DistWalk: a Distributed Workload Emulator

Remo Andreoli
Sant'Anna School of Advanded Studies, Pisa, Italy
remo.andreoli@santannapisa.it

Tommaso Cucinotta
Sant'Anna School of Advanded Studies, Pisa, Italy
tommaso.cucinotta@santannapisa.it

***Abstract*—This paper introduces DistWalk, a flexible, distributed, scalable, and open-source toolkit designed to emulate compute, network, and storage workloads across a networked infrastructure, and measure the resulting end-to-end latency. DistWalk provides fine-grained control over the workload behavior, which consists of a graph-like sequence of operations spanning multiple servers, It supports several communication protocols and traffic patterns, and enables the customization of several factors, such as the duration and parallelism of compute-intensive operations, and the I/O data access and synchronization mode, among others. The proposed toolkit may be used to experiment with a variety of deployment models, from bare-metal to virtualized or containerized environments, e.g., using Cloud/Edge infrastructures, OpenStack, Kubernetes, or other orchestrators, allowing for experimental comparisons of the achievable latency across a wide range of system-level configurations.**

***Index Terms*—Cloud Computing, Distributed System, Benchmarking.**

I. Introduction

Nowadays, many modern IT services are delivered as distributed software deployed across clusters or large-scale cloud infrastructures, offering exceptional scalability, reliability, and flexibility to handle diverse workload conditions. When designing a distributed or Cloud-native software service, it is useful to assess the expected achievable performance early in the design stage, when the implementation may not be (completely) available yet. This is particularly critical for interactive and time-sensitive applications, where the experienced end-to-end latency is heavily influenced by design choices such as the parallelism degree within the application, the placement of its components over a Cloud/Edge infrastructure, the load-balancing strategy for distributing the workload, and the tuning and configuration of the machines, software stack, and kernel, or other system-level parameters [1]. This requires a deep understanding of the bottommost layers of the software stack, along with knowledge of the appropriate low-level tunables to fully exploit the potential of the underlying infrastructure.

It is commonplace to resort to experimental approaches to estimate the performance implications of various design, deployment and tuning options for upcoming software. This may be conveniently performed by using computational, networking and disk I/O *microbenchmarks*. However, these tools usually test a single component or aspect of a distributed system (e.g., the computational, disk I/O, or networking performance of a single component, link, or communication path). As a result, they provide limited insight into how a distributed application with geo-dislocated components might perform under certain conditions. As discussed in Section II, we are still lacking a comprehensive tool for Cloud developers to: i) explore several low-level optimization opportunities in the design space and deployment environment; and ii) verify experimentally the impact of these design choices, and associated tunables, on the end-to-end performance that might be expected by the final software.

A. Contributions

This paper presents DistWalk, a versatile, easy-to-use, and open-source toolkit that allows for fine-grained emulation of various workloads, traffic, and access patterns. The toolkit is publicly available on GitHub[1], under a GPLv3 license. DistWalk focuses on assessing timing behaviors, with an emphasis on end-to-end latency implications within a geo-dislocated infrastructure. With a command-line interface, DistWalk can deploy, on real systems (i.e., physical hosts, virtual machines, containers, clusters, etc.), a workload specified in terms of a graph-like sequence of data-driven operations spanning across multiple nodes, triggered by a wide range of available traffic patterns. The individual components of DistWalk can exercise on the underlying system CPU-intensive operations with diverse parallelism, multi-threading, and scheduling configurations, as well as I/O-intensive operations with different data access and synchronization modes, and communicate with each other using different protocols and payload sizes. The tool has been developed in the C programming language on the Linux Operating System (OS), exploiting mostly POSIX APIs, alongside some Linux-specific APIs to enhance performance and scalability when possible (as typical in nowadays industrial-grade Cloud services).

In Section IV, we demonstrate how DistWalk can be used to explore various aspects of system performance and benchmarking from the perspective of a user who has complete control over both the underlying hardware and the low-level details of the software to be deployed.

