratio of READ and WRITE operations. Nine workloads were tested: 100% READ/0%WRITE, 99% READ/1% WRITE, 95% READ/5% WRITE, 90% READ/10% WRITE, 85% READ/15% WRITE, 80% READ/20% WRITE, 75% READ/25% WRITE, 70% READ/30% WRITE and 65% READ/35% WRITE. The spectrum of workload ratios selected is consistent with other works that benchmark web-scale transactional databases [40], and includes the ratios WRITE and 65% READ/35% WRITE and WRITE, 90% READ/10% WRITE, which are defined by the popular transactional workloads from the standard TPC-C and TPC-E benchmarks, respectively [41].
4. The data set size over which the workload executes. Four data set sizes were tested: 10,000 records; 100,000 records, 500,000 records; 1,000,000 records.
5. The number of virtual users, i.e., the number of threads that the YCSB benchmarking tool spawns in parallel. Seven variations were tested: 60, 120, 240, 480, 960, 1920 and 3840 virtual users.

Therefore, two thousand and sixteen (2016) configurations were considered, with three benchmark runs executed for each configuration, for a total of six thousand and forty-eight (6048) benchmark runs.

Furthermore, the same approach was used to test the SUT `DT_1S_1DC_A`: the D-Thespis configuration running on a single server and using the Akka.NET toolkit was tested for all possible combinations of type of workload, data set size and number of virtual users, yielding another two hundred and fifty-two (252) configurations and seven hundred and fifty-six (756) benchmark runs.

Thus, a total six thousand, eight hundred and four (6804) configurations were tested. A Powershell script was developed to execute these benchmarks. A high-level summary of the benchmark procedure is then given in Algorithm 1.

Algorithm 1 Benchmark routine executed for every type of SUT

for $clusterSize \leftarrow [1, 2, 3, 5]$ **do**
  Scale the *Application Tier* node pool of the GKE cluster to $clusterSize \times 2$
  for $noOfVUsers \leftarrow [60, 120, 240, 480, 960, 1920, 3840]$ **do**
    for $workloadType \leftarrow [100_0, 99_1, 95_5, \ldots, 65_{35}]$ **do**
      for $dataSetSize \leftarrow [10000, 100000, 500000, 1000000]$ **do**
        Restart the PostgreSQL Database server
        Restart the SUT running in GKE
        for $runNo \leftarrow [1 \ldots 3]$ **do**
          for $warmupNo \leftarrow [1 \ldots 3]$ **do**
            YCSB($workloadType, dataSetSize, 1$) ▷ *Warmup with 1 vUser and 100 reqs*
          end for
          $noOfYCSBInstances \leftarrow CEILING(noOfVUsers \div 60)$
          $ycsbJobs \leftarrow \emptyset$
          for $i \leftarrow [1..noOfYCSBInstances]$ **do**
            ▷ *Spawn a standard YCSB run for 60 s*
            $ycsbJobs \leftarrow$ YCSB($workloadType, dataSetSize, noOfVUsers$)
          end for
          Wait for all $ycsbJobs$ to finish
        end for
      end for
    end for
  end for
end for

5.5. Performance Benchmarks: Tool Calibration

Similar to our previous work [12], we configured the YCSB benchmarking tool to yield workloads that fit the Zipfian distribution, a commonly observed distribution in real-life workloads [42]. We performed these benchmarks using a customised version of the YCSB tool, which incorporates a Zipfian sequence generator. Although we did not modify this module in the customised version of YCSB, we tested it to ensure that the sequence generation was still yielding an output that conforms to the Zipfian distribution, as expected.

We performed a calibration run with the customised version of YCSB. The run duration was set to 60 s, which was deemed sufficiently long to consider a sizeable set of values from the set of possible values, and thus sufficient for this assessment. Running YCSB workloads for 60 s is also a practice adopted in orthogonal works [43,44]. We configured YCSB to spawn one virtual user and to generate values out of a possible sequence of one million values. Each value generated from the Zipfian sequence generator was logged during the run.

This calibration run considered 427,799 values out of the possible million. Figure 8 illustrates the top fifty values generated, ordered in descending order of the number of times that they were generated by the distribution generation. The graph shows that some values appeared much more frequently than others in the output from the distribution generator. For example, value `'850018'` appeared 179,832 times, while value `'100044'` appeared 3722 times. We could therefore assert that the YCSB tool was generating workloads that conformed to a Zipfian distribution.

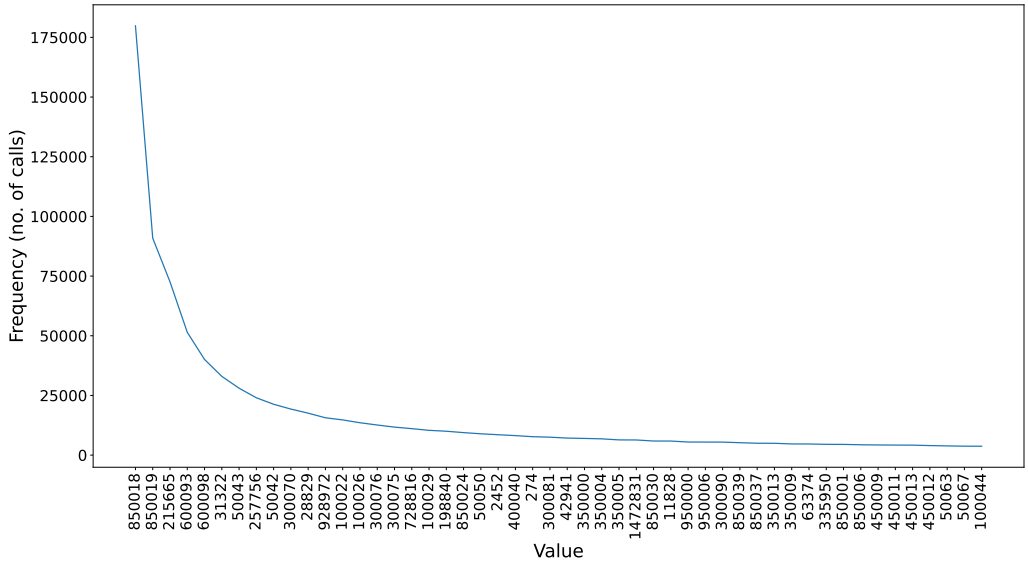

Figure 8. Output of the Zipfian workload generator in the YCSB as customised for D-Thespis for the top 50 values during a run.

5.6. Performance Benchmarks: Results

In this section, we summarise the results of all six thousand, eight hundred and four (6804) benchmark executions and discuss the most salient observations from our analysis. We also illustrate our results in a series of graphs while referring to variations across the dimensions discussed in Section 5.4.

Each graph in Figures 9–11 illustrates statistics obtained from a set of twenty-four (24) benchmark runs, carried out against a specific configuration of D-Thespis (i.e., a D-Thespis SUT) and a specific (and comparable) configuration of REST API executing operations against the relational DBMS (i.e., a DBMS SUT). Each graph represents benchmark runs with two of our benchmark dimensions (the type of workload and the number of virtual users) pegged to particular values for all the twenty-four (24) benchmark runs that it illustrates. The graphs give the throughput (operations per second) of three (3) runs of the benchmark workload against each SUT across four (4) different variations of the data set size dimension. The graphs also show the average latency (in milliseconds) of READ and WRITE operations of both SUTs, for the twelve (12) benchmark runs against each of the two (2) SUTs.

Each of the visuals in Figures 12–15 illustrates nine (9) graphs, with each graph giving twenty-eight (28) values for two specific SUTs: a specific D-Thespis SUT using the Microsoft Orleans actor framework and a specific (and comparable) DBMS SUT. Each graph within these figures shows the percentage difference in average throughput of the D-Thespis SUT relative to the average throughput of the comparable DBMS SUT under study, for a range of values of the data set size and number of virtual users dimensions. Effectively, the statistic S represented by each bar in these graphs is calculated as follows:

1. Let TD_1, TD_2 and $TD_3 \equiv$ the throughput (operations per second) of the D-Thespis SUT under study, obtained by executing three (3) benchmark workloads for a particular benchmark variation (i.e., a particular set of values for each dimension discussed in Section 5.4).
2. Let TB_1, TB_2 and $TB_3 \equiv$ the throughput (operations per second) obtained like TD_1, TD_2 and TD_3, but for the DBMS SUT under study.
3. $AD = \frac{TD_1+TD_2+TD_3}{3}$
4. $AB = \frac{TB_1+TB_2+TB_3}{3}$
5. $S = \frac{AD-AB}{AD} \times 100$

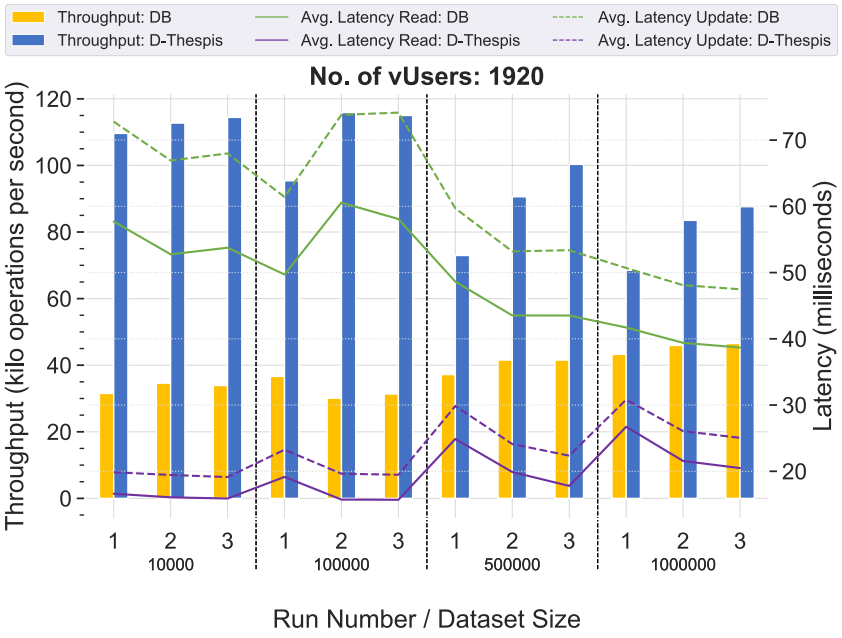

Figure 9. Results of D-Thespis SUTs `DT_3S_1DC_0` and `DB_3S_1DC` (85% READ, 15% WRITE).