II. Related Works

Benchmarking is the preferred method for assessing a system performance, as witnessed by the plethora of contributions in the field. This approach uses real systems to deploy real or synthetic workloads. This section provides a brief overview of related contributions on benchmarking for large-scale distributed systems, comparing them to our tool.

[1] https://github.com/tomcucinotta/distwalk

An interesting open-source tool is `stress-ng`, designed to stress various components of a computing system in several possible ways. It can stress computational capabilities of the physical CPU, cache, and the whole memory hierarchy, the I/O subsystem, various features of the OS kernel, from page swapping to inter-process communications through pipes or local networking, to high-frequency timers, and others. Due to its nature, `stress-ng` aims at saturating the physical resources of a *single* system. In contrast, our proposal emulates certain distributed workload conditions in a typical production environment, and focuses on measuring end-to-end latencies. These workloads are generally far from the resource saturation use case, and include scenarios such as web-based and service-oriented architectures, database or NoSQL data store systems, interactive distributed applications, and more.

TailBench [12] is a benchmarking suite for latency-critical applications. It aggregates a series of representative applications to harness tail-latency behaviors, including a search engine, an in-memory key-value store, and a speech recognition system, to name a few. TailBench employs a traffic shaper, a request queue, and statistic collector to control and aggregate the timing characteristics of a request stream. However, the focus is on single-tier applications, making it less relevant for nowadays Cloud services.

Similarly, DeahStarBench [8] is a suite of demo applications used to study hardware and software performance implications of microservice architectures. It provides six use cases that represent typical Cloud services: a social network, a media service, an e-commerce website, a secure banking system, and an IoT service for coordination control of drone swarms. The suite is built using popular open-source applications, such as NGINX for load-balancing, MongoDB for persistence, and HTTP requests for communications.

ServerlessBench [22] is an open-source benchmark for studying the performance of several commercial and open-source serverless platforms. The focus is on communication performance, start-up latency, resource efficiency, and performance isolation of co-located serverless functions.

μBench [6] is an open-source tool capable of deploying, via Kubernetes and Docker containers, a set of microservices, according to a configurable execution model and service mesh describing how these microservices call each other to serve HTTP requests. It includes several Python components that can: generate the Kubernetes files needed to configure the desired microservice mesh and handle the communications among them; configure microservices to impose on the underlying hardware the desired resource-intensive workload, combining a "portfolio" of Python functions, that can be activated based on stochastic distributions or trace files. At a glance, μBench seems to have similar objectives to our DistWalk proposal. However, its focus is clearly on Kubernetes deployments, rather than on the efficiency of execution of the deployed microservices. These, being based on the Python programming language, fall short in terms of applicability in low-latency or high-throughput scenarios. On the other hand, DistWalk is written in C and it is designed to provide a flexible, yet efficient tool to experiment with low-latency and performance-critical scenarios, thanks to its capability to customize a number of low-level characteristics of the software stack, as explained in Section III.

Yahoo! Cloud Serving Benchmark (YCSB [5]) is a well-known toolkit that has been extensively used to evaluate and compare the performance of NoSQL data stores in the literature [2], [7], [20]. It comprises a set of workloads that can be customized in terms of proportion of read, update, and insert operations, request distribution across the key space, and number of pre-inserted records, to name a few. YCSB has a quite focused scope, being a tool designed specifically to submit traffic to a number of different data stores.

Fogify [21] is an open-source emulation framework designed for the deployment of microservices in Docker containers, mimicking the conditions of a Cloud/Fog heterogeneous infrastructure. It exploits Docker Swarm to deploy containers in a local distributed infrastructure, providing them with restricted memory, processing and networking capabilities, leveraging on the ability of the Linux kernel to partition the available RAM and configure CPU shares for individual containers through `cgroups`, and configure traffic shapers exploiting `qdisc`. This way, a powerful multi-core server may be configured through Fogify to emulate a geo-distributed set-up having both powerful Cloud nodes, and restricted-resources Fog nodes. Fogify might be considered as a possible complement to our workload emulator, i.e., one might deploy workloads emulated with DistWalk across a Fogify infrastructure to evaluate and compare experimentally a number of Cloud/Fog heterogeneous and geo-dislocated deployments.

ScaleBench [11] is an interesting tool aiming at assessing experimentally the horizontal scalability limits of nowadays Cloud infrastructures. However, the tool has quite a focused objective and scope of applicability: it is implemented as a Java benchmark with a fixed, albeit elastic, topology, exploiting a RabbitMQ instance for workload distribution across workers, and a MySQL instance for gathering the results. Instead, our DistWalk is a more general tool that can be used for modeling distributed applications with flexible service graph topologies, and gathering their expected performance from real deployment scenarios and configurations.

In conclusion, most related works address only specific, high-level aspects of system performance: some target the performance implications of particular software architectures, such as microservice-based applications [6], [8] and programming models [9]; others evaluate the transaction processing power of database systems [5] or target the evolution of the Cloud Computing paradigm, such as serverless platforms [22]. None of them tackle the benchmarking challenge in distributed systems from a low-level perspective that encompasses both the infrastructure and the software stack.

Differently from these approaches, DistWalk is a benchmark aiming at emulating the non-functional part of a prospective application design. It does not include a pre-configured collection of software services, as done in the TailBench and DeathStarBench projects discussed above. Instead, in

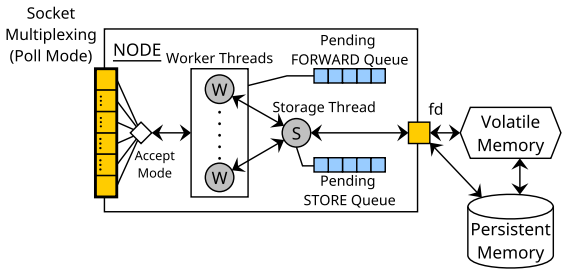


Fig. 1: Architecture of a DistWalk node: each component offers a high degree of configurability.

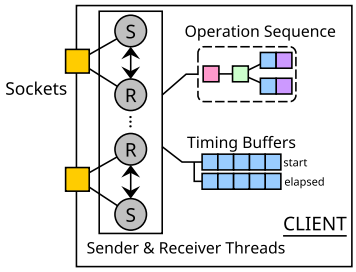


Fig. 2: Architecture of a DistWalk client.

DistWalk, the user has full control over the components that are deployed in a distributed infrastructure, the resources they consume, the topology of the interactions among them, and the pattern of the traffic hitting the system. Therefore, it allows to study the latency implications of large-scale deployments from a low-level perspective.

III. DistWalk

DistWalk is an open-source distributed workload emulator, capable of measuring the end-to-end latency impact of given resource consumption distributed workload patterns, providing a wide range of low-level options to mimic common designs and operating conditions. Its main features are: i) a highly scalable architecture featuring asynchronous, socket-based communications; ii) low-level tunables for fine-grained control over the workload behaviors (i.e., data access pattern, compute processing duration, etc.) and server configuration details (i.e., communication model, data synchronization, etc); iii) ready-to-use executables that can be integrated, in part or its entirety, within or alongside other third-party toolkits; and iv) a platform-agnostic tool that can be deployed on-premise, on small-scale clusters, or on cloud platforms, using any orchestrator (OpenStack, Kubernetes, or others).

A. Architecture and Design

DistWalk follows a client-server architecture. The simplest scenario involves a *client* that defines a sequence of operations, and a *server* (referred to as a *node* in DistWalk) that executes them. The sequence is encapsulated in a network packet and sent to the node as a request multiple times, following a traffic pattern defined by an initial request rate and a series of rate ramp-ups. For each request, the server performs the user-defined operations, then it sends a response to the client. The client awaits for responses to all requests, optionally up to a maximum timeout, before displaying the measured round-trip latencies, a.k.a., response time, for each request. More interesting scenarios involve multiple nodes and a client submitting workloads structured as graph-alike topology that span across multiple nodes. For each request, the server performs its associated operations, and then forwards the remaining payload to the next server(s). Once the payload is fully exhausted, a response is routed back to the client, retracing the communication path in reverse. Finally, the use of multiple clients enables submitting multiple workloads concurrently on top of the same underlying infrastructure.