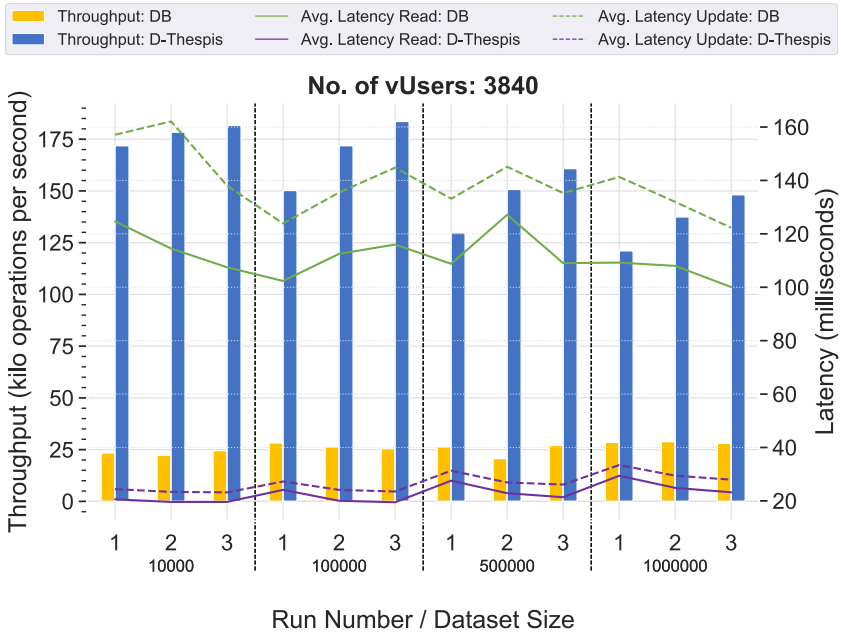

Figure 10. Results of D-Thespis SUTs `DT_5S_1DC_0` and `DB_5S_1DC` (65% READ, 35% WRITE).

Figure 16 also illustrates nine (9) graphs. Each of these graphs, like the ones in Figures 13–15, shows the percentage difference in average throughput of a D-Thespis SUT relative to the average throughput of a comparable DBMS SUT. However, in the case of this Figure 16, the D-Thespis SUT is configured to use the Akka.NET actor toolkit.

Finally, the visuals in Figures 17–19 are also like the ones in Figures 13–15, but they show the difference in average throughput of a specific D-Thespis SUT using the Microsoft Orleans actor framework, scaled horizontally across five, three and two machines (respectively, Figures 17–19), compared to the throughput of a different D-Thespis SUT using the Akka.NET actor toolkit and running on a single machine. Therefore Figures 17–19 illustrate how the throughput of D-Thespis changes as the system is scaled horizontally over several machines in a single data centre.

Figure 11. Results of D-Thespis SUTs `DT_1S_1DC_0` and `DB_1S_1DC` (65% READ, 35% WRITE).

5.6.1. The Cache Effect

Figure 9 illustrates the performance characteristics for a workload with a segmentation of 85% READ and 15% WRITE operations, with 1920 virtual users, executed three times over four data set sizes. This case considers the `DT_3S_1DC_0` and `DB_3S_1DC` SUTs, specifically running a D-Thespis configuration with three servers for the REST API and three servers for the Orleans cluster, compared to running a REST API over three servers and accessing the RDBMS directly.

It can be noted that for D-Thespis, the performance improves as more runs are executed. This "cache effect" was observed for all benchmark runs and is in line with the observations reported for Thespis [12]. Indeed, for smaller data set sizes, the improvement in throughput from the first run to the third one is less marked than for larger dataset sizes. This is due to the fact that with a smaller data set size, the cache is primed more quickly and the benefits of reading from the cache are observed earlier in the benchmark execution.

5.6.2. D-Thespis and the RDBMS

Figure 12 gives an analysis of the percentage difference in average throughput when comparing SUT `DT_5S_1DC_0` to `DB_5S_1DC` across all benchmark combinations. Hence, the graphs illustrate how the throughput of D-Thespis running on five servers for the REST API and five servers for the Microsoft Orleans cluster compares to the throughput of a REST API running on five servers that queries the RDBMS directly. Similarly, Figures 13–15 illustrate the comparison of `DT_3S_1DC_0` vs. `DB_3S_1DC`, `DT_2S_1DC_0` vs. `DB_2S_1DC` and `DT_1S_1DC_0` vs. `DB_1S_1DC`, respectively.

Figures 12–14 show that D-Thespis has a higher throughput than the RDBMS under most conditions. For example, as per Figure 12, for the scenario with a cluster size of 5, 3480 virtual users and a workload of 65% READ and 35% WRITE, the throughput of D-Thespis reaches an improvement of 800% compared to the RDBMS. Analysing the detail of this particular scenario shown in Figure 10, we see that D-Thespis is faster than the RDBMS in all benchmark executions, including the first execution where the benefits of the cache produces the least benefits (in line with the "cache effect" discussed earlier).

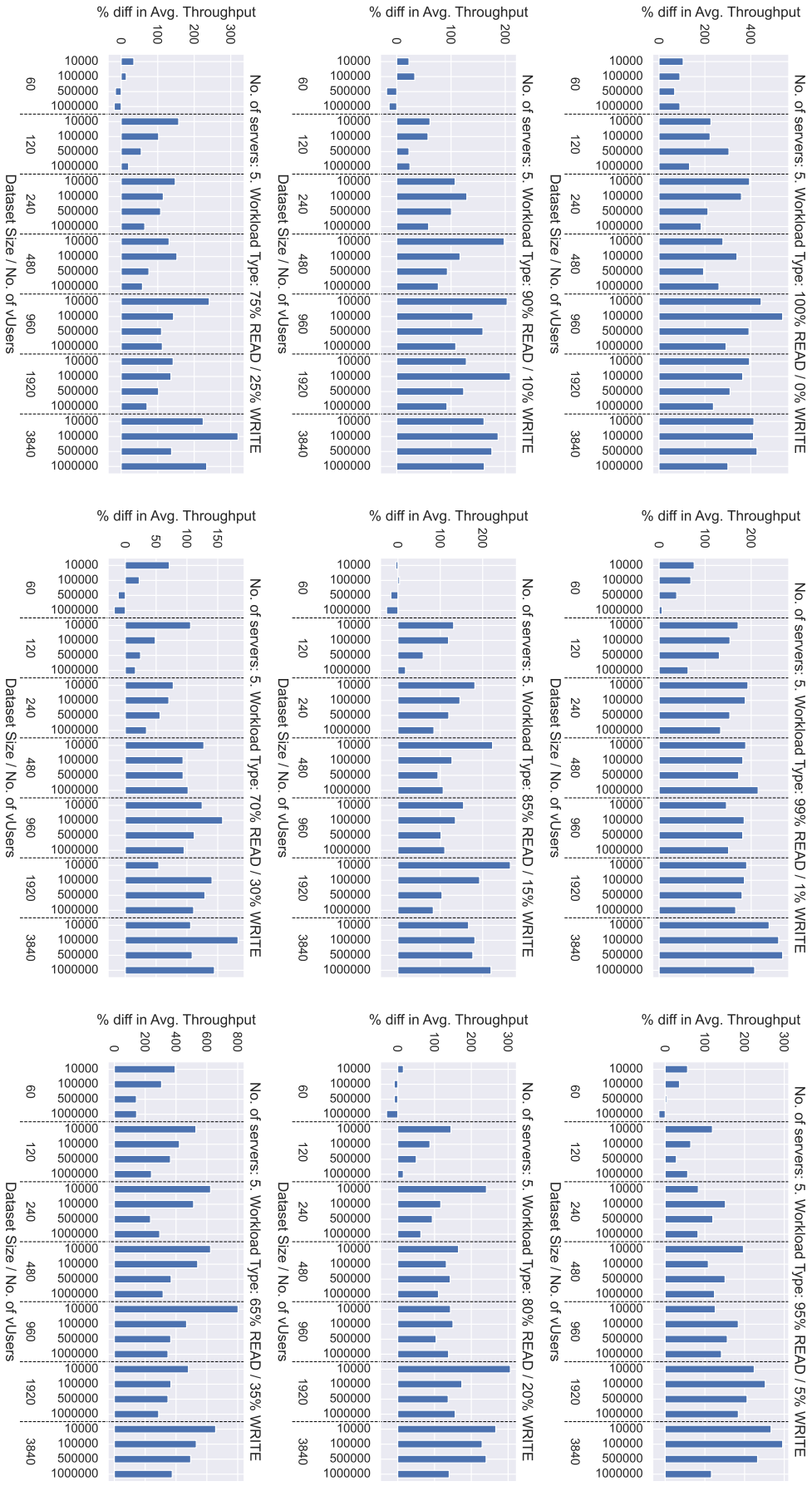

Figure 12. Percentage difference in average throughput for SUT `DT_5S_1DC_0` vs. `DB_5S_1DC`.

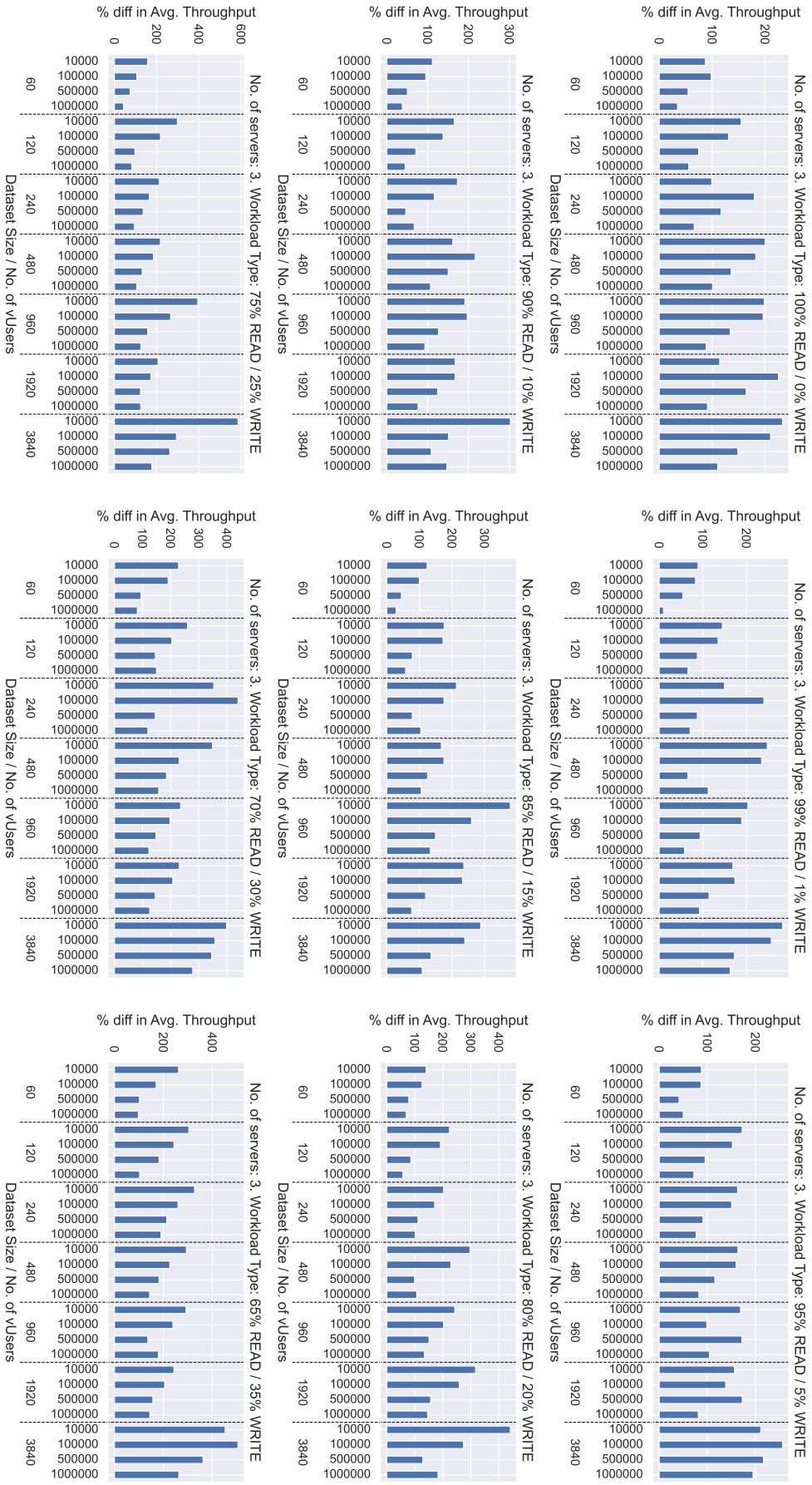

Figure 13. Percentage difference in average throughput for SUT `DT_3S_1DC_0` vs. `DB_3S_1DC`.