The client and node components are multi-threaded C programs with no dependencies on external libraries. This design ensures that DistWalk is a high-performance, highly scalable, plug-and-play toolkit. Both components are controlled via the command line using a versatile syntax. The DistWalk node is a networked server designed to be reused across several client workload runs. Figure 1 depicts the major features of the node architecture: i) the ability to choose between different connection accept policies (called "accept modes" in DistWalk) and multiplexing mechanisms (called "poll modes"); ii) a pool of threads to handle the user-defined operations, both synchronously and asynchronously; and iii) a dedicated thread for interacting with persistent memory. Section III-B provides an in-depth description of each customization aspect within the node. Figure 2 illustrates the client architecture, featuring pairs of sender and receiver threads to interact with the nodes. This two-thread architecture has the advantage of keeping the request submission period/rate as stable and precise as possible. Further details on the customization aspects of the client are provided in Section III-C.

A DistWalk node supports the following operations:
`COMPUTE`: Emulate a CPU-bound operation by keeping the serving thread busy on a loop for a user-specified amount of time, optionally sampled from a probability distribution, such as Uniform, Exponential and Gamma, among others.
`LOAD/STORE`: Emulate an I/O-bound operation by loading from or storing to a storage device using the `read()`, `write()` low-level primitives. The user defines the amount of random bytes to be read or written, with an optional offset. Both parameters can be optionally sampled from probability distributions. The ability to specify an offset allows for emulating either sequential or random access I/O request patterns using the `lseek()` primitive. For the store operation, the client can also mimic common syncing behaviors with the `fsync()` primitive: per-request synchronization, no synchronization, or periodic synchronization. The latter option is typically employed by NoSQL data stores such as MongoDB.
`FORWARD`: Emulate a Remote Procedure Call (RPC) to a different node by sending a message with a user-specified payload size, and expecting a response back, with another user-

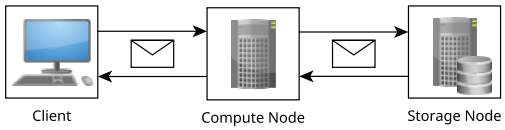


Fig. 3: A simple chain topology emulated using a DistWalk client and two nodes.

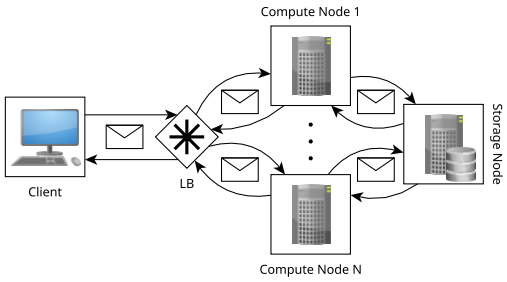


Fig. 4: A load-balanced deployment consisting of N+1 DistWalk nodes. It may be used to emulate a web server.

specified payload size. This is DistWalk's essential feature that allows emulating a distributed workload. Moreover, a client can request a node to forward (optionally different) sequences of commands to *multiple* nodes, and wait for all or a subset of responses, before moving on with its "local" sequence of operations. This can be used to emulate the behavior of *quorum-based* consensus protocols, such as PAXOS [14] and RAFT [18], which are the building blocks for replication in NoSQL data stores to avoid inconsistencies in the data copies [10], [23]. As for the previous operations, payload sizes can optionally be sampled from a probability distribution.

Client requests may contain an arbitrary sequence of these operations, with varying parameters and sequence length, in order to mimic a realistic distributed workload. A DistWalk deployment is highly customizable on both the node and client sides via command line or trace files. In summary, the user has full control over many system-level aspects, such as the location of the components, the sequence of operations to be performed, the payload size of the communications, and so on. With an intuitive client syntax, DistWalk can emulate simple service chains as in Figure 3, or more complex scenarios mimicking the architecture of today's web servers (such as Figure 4). The next subsections provide a description of the various parameters used to fine-tune low-level details spanning multiple domains (i.e., networking, computation, and storage).