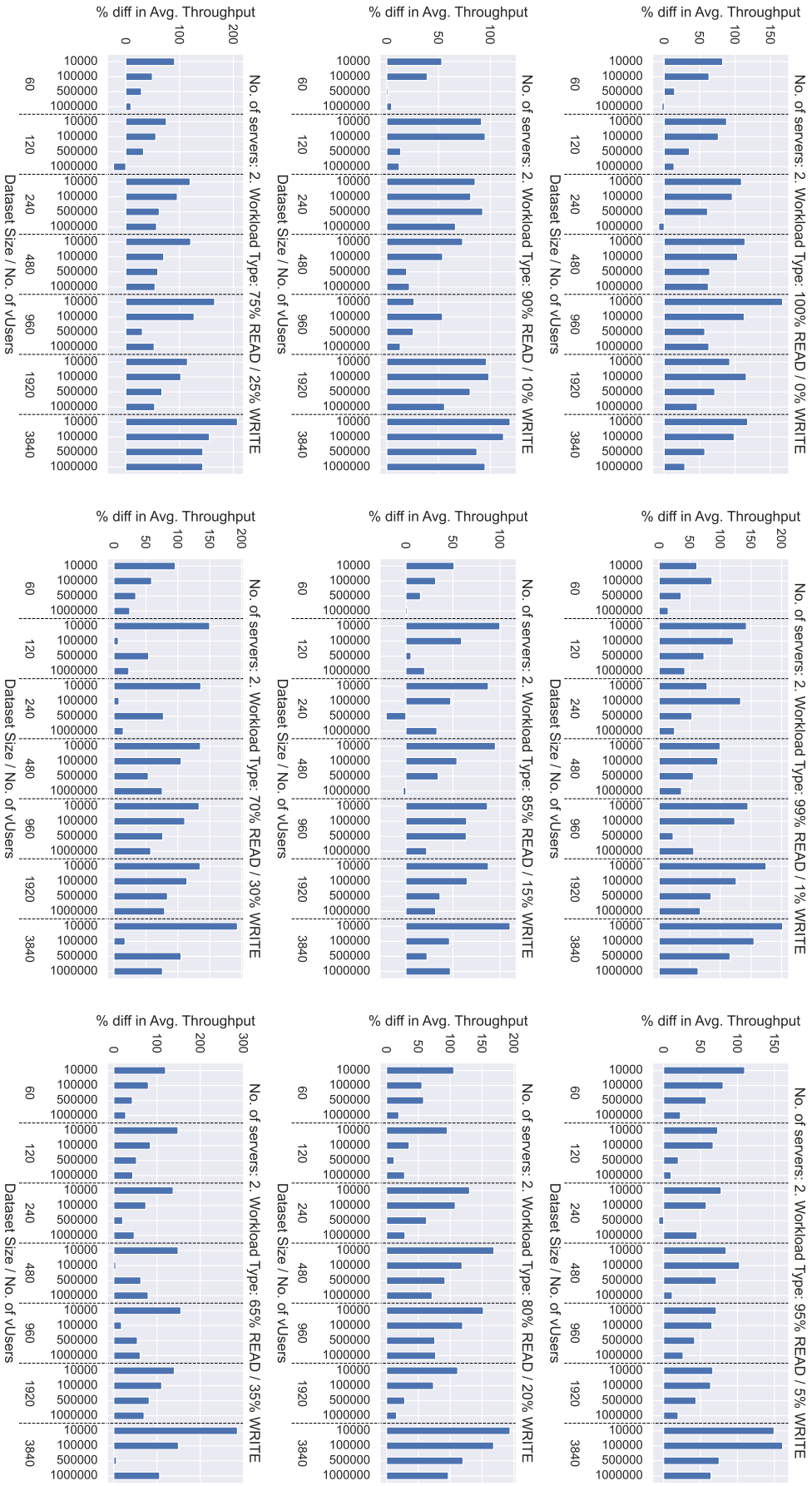

Figure 14. Percentage difference in average throughput for SUT `DT_2S_1DC_0` vs. `DB_2S_1DC`.

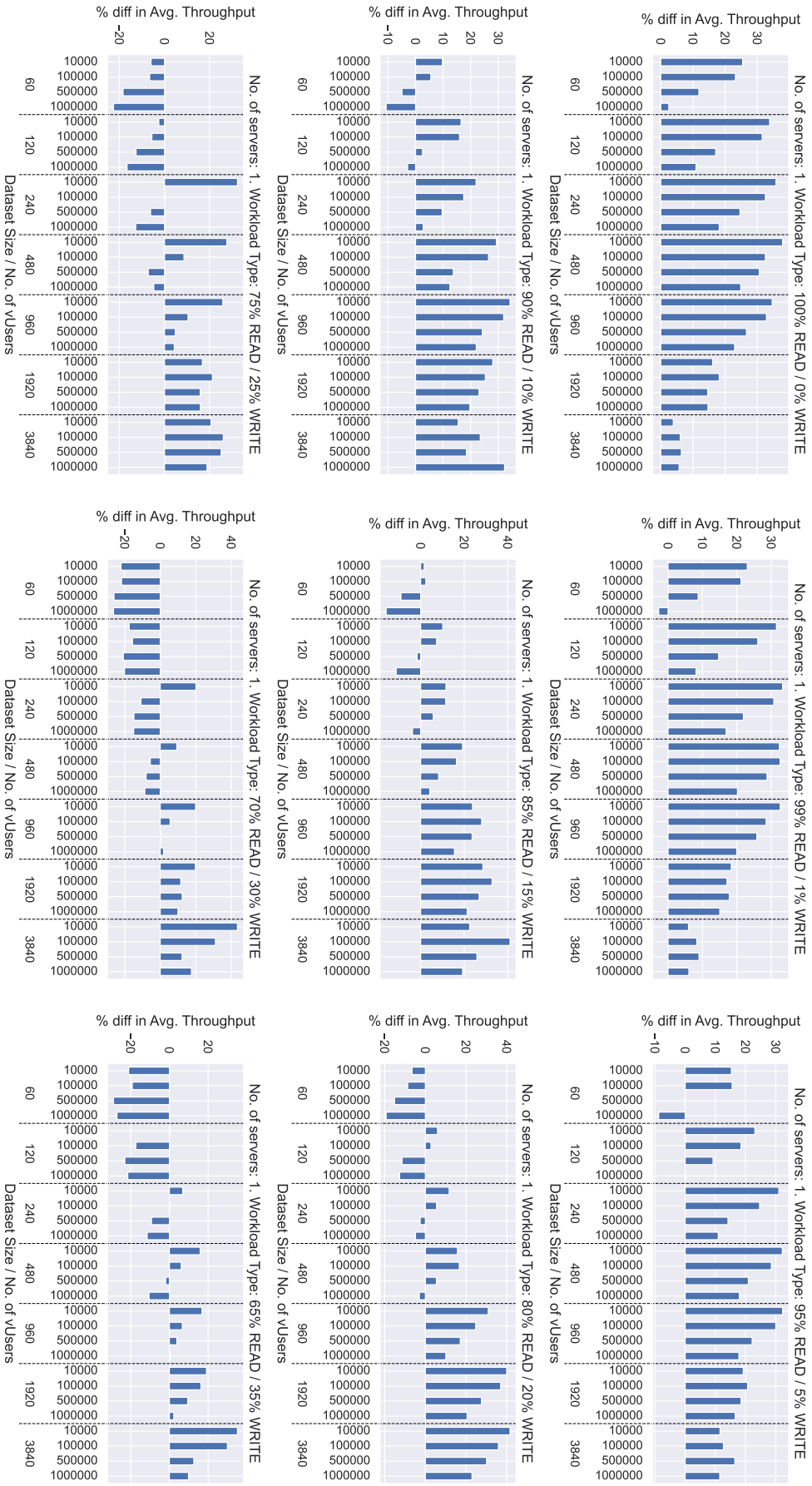

Figure 15. Percentage difference in average throughput for SUT `DT_1S_1DC_0` vs. `DB_1S_1DC`.

However, the comparison of DT_1S_1DC_O vs. DB_1S_1DC in Figure 15 shows that the throughput of D-Thespis degrades as the percentage of WRITE operations increases. In some scenarios, indicated by negative values in the graphs, the D-Thespis middleware is slower than the relational DBMS. Figure 11 shows the detail of one such scenario, specifically the respective detail of the benchmark execution with 480 virtual users and a workload of 65% READ and 35% WRITE from Figure 15. In this scenario, D-Thespis outperforms direct RDBMS access only for the two smaller data sets. This shows that the version of D-Thespis running the Microsoft Orleans actor framework and tuned for horizontal scalability introduces substantial overheads that, in some cases, make it significantly slower than direct access to the RDBMS. This is especially encountered in cases where the hit ratio on the distributed cache is relatively low, such as for write-heavy workloads over a larger dataset, as detailed in Figure 11. These results are different than those achieved for Thespis [12], where the middleware running on a single server was faster than an application programming interface (API) with direct RDBMS access.

Conversely, the comparison of DT_1S_1DC_A vs. DB_1S_1DC in Figure 16 shows that when D-Thespis is deployed on a single server and configured favourably (using the Akka.NET actor model implementation), it always outperforms direct access to the RDBMS. In most cases, the difference in throughput is considerably higher on D-Thespis, e.g., D-Thespis is >150% faster when handling a 100% READ workload from 3840 virtual users over a 10,000 record dataset.