B. Node Customization

The DistWalk node is the core contribution, encapsulating all the functionality required to emulate compute, storage, and network activities within a physical or virtual host. The node computational behavior is configured specifying:

- the number of *worker threads*;
- the CPU scheduling policy for worker threads;
- the thread-to-core pinning options.

The use of multiple worker threads enables the analysis of the multi-core capabilities of the underlying host. A worker thread manages communication with one or more clients and executes the requested operations. Storage-related activities are dispatched to a dedicated *storage thread* to ensure that worker threads are not blocked on disk I/O, and that disk operations with large payloads are serialized so to make I/O more efficient. Client connections are monitored via a `select()`, `poll()` or `epoll()` loop, depending on the configured poll mode. DistWalk exposes a tunable to customize the CPU scheduling policy for worker threads: `nice` levels for the default (`SCHED_OTHER`) scheduler, and a priority for the real-time scheduling policies (`SCHED_RR` or `SCHED_FIFO`), all available on POSIX-compliant systems. DistWalk also supports the reservation parameters of the Linux-specific `SCHED_DEADLINE` [16] scheduler. Finally, the core pinning options enable the measurement of the performance impact of system partitioning and NUMA-awareness, which is a critical consideration in large-scale Cloud and HPC environments.

The storage behavior is customized specifying:

- the path of the storage file for reading and writing data;
- the maximum size of the storage file;
- the optional use of direct disk access;
- the optional periodic interval, in milliseconds, for synchronizing the written data to persistent storage.

The storage file location determines which device to use, allowing to investigate performance of different storage solutions. The maximum size causes the file pointer to wrap around and `lseek()` back to the beginning, if a write operation exceeds this limit. This is useful to analyze performance implications of rotational devices, where seek times and rotational latency can significantly impact I/O performance.

Direct disk access, which is enabled by opening a file with the `O_DIRECT` flag, bypasses the read/write OS caches. This ensures that I/O operations are done directly to/from user-space buffers without copies. DistWalk handles the block-aligning constraints that come with direct I/O operations. However, notice that there is no guarantee that written data will be stored on non-volatile memory. It is the data synchronization mode specified by the client message that requests a flush of all changes into persistent storage. In practice, this is performed by the storage thread using the `fsync()` system call.

Finally, the node networking behavior is tuned specifying:

- the bind address and port, and communication protocol;
- the maximum number of pending TCP connections, which is configured through the `listen()` system call;
- the accept mode for incoming connections;
- the poll mode for I/O monitoring;
- the optional use of the `TCP_NODELAY` socket option.

DistWalk supports TCP- and UDP-based communications. For TCP, incoming connections can be accepted in 3 ways: the `parent` accept mode has a single thread accepting connections using the `accept()` syscall, that are handed over to one of the worker threads. Albeit being quite popular, this choice intro-

duces additional context switches on connection establishment. These can be avoided using the `shared` and `child` accept modes. The former simply exploits the ability of multiple threads to accept connections directly from the same bound socket. This shares the same queue of incoming connections in the kernel, which might have scalability drawbacks on large multi-cores. The `child` accept mode mitigates this issue by letting worker nodes independently bind, listen, and accept connections from the same IP and port, using dedicated bind sockets, setting the `REUSE_PORT` socket option available on Linux. This results in the worker threads using separate incoming connection queues, which provides enhanced scalability on large multi-core servers. Finally, `TCP_NODELAY` allows to disable the Nagle's algorithm [17] (deferring the transmission of small packets), that might hurt low-latency, interactive applications due to the introduced extra-delay.

In summary, a DistWalk node enables the exploration of a multitude of configuration parameters for a server application, from a low-level perspective.

C. Client Customization

The DistWalk client contains the parsing logic responsible for formatting messages, and the logic for submitting requests at a user-defined period/rate. It also handles receiving responses and measuring the end-to-end latency. The client offers numerous auxiliary parameters to further customize the operations that constitute the emulated workload. General parameters affect the whole client set-up, and are normally specified just once on the command line. Operation-specific parameters define and fine-tune individual operations, so they can be specified multiple times.

Examples of general parameters are the following:

- number of sender-receiver thread pairs used for submitting requests;
- number of per-pair requests to submit to the node(s);
- number of per-pair sessions to be established;
- initial request rate/period, and subsequent ramp-ups;
- disable the Nagle's algorithm [17], as for the node;
- enable spin-waiting in the client sender thread.

The emulated client workload consists of a number of sender-receiver thread pairs submitting a specified number of requests to the node(s), with the specified inter-arrival period or rate. This allows a single client process to use multiple communication channels to the node(s), where different thread pairs can act either synchronously, causing bursts, or in a staggered way. Each client sender thread can further divide its specified number of requests across multiple consecutive sessions, by closing the connection to the node and re-establishing it during the run. This feature is useful for analyzing load-balancing techniques when using the TCP protocol, at different levels of abstraction: i) node-level, by leveraging the "parent" accept mode within the DistWalk node, described in Section III-B; and ii) application-level, by placing a load balancer in front of a pool of DistWalk nodes.

The request rate parameter is one of the most versatile features of DistWalk in order to define workload patterns and capture the dynamics of a real-world scenario. Given a starting rate, a DistWalk client can precisely characterize subsequent rate ramp-ups in terms of the number of ramp-up steps, step duration, and delta increment/decrement at each step. These ramp-up details can be loaded from a trace file. Notice that this option allows the generation of mixed ramp-ups and ramp-downs to precisely characterize the desired traffic pattern.

By default, client sender threads sleep between a request and the next one to respect precisely the current rate or period, using a `clock_nanosleep()` with an absolute time. Alternatively, a sender thread can also be configured to busy-wait between requests: this gives the ability to generate higher throughput values, saving context switch overheads, at the cost of having each sender thread occupying a full CPU.

The client supports these operation-specific options:

- Duration of a `COMPUTE` operation, in microseconds;
- Number of bytes to be read or written for `LOAD` and `STORE` operations, respectively;
- The storage offset for reading or writing data;
- Data synchronization policy for a write operation;
- Packet payload size for the node reply;
- Packet payload size for the send request.

The storage offset parameter is used to emulate random or sequential access patterns, while the synchronization policy determines whether the node should immediately flush the written data to disk. When combined, these two parameters enable the emulation of various access patterns and allow for the study of different performance implications related to data synchronization. The send and reply payload size parameters enable to stress the network.

Finally, the DistWalk client features a special `PSKIP` operation for the probabilistic execution of arbitrary segments of a DistWalk command. In particular, it is used to instruct the serving node to skip certain operations within the sequence, with a specified probability: for instance, this may be useful for emulating different disk read/write ratios, as in YCSB [5]; Another example is that of a web server that looks up within a cache of recent query results before actually querying a database system (if the cache misses). This can be modeled in DistWalk as the probability of skipping a FORWARD to the node acting as the database system, using the envisioned cache hit rate as skip probability.

Many of the arguments to the above-described options (e.g., inter-request period, request or reply size, compute duration, storage transfer size, etc...) can be specified both as fixed values, or as samples drawn from a probability distribution, as a term drawn from an arithmetic or geometric sequence, or even as a value read from a specified column of a CSV trace file. The available probability distributions include uniform, exponential, Gaussian, and Gamma distributions, with customizable parameters. These have been implemented in DistWalk using exclusively the `drand48_r()` function. The argument specification can include an optional min and/or max constraint, causing a rescaling of the distribution within the specified range. The ability to explicitly seed the random number generator and to read samples from trace files ensures

reproducible runs. The command-line interface allows for automated script-based deployments of several clients, and aggregation of their outputs.

In summary, the DistWalk client enables the exploration of various parameters to closely mimic a realistic workload and its dynamic behavior.