5.6.3. Horizontal Scalability

Figure 17 gives an analysis of the percentage difference in average throughput when comparing SUT DT_5S_1DC_O to DT_1S_1DC_A across all benchmark combinations. The graphs illustrate the change in throughput for D-Thespis running on five servers for the REST API and five servers for the Microsoft Orleans cluster compared to the throughput of D-Thespis running on a single machine and optimally configured with the Akka.NET actor model implementation.

Similarly, Figures 18 and 19 illustrate the comparison of DT_3S_1DC_O vs. DT_1S_1DC_A and DT_2S_1DC_O vs. DT_1S_1DC_A, respectively. The comparison of DT_5S_1DC_O vs. DT_1S_1DC_A shows that the throughput of D-Thespis in general improves considerably when it is horizontally scaled in a single data centre. For example, the performance of D-Thespis improves by more than 300% when handling a workload of 65% READs and 35% WRITEs from 3840 virtual users, for all datasets. This is over and above the improvement in the same scenarios of DT_1S_1DC_A when compared to direct access to the RDBMS, as shown in Figure 15. Only a few cases show that D-Thespis hosted on one machine outperforms the configuration of D-Thespis that leverages horizontal scalability, and these are cases where the system sustains a relatively low volume of requests (i.e., workloads generated by 60 virtual users).

From Figure 18, SUT DT_3S_1DC_O also performs better than DT_1S_1DC_A. In contrast to DT_5S_1DC_O, DT_3S_1DC_O outperforms DT_1S_1DC_A in all cases, but in general, the improvement in throughput is less considerable. For example, the performance of D-Thespis improves by more than 200% when handling a workload of 65% READs and 35% WRITEs from 3,840 virtual users, for all datasets, as compared to the improvement of more than 300% by DT_5S_1DC_O.

Finally, from Figure 14, SUT DT_2S_1DC_O also performs better than DB_2S_1DC in most cases, but, as shown in Figure 19, there are several cases where the horizontally scaled configuration of D-Thespis performs worse than when D-Thespis is deployed on a single server and configured optimally to use the Akka.NET actor model implementation. This indicates that the overheads of horizontal scaling are not fully amortised when only a single machine is added.

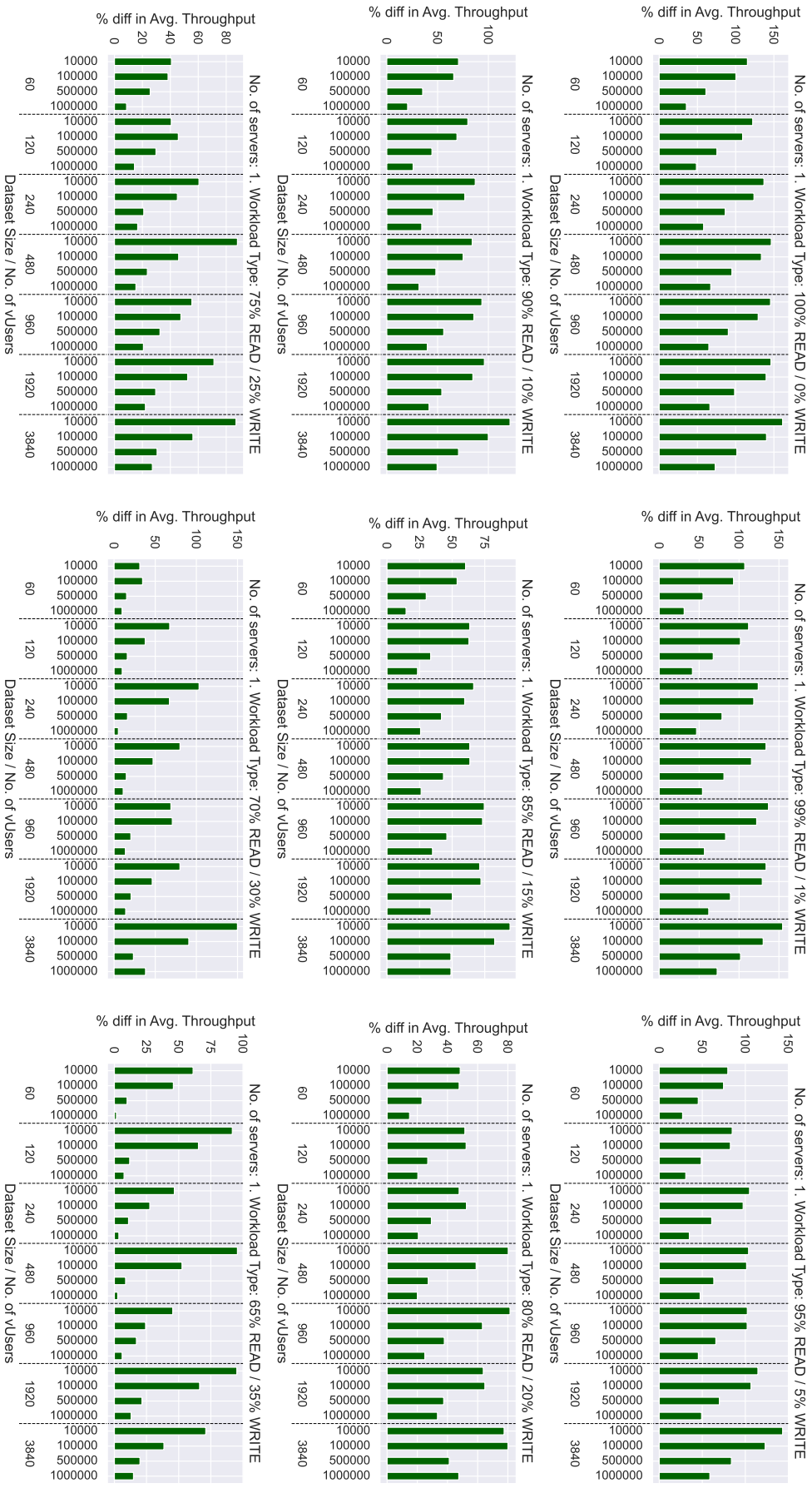

Figure 16. Percentage difference in average throughput for SUT `DT_1S_1DC_A` vs. `DB_1S_1DC`.

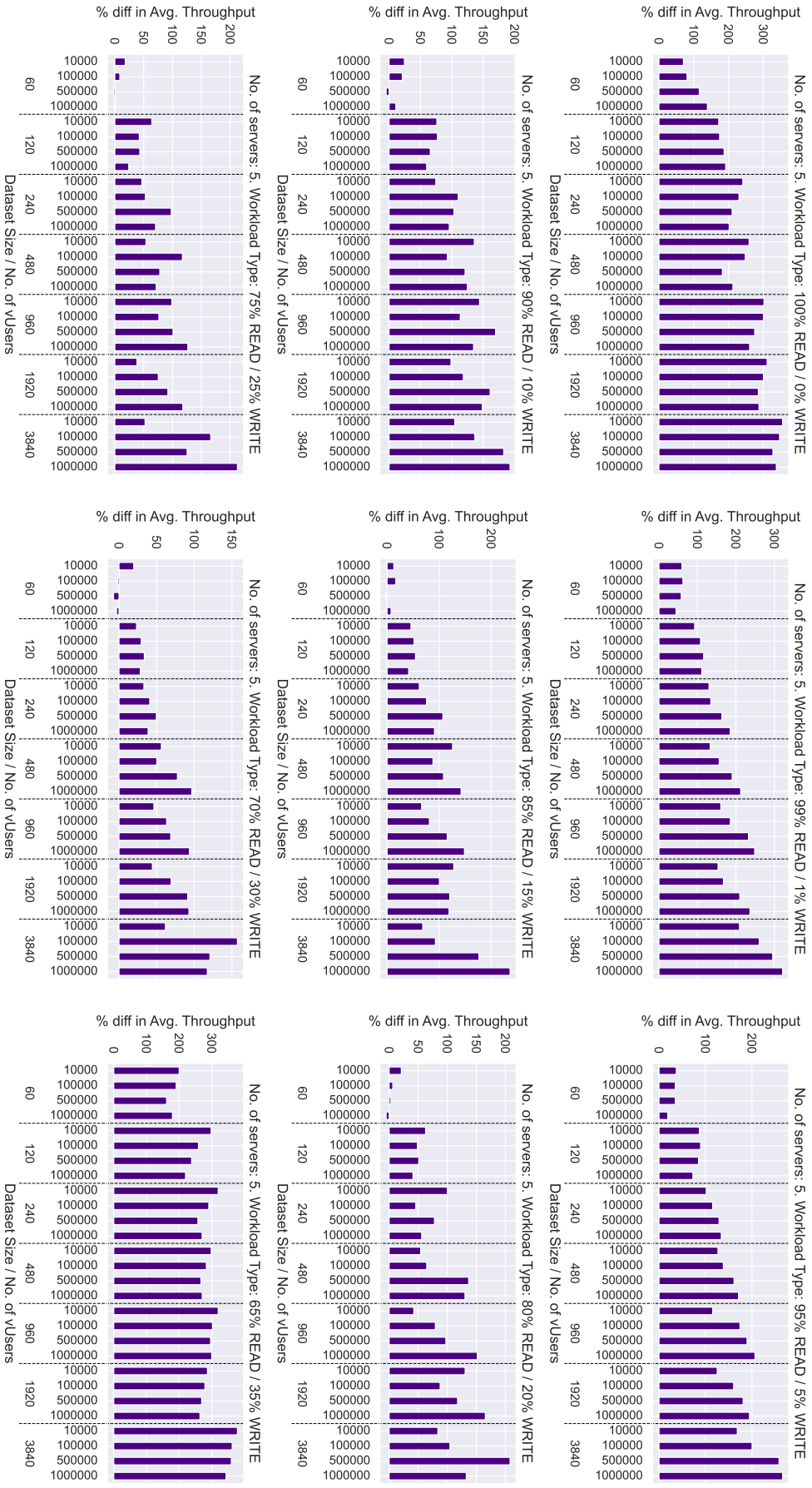

Figure 17. Percentage difference in average throughput for SUT `DT_5S_1DC_O` vs. `DT_1S_1DC_A`.

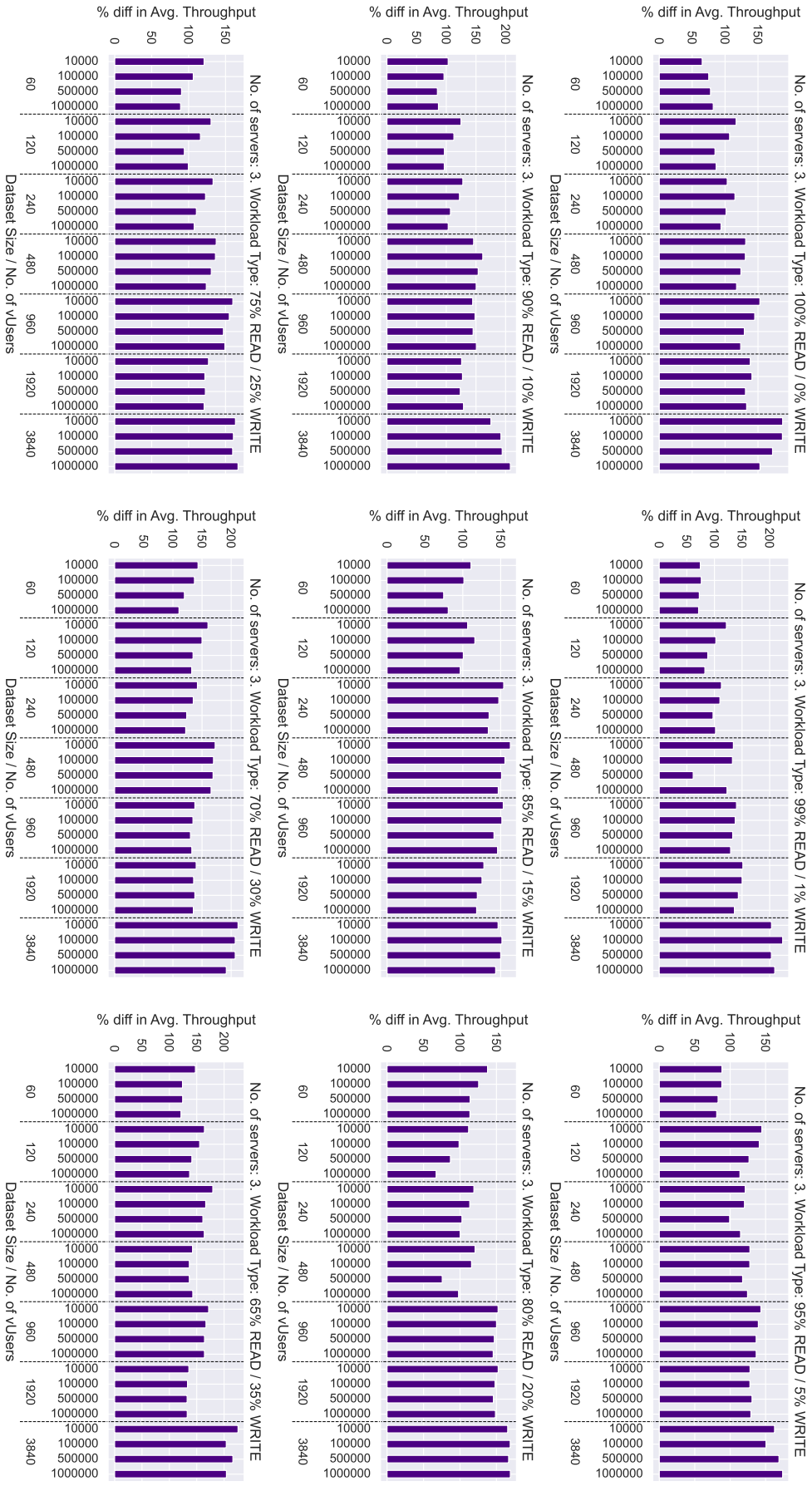

Figure 18. Percentage difference in average throughput for SUT `DT_3S_1DC_O` vs. `DT_1S_1DC_A`.

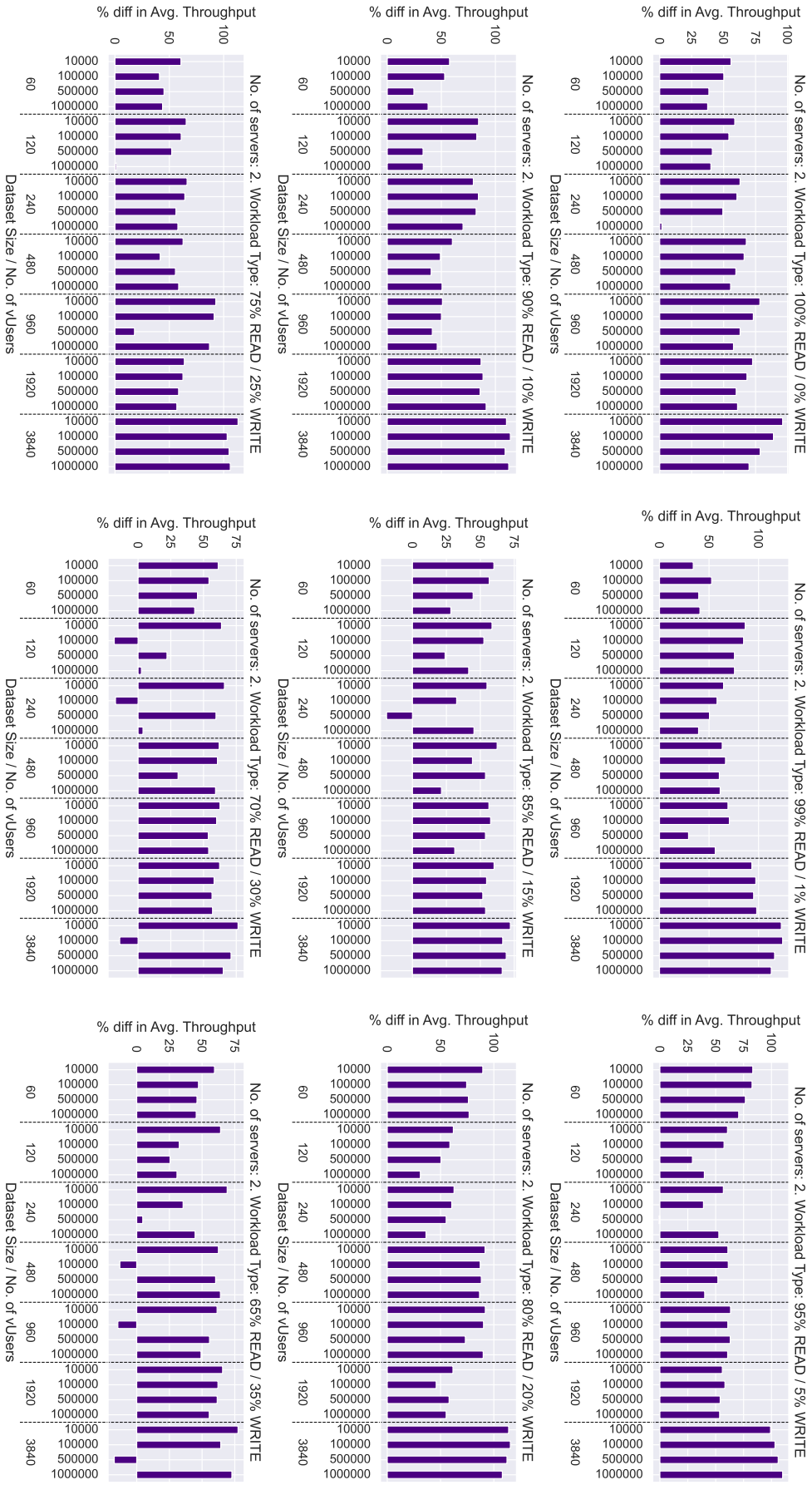

Figure 19. Percentage difference in average throughput for SUT `DT_2S_1DC_O` vs. `DT_1S_1DC_A`.

5.7. Update Visibility Latency Benchmark: Configuration and Procedure

Update visibility latency was measured in terms of how long it takes for a data entity event to be replicated to a remote data centre. Since the D-Thespis protocol timestamps events use vectors of physical timestamps, rather than a scalar physical timestamp (as in Thespis), under normal conditions, a data entity event is made visible to clients of a remote data centre as soon as it is replicated, as discussed in our objective to improve update visibility latency [13]. This analysis therefore focuses on the efficiency of the replication protocol in the context of varying workloads.

D-Thespis was deployed in two data centres, `europe-west-1d` (Belgium) and `us-west1-c` (Oregon, USA). In both data centres, D-Thespis was deployed in the configuration `DT_2S_1DC_0`, i.e., using a Microsoft Orleans actor cluster horizontally scaled over two machines within each data centre, using hardware of the specifications given in Table 1.

The network round trip time (RTT) between both data centres was measured to be approximately 135 ms using the ping network diagnostic tool and is in line with the RTT reported in other works for similar infrastructure setups [45].

In both data centres, the event replication protocol was configured to be executed every 10 ms, consistent with similar methods described in the literature [19]. The event snapshotting protocol was executed every 100 ms, a period that was slightly less than the RTT between both data centres.

Update visibility latency was measured by running a workload segmented at 65% READ/35% WRITE operations in `us-west1-c` for 60 s over a dataset of one million records. In `europe-west-1d`, we logged the difference in time between the event generation time in `us-west1-c` and the time it was received in `europe-west-1d` for each event received by the replication protocol running in `europe-west-1d`. Similar to approaches in the literature [19], this measure of update visibility latency is an approximation because

1. The time of event generation does not include any latency encountered in saving the event to the event log in `us-west1-c`, while the replication protocol can only consider the event for replication once it is saved;
2. The clocks in the different data centres were not assumed to be highly synchronised and the measure could be happening in the presence of some clock skew. The computer processors are equipped with in-built physical clocks. These operate independently can have a different rate of progress, resulting in different frequencies. This, in turn, produces clock skew, which is the difference between the time produced by a clock and time produced by the perfect clock or by another physical clock [46].

However, it was assumed that the RTT between the data centres was much larger than the clock skew and the duration to save the event to the event log, making such discrepancies negligible.

5.8. Update Visibility Latency Benchmark: Results

Figures 20 and 21 show how the update visibility latency varies when the D-Thespis in `us-west1-c` was exercised with a workload from 30 virtual users and 60 virtual users, respectively, together with the relative cumulative frequency plots.

From Figure 20, we ascertain that most events were received in `europe-west-1d` in less than 400 ms when running a workload of 30 virtual users in `us-west-1c`. Ninety-five percent of the events were received in around 2.4 s or less, with the minimum observed latency being 72 ms.

In Figure 21, we observe that when running a workload of 60 virtual users in `us-west-1c`, there was an overall increase in update visibility latency. In this case, the majority of the events were received in `europe-west-1d` in 700 ms or less. Ninety-five percent of the events were received within less than 2.5 s, and the minimum observed latency was 193 ms.

These results indicate a correlation to both the network round trip time between the data centres, as well as the volume of events that need to be replicated at any point in time. No event can arrive in a remote data centre in less than $\frac{1}{2}$ of RTT, while the more events generated in a data centre, the longer it takes for an event to be replicated to a remote data centre.