IV. Experimental Evaluation of the Performance Impact of Design Choices

This section shows how DistWalk can be used to study the latency implications of various choices in a distributed system design. We do not aim to provide a complete analysis, evaluation, or comparison of specific system configurations, nor to advocate a particular optimization choice, such as which load balancing technique, resource management policy, or disk access strategy is best suited for a given workload. For simplicity, we will focus on the computing and storage aspects, while keeping the network component at default settings and utilizing basic TCP communications with `SO_NODELAY`. All the artifacts (i.e., Ansible playbooks, shell scripts, Docker images and Kubernetes deployment files) used to produce the results are available within the main GitHub repository of DistWalk, and can be used to evaluate your infrastructure in a plug-and-play manner. Moreover, the results presented in this paper can be partially reproduced using the companion repository available on Zenodo[2].

The first subsections focus on simple experiments showing how to evaluate low-level performance aspects of a single, server-grade computer system. Next, the full potential of DistWalk is shown on a distributed environment composed of consumer-grade personal computers, exploring common aspects of a distributed system, such as load balancing techniques. Lastly, we show how it is possible to emulate realistic behaviors, showcasing an emulation of the popular Apache HTTP web server. Each machine has turbo-boosting, deep idle states disabled, and CPU frequency locked to the maximum value recommended by the vendor. Unless stated otherwise, each DistWalk component is deployed on a dedicated physical machine. Most of the results are presented as boxplots with whiskers set to the 1st and 99th percentile, and a red "×" marker denoting the average latency.

A. Single System: Ramp-up Testing

In this subsection, we demonstrate how to stress test a single node by using the ramp-up parameters of DistWalk. The request rate parameter "-r" allows to specify a fixed value, or an arithmetic sequence "aseq" or a geometric sequence "gseq", which are defined by a starting rate "min", a finishing rate "max", and an incremental (or multiplicative) increase (or decrease) "step"; alternatively, a probability distribution is also supported, to be used to draw rate samples from.

The goal is to determine the system's robustness, under different configurations, beyond the limits of normal operating conditions. Only a few benchmarking tools offer traffic ramp-up configuration parameters as granular as those of DistWalk.

[2]See: https://zenodo.org/records/14923780

Figure 5 depicts several ramp-ups generated from a single-core client, using an arithmetic sequence, with the following command line:

```
--to addr:port -C 10ms -n 1000
               -r aseq:min=2,max=100,<step> --rss 1
```

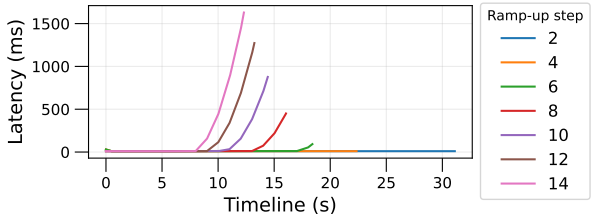


Fig. 5: Timeline plot of ramp-up loads generated using arithmetic sequences.

which requests a `COMPUTE` operation ("-C") of duration $10ms$, repeated 1000 times. The "addr:port" parameter value refers to the IP address and port to which the targeted DistWalk node is bound, and "–rss" defines the duration of a rate step, in seconds. The experiment highlights the point-in-time when the node reaches its computational bottleneck, if any, for several rate step variations.

B. Single System: HDD vs SSD

In this subsection, we analyze the storage performance of a single node, evaluating the latency of random writes performed on a hard disk drive (HDD) and a solid-state drive (SSD). The workload consists of a single `STORE` operation of 10 kilobytes, repeated 500 times. The advantage of SSDs over HDDs from a performance standpoint is well known, due to the absence of mechanical spinning disks, which allows for significantly faster data access and transfer speed. Each write operation accesses the storage device at random locations spanning over 32 gigabytes, sampled from a uniform distribution. On the command line, the DistWalk client is launched as follows (for brevity, we omitted the node address and port):

```
--store-offset unif:min=0,max=32g
 -S 10k,<sync|nosync|psync> -n 500
```