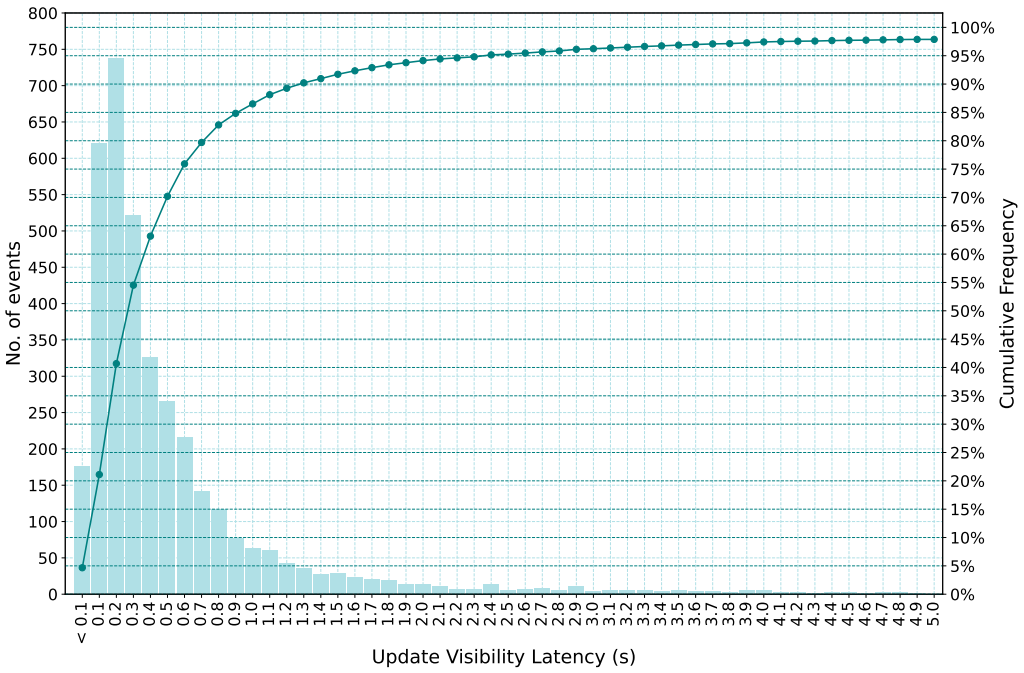

Figure 20. D-Thespis update visibility latency for events generated by 30 virtual users.

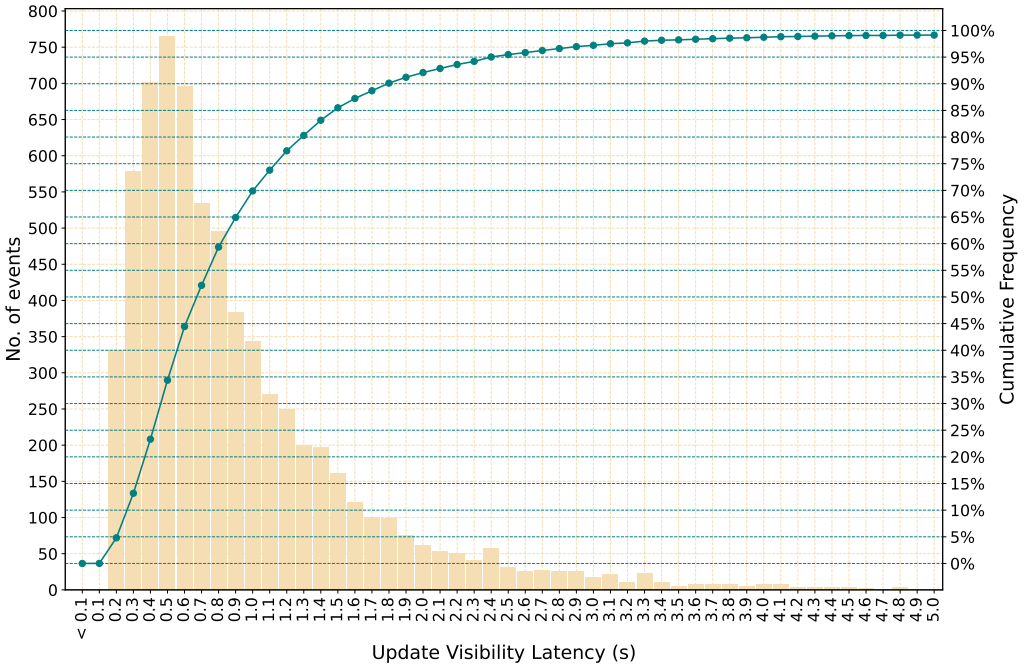

Figure 21. D-Thespis update visibility latency for events generated by 60 virtual users.

5.9. Memory Profile Benchmark: Configuration and Procedure

The memory-bound nature of D-Thespis necessitates awareness of the memory footprint characterisation of the system, which was studied using a benchmark designed for this purpose.

We deployed D-Thespis in one data centre, `europe-west-1d` (Belgium), using the configuration `DT_2S_1DC_0`, similar to that used in Section 5.7. The benchmark consisted of running a workload of 100% READ operations for 15 min over a dataset of one million records. The memory footprint of the specific GKE workload as reported in the GCP console was then noted.

5.10. Memory Profile Benchmark: Results

Figure 22 shows a profile of the memory footprint before, during and after the benchmark. The top graph shows the memory footprint for one of the replicas in the GKE workload, while the bottom graph shows the total memory used by the GKE workload running two replicas.

The total memory usage of D-Thespis hovered at around 10 GB at peak. The graphs also show that the memory usage was evenly distributed across both replicas, thus showing the efficacy of the actor placement strategy configured in the implementation based on the Microsoft Orleans framework in distributing the entity actors evenly within the cluster. The results also show how memory usage started dropping once the benchmark finished and the system was left idle. This was expected behaviour, as a period of idle time allows memory to be reclaimed by both the managed memory garbage collector of the .NET framework, as well as the actor graceful passivisation routine of the Microsoft Orleans framework.

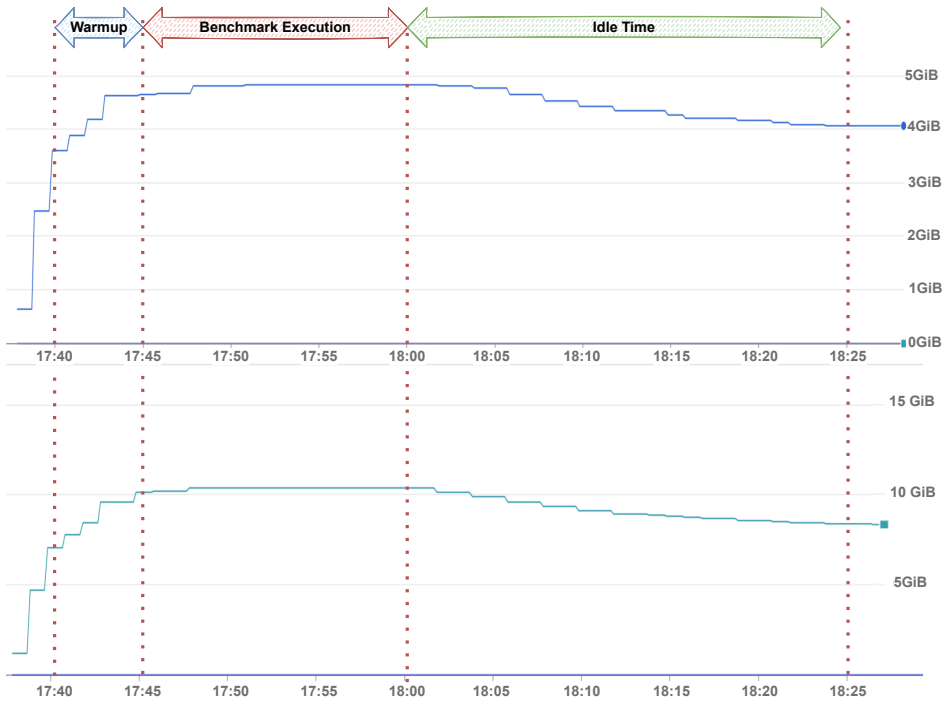

Figure 22. D-Thespis Memory Footprint: Memory usage in GKE for 1 workload (**top**) and total for 2 workloads (**bottom**).

6. Discussion

In our prior study [13], we established the D-Thespis framework. The D-Thespis middleware was designed to meet the requirements and objectives of Thespis [12]. It achieves this by implementing a causal consistency model based on the actor model of computation. Additionally, D-Thespis simplifies the intricacies of causal consistency by providing a REST interface that is familiar to application developers.

In addition, our design of D-Thespis addressed another crucial aspect: the capacity to horizontally scale elastically. This property guarantees that our causally consistent middleware can be used for data sets that have specific workloads, even if an in-memory representation using actors cannot fit into the main memory of a single machine due to

its memory-bound characteristics. D-Thespis stands apart from state-of-the-art systems by offering elastic horizontal scalability, a feature that is not seen in systems that rely on a strictly partitioned key-value data format. Unlike other approaches, our method does not rely on any predetermined number of servers. The number of servers can fluctuate over time and may change across various data centres. The D-Thespis architecture may be easily deployed in many data centres, with the flexibility to optimise the design of each data centre based on the predicted workloads and geographic location.

Our prior research [13] validates the D-Thespis approach theoretically. We give detailed explanations as to why the newer D-Thespis approach is superior to the Thespis approach (e.g., improved performance and improved update visibility latency). We also provide sketches of proofs that illustrate the correctness of the D-Thespis protocols.

In this paper, we further confirm our initial findings empirically. We follow the system design and algorithms of D-Thespis [13] to achieve an efficient implementation. The implementation is grounded on sound software engineering concepts, which, among other advantages, allows us to achieve a setup that is not forced to trade off optimisations for a single-machine setup to accommodate a distributed setup within a data centre.

The empirical results from our previous research [33] also guided us in selecting a technological stack that is well-suited for achieving high performance. For instance, while various implementations of the actor model offer the same assurances that may be utilised to ensure the accuracy of causal consistency, empirical evaluation shows that the performance in terms of system throughput can differ, particularly in relation to horizontal scalability [33]. Given these findings, we had previously developed the D-Thespis architecture [13] to separate the protocol responsible for ensuring causal consistency from the individual implementations of the various tiers. This work utilised an architectural design to create a version of D-Thespis that is optimised for both single-node and horizontally scaled configurations. The design contains the protocol in both the Actor model (as described by Agha [23]) and the Virtual Actor model (as described by Bernstein et al. [47]).

In this work, we also study the applicability of D-Thespis to modern hardware infrastructure. In fact, our implementation D-Thespis shows that it can be deployed natively in a public cloud infrastructure but without depending on vendor-specific capabilities. Thus, we show that D-Thespis generalises also to multi-cloud infrastructure setups.

Furthermore, here, we also study the performance of D-Thespis empirically. A standard benchmarking tool, YCSB, was customised to include connectivity to D-Thespis. This exercise was driven both by the need to run standard performance benchmarks on D-Thespis and also to analyse the approachability of the D-Thespis REST interface. In fact, here we have shown that YCSB (a Java application) could be adapted to connect to D-Thespis and execute the necessary workloads without having any intrinsic knowledge of causal consistency and without relying on any additional components or drivers specific to its ecosystem.

The performance benchmarks showed the efficacy of having a configurable design and implementation. Specifically, deploying D-Thespis on a single machine but tuning it for horizontal scalability resulted in a degradation in performance within a single data centre. Arguably, the workload of an information system that makes use of D-Thespis would benefit from the distributed nature of the middleware across more than one data centre, and thus still improve, in terms of performance, over an installation that depends on one RDBMS server. Nonetheless, by tuning D-Thespis correctly, it was shown that performance could be improved considerably in most of the scenarios tested, even in a single data centre.

In addition, we examined the performance features of the replication protocol and found that the duration for the impacts of a WRITE operation to become apparent to clients in a distant data centre is primarily consistent with the time it takes for data to travel between two data centres. Nevertheless, these benchmarks also show that the delay in updating visibility is influenced by the number of events that a data centre must distribute.

Finally, the effectiveness of utilising robust, widely accepted implementations of the actor model is demonstrated in the profiling of memory characteristics. These findings demonstrated that the benchmark workload, when applied to a dataset containing one million records, produced a memory footprint that was small enough to be accommodated inside the primary memory of a single computer. However, it also demonstrated that while horizontally expanding the middleware, the memory usage was uniformly divided among the servers in the same data centre.

6.1. Synopsis

Based on our observations, we can now refer to the questions posed at the start of Section 5 and give a summary to each.

Q1: *What is the impact of the workload profile (i.e., read/write ratio) on the performance of D-Thespis under every configuration?*
Summary: The D-Thespis approach introduces the "cache effect", as discussed in Section 5.6.1. Our results show that the throughput of D-Thespis is higher as the read ratio increases. Nonetheless, our benchmarks show that D-Thespis configuration is able to produce higher throughput from an RDBMS across all workload profiles.

Q2: *Does the dataset size have any bearing on the performance of D-Thespis under every configuration?*
Summary: The throughput of D-Thespis is generally higher for smaller data sets when running on a single server but generally higher for larger data sets when running on multiple servers and sustaining a sufficiently large workload. Here, we see that a higher cache–hit ratio is achieved in a shorter time in smaller data sets; however, the effects of contention (i.e., a larger chance of concurrent requests to the same data record) on a smaller data set causes overheads when running on multiple servers (therefore, using a distributed cache rather than an in-memory actor-based cache). We consider this a positive result; running D-Thespis on a single server fits better for use cases that handle smaller data sets, while a larger data set is likely to require D-Thespis to run on multiple servers.

Q3: *How does the performance of all D-Thespis configurations change in relation to the number of requests submitted?*
Summary: Our results show that, in general, the throughput of D-Thespis increases as the workload increases. This correlates with our observation that the hardware infrastructure was not saturated in most of our tests. Naturally, this pattern is not considered infinite, i.e., given a fixed infrastructure size, the system throughput will degrade as the workload size increases to a level that saturates the available hardware resources. Indeed, the last case in Figure 16 is such an example: with D-Thespis running on a single server, increasing the workload to 3840 virtual users with a read/write ratio of 65%/35% yields lower throughput than the same type of workload with 1920 virtual users. Nonetheless, by introducing the concept of elastic horizontal scalability in D-Thespis, larger workloads can be handled by introducing additional hardware capacity.

Q4: *How does the performance of D-Thespis under every configuration compare to the* `DB_1S_1DC` *and* `DB_NS_1DC` *SUTs, under all the different workload conditions?*
Summary: D-Thespis is generally faster than the `DB_1S_1DC` and `DB_NS_1DC` SUTs, with two exceptions. Firstly, D-Thespis is slower than the `DB_1S_1DC` SUT when running on a single server and not configured correctly (i.e., when not configured to use the Akka.NET actor provider), as shown in Figure 15. Secondly, D-Thespis is slower than the `DB_NS_1DC` SUT when running on multiple servers and the workload is not sufficiently high, as shown, for example, in several graphs of Figure 12, where the workload was generated by 60 virtual users. We consider this a positive result in that D-Thespis should be configured correctly to achieve its best performance, and the system should not be horizontally scaled unless it needs to sustain a substantial workload.

Q5: *For `DT_NS_1DC_0`, what is the effect of horizontally scaling D-Thespis within a single data centre?*
Summary: The effect is generally positive, as discussed in **Q4**, and as shown in Figure 17, by horizontally scaling D-Thespis, the throughput could be improved by more than 300% in some scenarios. However, as shown in the results of Figure 19, scaling D-Thespis by a single machine might actually result in loss of performance, and therefore, our results indicate that a horizontally scaled installation of D-Thespis would benefit from running on three machines or more.

Q6: *What is the update visibility latency of D-Thespis? Thus, for `DT_NS_2DC_0`, how long does it take for an event to be visible in a remote data centre?*
Summary: As detailed in Section 5.8, our experiments indicate that, on average, the update visibility latency varies between 400 and 700 ms. The results also indicate a correlation between the update visibility latency, the network round trip time between the data centres, as well as the volume of events that need to be replicated at any point in time.

Q7: *How can the memory footprint of D-Thespis be characterised?*
Summary: During our experiments, D-Thespis did not require additional memory than that allocated to the machines on which it was installed, i.e., 32GB. By the results presented in Section 5.10, when D-Thespis was scaled horizontally, the memory requirements were evenly distributed across the available machines.

6.2. Strengths and Weaknesses

Causal Consistency: Our work addresses problems in distributed data management by adopting causal consistency. Based on our results, we postulate that causally consistent DBMSs are indeed a strong solution for distributed information systems. However, it is still a fact that causal consistency is not as prevalent as other types of data consistency models. Gaining the necessary knowledge to architect solutions based on causally consistent DBMS may constitute a steeper learning curve for application developers than other approaches, such as ones that rely on mixed-consistency models. Whilst not specific to our approach, we consider this a weakness.

Approach to Causal Consistency: We consider that our choice of the actor model of computation to model causal consistency has been a strong point throughout our various contributions, particularly allowing us to manage correctly and efficiently the complexities of concurrency and parallelism in a distributed system. Adopting other approaches, such as vectors of timestamps as the only meta-data to determine causality, also served us well and proved to be a correct and efficient way of modelling causal consistency. However, in some aspects, our approach can be considered unnecessarily restrictive. The actor model serialises operations on the same data item, even when it is not strictly necessary to do so; for example, it does not allow multiple READ operations for the same data item to proceed concurrently, even though it is safe to do so. Using a vector of physical timestamps to determine causality also does not provide any guarantees against false positives; every event is considered possibly causally related to every other event that happened before it, without considering that some events may happen sequentially in time, but are not necessarily related by causality and can therefore be allowed to be processed independently, and faster, without jeopardising the correctness of causal consistency.

Performance and Efficiency: Our choice to evaluate the performance of the various implementations in relative terms to the performance of an RDBMS is considered a strength of an empirical evaluation. We believe that the decision to adopt a distributed DBMS in an information system ecosystem should not be taken lightly, and that such a DBMS should only be opted for when the requirements of the information system cannot be satisfied by a centralised database. Therefore, the results of an empirical evaluation that contrasts the performance of an RDBMS to that of our causally consistent distributed DBMS constitute important information as to the applicability of such a distributed DBMS to an information system ecosystem. Nonetheless, we acknowledge that an empirical evaluation also has its limitations. For example, synthetic workloads (such as the ones

in our evaluation) are not always representative of the specific types of workloads that information systems are expected to handle. In reality, transactional loads for a particular information system may also change over time, and thus, even if the empirical evaluation is carried out using calibrated workloads, the results would only remain representative until the context changes. Finally, the performance of the distributed DBMS may also be significantly influenced by infrastructure components (e.g., performance characteristics of the machines, the efficiency of virtualisation and the speed of the network).

Infrastructure Compatibility: We showed that our approaches lend themselves well to being deployed in diverse types of infrastructures. All our implementations were installed and tested using commodity hardware. Even though we have deployed them in a public cloud infrastructure, we have sometimes used this in IaaS mode, which can be effectively replicated in any data centre. Our forage for a massively scalable solution in D-Thespis used container orchestration to facilitate the management of a large infrastructure. We consider that this introduced complexities in a similar manner as other horizontally scaled systems and thus represents a burden of our approach. It would require significant investment from a data owner, both in hardware and skillset, to maintain an elastically scaled setup of our middleware.

Developer Experience: We also consider that emphasising on exposing the causally consistent middleware via an intuitive REST interface to any information system was a solid choice. We felt the benefits of this approach even during our work. It allowed us to use the existing robust benchmarking tools, such as YCSB and OLTP-Bench, with minimal adaptations. We would therefore expect that the integration of our middleware in an information system ecosystem would be of similar complexity. We consider that exposing only this type of interface at the same time constitutes a weakness, as more efficient protocols (e.g., ones based on compressed binary formats) would be better suited for a high-performance transactional DBMS. Furthermore, the middleware's APIs should provide richer semantics to truly allow application developers to access the relational data set in the RDBMS in a causally consistent manner but with the same ease of access offered by the RDBMS's API.

7. Conclusions

This paper presents an implementation of D-Thespis, an approach that we described in earlier works [13] and which extends Thespis [12] to deliver causal consistency over a relational DBMS.

For every aspect of our implementation of the D-Thespis design, we describe the technical choices made. Subsequently, we perform a detailed empirical analysis of our implementation, comprising a total of six thousand and forty-eight (6048) test runs. Here, we show that D-Thespis can be deployed in a public cloud data centre, running on five machines, to sustain a throughput of up to 400 thousand requests per second, outperforming an RDBMS running on the same infrastructure by more than 300%.

Other empirical analysis efforts focused on measuring the update visibility latency of our implementation. Results from these tests showed that when D-Thespis was deployed on two data centres having an inter-DC network round trip latency of 135 ms, the effects of the majority of the data change operations in one DC were available in the other DC within 400 ms.

7.1. Contributions and Innovations

In this paper, we present multiple novel contributions. Firstly, we detail an implementation of the design of D-Thespis [13] and empirically show that a robust implementation of this design delivers enhancements over and above the novelties of the techniques in Thespis [12] in delivering causal consistency over a relational DBMS by introducing elastic horizontal scalability within a single data centre. Our technical choices for the implementation (e.g., using cloud-native capabilities) demonstrate how D-Thespis distinguishes itself from other approaches in the existing research in that it does not need to rely on a consistent

and static infrastructure setup in every data centre. The architecture of a horizontally scalable middleware, operating on datasets that are not strictly partitioned, proves particularly efficacious for information systems interfacing with full replicas of databases managed by relational database management systems.

Furthermore, our analysis of diverse capabilities in contemporary infrastructures and industry-standard actor model frameworks and libraries demonstrated that an efficient technique would leverage the strengths of different combinations of these capabilities. Benchmarks of an implementation of the middleware, based on the modular design given for D-Thespis, showed that such an approach could improve the performance in a single data centre even beyond the improvements of Thespis. Therefore, an information system connected to D-Thespis could expect to handle larger workloads more efficiently than if it connected directly to the relational DBMS, even in a single data centre. The architecture that we presented here offers a distinctive approach compared to related works, as the system is configurable and can be fine-tuned for both single-server and horizontally scaled deployments. This flexibility is achieved through the deliberate decoupling of the core protocol from the system model, allowing for adaptable configurations that can be optimised for various deployment scenarios. Such adaptability stands in contrast to many existing systems, where the protocol is often tightly integrated with the system model, limiting deployment options and scalability.

Lastly, this paper describes a cloud-native implementation of D-Thespis. The implementation adopts two different actor model frameworks. These vary in their approaches but still provide the necessary guarantees for the correctness of the middleware protocols. This demonstrates that the D-Thespis protocol is capable of generalising to multiple implementations of the actor mathematical model for concurrent computation. Furthermore, we show that a cloud-native implementation of D-Thespis can be deployed and configured for different use-cases.

7.2. Future Work

Several aspects are left as an exercise for future work, including enabling a more semantically rich API for D-Thespis such as proposed in [48], including the capability of WRITE transactions of transactional causal consistency [49] to improve the developer experience when handling real-world transactional workloads and considering more efficient event logging and replication mechanisms to improve further the solution's update visibility latency characteristics and running real-world workload simulations to widen the assessment of the performance of the middleware with more diverse transactional workloads and different infrastructures (e.g., infrastructures built on different cloud providers). Several aspects that would facilitate the industrialisation of D-Thespis are also in the scope of future work, including the development of binary interfaces as well as client-side libraries and relevant educational and technical documentation for an even smoother developer experience, engagement with other researchers and the developer community to foster adoption and onboard feedback in an evolution roadmap of the middleware, and expanding the software ecosystem around the middleware to facilitate installation and management operations (e.g., tooling for node visualisation, failure monitoring and alerting and automatic recovery).

Author Contributions: Conceptualization, C.C., J.G.V. and V.N.; Methodology, C.C., J.G.V. and V.N.; Software, C.C., J.G.V. and V.N.; Investigation, C.C., J.G.V. and V.N.; Writing—original draft, C.C., J.G.V. and V.N.; Writing—review & editing, C.C., J.G.V. and V.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research was carried out following the award of an Endeavour B Scholarship Scheme funded out of national funds from the Government of Malta.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Brewer, E.A. Towards robust distributed systems. In Proceedings of the PODC, Portland, OR, USA, 16–19 July 2000; Volume 7.
2. Gilbert, S.; Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* **2002**, *33*, 51–59. [CrossRef]
3. Herlihy, M.P.; Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1990**, *12*, 463–492. [CrossRef]
4. Vogels, W. Eventually consistent. *Commun. ACM* **2009**, *52*, 40–44. [CrossRef]
5. Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. (TOCS)* **1998**, *16*, 133–169. [CrossRef]
6. Elbushra, M.M.; Lindström, J. Eventual Consistent Databases: State of the Art. *Open J. Databases (OJDB)* **2014**, *1*, 26–41.
7. Ahamad, M.; Neiger, G.; Burns, J.E.; Kohli, P.; Hutto, P.W. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.* **1995**, *9*, 37–49. [CrossRef]
8. Mahajan, P.; Alvisi, L.; Dahlin, M. *Consistency, Availability, and Convergence*; Technical Report; University of Texas at Austin: Austin, TX, USA, 2011; Volume 11.
9. Bailis, P.; Ghodsi, A.; Hellerstein, J.M.; Stoica, I. Bolt-on causal consistency. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 761–772.
10. Spirovska, K.; Didona, D.; Zwaenepoel, W. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 527–542. [CrossRef]
11. Braun, S.; Bieniusa, A.; Elberzhager, F. Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems. In Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data, Online, 26 April 2021; pp. 1–12.
12. Camilleri, C.; Vella, J.G.; Nezval, V. Thespis: Actor-Based Causal Consistency. In Proceedings of the 2017 28th International Workshop on Database and Expert Systems Applications (DEXA), Lyon, France, 28–31 August 2017; pp. 42–46. [CrossRef]
13. Camilleri, C.; Vella, J.G.; Nezval, V. D-Thespis: A Distributed Actor-Based Causally Consistent DBMS. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems LIII*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 126–165.
14. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **1978**, *21*, 558–565. [CrossRef]
15. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. (TOCS)* **2013**, *31*, 8. [CrossRef]
16. Terry, D.B.; Theimer, M.M.; Petersen, K.; Demers, A.J.; Spreitzer, M.J.; Hauser, C.H. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Oper. Syst. Rev.* **1995**, *29*, 172–182. [CrossRef]
17. Lakshman, A.; Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
18. Lloyd, W.; Freedman, M.J.; Kaminsky, M.; Andersen, D.G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 23–26 October 2011; pp. 401–416.
19. Du, J.; Iorgulescu, C.; Roy, A.; Zwaenepoel, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, 3–5 November 2014; pp. 1–13.
20. Spirovska, K.; Didona, D.; Zwaenepoel, W. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; pp. 1–12.
21. Kulkarni, S.; Demirbas, M.; Madeppa, D.; Bharadwaj, A.; Leone, M. *Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases*; Springer: Berling/Heidelberg, Germany, 2014.
22. Hewitt, C.; Bishop, P.; Steiger, R. A universal modular actor formalism for artificial intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Standford, CA, USA, 20–23 August 1973; Morgan Kaufmann Publishers Inc.: Burlington, MA, USA, 1973; pp. 235–245.
23. Agha, G.A. *Actors: A Model of Concurrent Computation in Distributed Systems*; MIT AI-TR; MIT Press: Cambridge, MA, USA, 1986; Volume 844.
24. Young, G. CQRS Documents. 2010. Available online: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf (accessed on 2 February 2024).
25. Meyer, B. *Eiffel: The Language*; Prentice-Hall, Inc.: Hoboken, NJ, USA, 1992. [CrossRef]
26. Fowler, M. Event Sourcing. 2005. Available online: https://martinfowler.com/eaaDev/EventSourcing.html (accessed on 2 February 2024).
27. Abadi, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* **2012**, *45*, 37–42. [CrossRef]
28. Camilleri, C.; Vella, J.G.; Nezval, V. ThespisTRX: Initial Results for Causally-Consistent Read Transactions. In Proceedings of the Information Systems and Management Science, Valletta, Malta, 22–23 February 2018.

29. Camilleri, C.; Vella, J.G.; Nezval, V. ThespisDIIP: Distributed Integrity Invariant Preservation. In *Proceedings of the Database and Expert Systems Applications*; Elloumi, M., Granitzer, M., Hameurlain, A., Seifert, C., Stein, B., Tjoa, A.M., Wagner, R., Eds.; Springer: Cham, Switzerland, 2018; pp. 21–37.
30. Barbará-Millá, D.; Garcia-Molina, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB J.-Int. J. Very Large Data Bases* **1994**, *3*, 325–353. [CrossRef]
31. Neumann, T.; Mühlbauer, T.; Kemper, A. Fast serializable multi-version concurrency control for main-memory database systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Australia, 31 May–4 June 2015; pp. 677–689.
32. Bernstein, P.A.; Hadzilacos, V.; Goodman, N. *Concurrency Control and Recovery in Database Systems*; Addison-Wesley: Reading, MA, USA, 1987; Volume 370.
33. Camilleri, C.; Vella, J.G.; Nezval, V. Actor model frameworks: An empirical performance analysis. In *Proceedings of the International Conference on Information Systems and Management Science*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 461–472.
34. Himschoot, P. Efficient Communication with gRPC. In *Microsoft Blazor*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 465–482.
35. Liu, S.; Benson, T.A.; Reiter, M.K. Efficient and safe network updates with suffix causal consistency. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, 25–28 March 2019; pp. 1–15.
36. Roohitavaf, M.; Ahn, J.S.; Kang, W.H.; Ren, K.; Zhang, G.; Ben-Romdhane, S.; Kulkarni, S.S. Session guarantees with raft and hybrid logical clocks. In Proceedings of the 20th International Conference on Distributed Computing and Networking, Bangalore, India, 4–7 January 2019; pp. 100–109.
37. Paksula, M. Introduction to store data in Redis, a persistent and fast key-value database. In *Proceedings of the AMICT 2010–2011 Advances in Methods of Information and Communication Technology*; University of Helsinki: Helsinki, Finland, 2010; p. 39.
38. Bernstein, P.A.; Burckhardt, S.; Bykov, S.; Crooks, N.; Faleiro, J.M.; Kliot, G.; Kumbhare, A.; Rahman, M.R.; Shah, V.; Szekeres, A.; et al. Geo-distribution of actor-based services. *Proc. ACM Program. Lang.* **2017**, *1*, 1–26. [CrossRef]
39. Camilleri, C.; Vella, J.G.; Nezval, V. ThespisTRX: Causally-Consistent Read Transactions. *Int. J. Inf. Technol. Web Eng. (IJITWE)* **2020**, *15*, 1–16. [CrossRef]
40. Dey, A.; Fekete, A.; Nambiar, R.; Röhm, U. YCSB+ T: Benchmarking web-scale transactional databases. In Proceedings of the 2014 IEEE 30th International Conference on Data Engineering Workshops, Chicago, IL, USA, 31 March–4 April 2014; pp. 223–230.
41. Chen, S.; Ailamaki, A.; Athanassoulis, M.; Gibbons, P.B.; Johnson, R.; Pandis, I.; Stoica, R. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Rec.* **2011**, *39*, 5–10. [CrossRef]
42. Yang, Y.; Zhu, J. Write skew and zipf distribution: Evidence and implications. *ACM Trans. Storage (TOS)* **2016**, *12*, 1–19. [CrossRef]
43. Sidhanta, S.; Mukhopadhyay, S.; Golab, W. DYN-YCSB: Benchmarking adaptive frameworks. In Proceedings of the 2019 IEEE World Congress on Services (SERVICES), Milan, Italy, 8–13 July 2019; Volume 2642, pp. 392–393.
44. Yamada, H.; Nemoto, J. Scalar DL: Scalable and practical Byzantine fault detection for transactional database systems. In Proceedings of the VLDB Endowment, Sydney, Australia, 5–9 September 2022; VLDB Endowment Inc.: Sydney, Australia, 2022; Volume 15, pp. 1324–1336.
45. Rossi, F.; Falvo, S.; Cardellini, V. GOFS: Geo-distributed scheduling in OpenFaaS. In Proceedings of the 2021 IEEE Symposium on Computers and Communications (ISCC), Athens, Greece, 5–8 September 2021; pp. 1–6.
46. Kshemkalyani, A.D.; Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*; Cambridge University Press: Cambridge, UK, 2011.
47. Bernstein, P.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J. Orleans: Distributed Virtual Actors for Programmability and Scalability. Available online: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/ (accessed on 2 February 2024).
48. Camilleri, C.; Vella, J.G.; Nezval, V. Thespis: Causally-consistent OLTP. In Proceedings of the 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), Sofia, Bulgaria, 2–5 September 2021; pp. 261–269.
49. Akkoorath, D.D.; Tomsic, A.Z.; Bravo, M.; Li, Z.; Crain, T.; Bieniusa, A.; Preguiça, N.; Shapiro, M. Cure: Strong semantics meets high availability and low latency. In Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 27–30 June 2016; pp. 405–414.