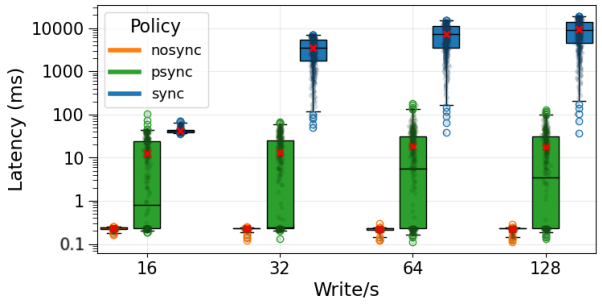
where "sync", "nosync" and "psync" are the three available data store synchronization policies (as described in Section III-A). Figure 6 showcases a series of experiments using different store rates. The first group of boxplots (orange color), in both Figure 6a and Figure 6b, illustrates the write latency when no data synchronization is manually enforced, meaning that a write operation is immediately acknowledged. As expected, there is no noticeable latency difference between SSDs and HDDs, however, a real application with this design would be unable to guarantee that the data is flushed to the storage device in the event of a system crash or reboot.

The second group of boxplots (green color) illustrates the write latency with periodic data synchronization set to $100ms$ (default value for MongoDB). In Figure 6a, it is evident that a subset of write operations are delayed by the data

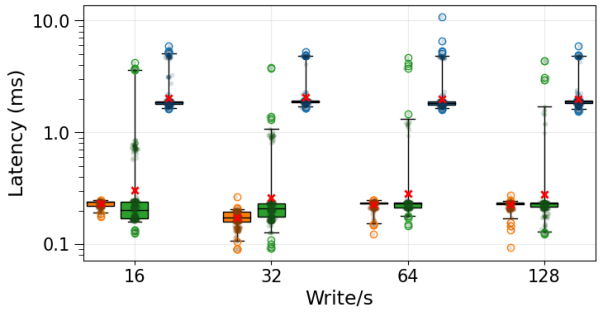
synchronization procedure, as indicated by the two hot spots at the 25th and 75th percentiles, an average latency of $15ms$, and $20ms$ of standard deviation. This behavior also affects SSD performance, though as outliers.

The last group of boxplots (blue color) shows the latency when `fsync()`ing each write operation to the disk before acknowledging the client, as typical in high-reliability data replication scenarios. The HDD shows a significant increase in latency, especially at higher rates. The standard deviation is significantly higher, reaching as high as $5s$, due to the continuous random offset repositioning causing the requests to pile-up. As expected, the write latency is affected by the synchronization choice much more on HDDs than on SSDs.

Simple experiments like these can be useful to determine how the choice of a storage device type, as well as the data access and synchronization pattern, are expected to affect the end-to-end latency of a distributed service being developed.



(a) HDD



(b) SSD

Fig. 6: Response times (Y axis) obtained with random writes, at various write request rates (X axis).

C. Single System: Impact of Thread-to-Core Pinning

Core pinning is one of the popular design choices that are available when customizing the deployment of a distributed multi-threaded application. Leaving the threads free to migrate is certainly the easiest setup, as it does not require specific interactions with the OS nor special configurations. However, it may result in the OS deciding to migrate the threads, impacting the resulting end-to-end latency. This may be avoided by pinning individual threads to physical cores, via special interactions with the OS, or some special directives in the configuration of a service, to fix the affinity mask of each thread set to a specific physical core.

This subsection shows the benefits of core pinning using a DistWalk node with 16 threads, serving 160K requests split among 100 sessions, fired in batches of 16 requests every $10ms$ from a 16-threads client from the same 1Gbps LAN. Both client and server machines are dual-socket, equipped with 2 Xeon(R) CPU E5-2640 v4 running at 2.40GHz, 25MB of L3 cache and 125GB of RAM. Each system has 20 physical cores, of which only 16 cores are utilized.

The DistWalk command-line instructions are:

```
node:   --nt=16 -c 2-18 -a parent
client: --nt=16 -C 1ms -p 10k -n 10000 --ns 100
```

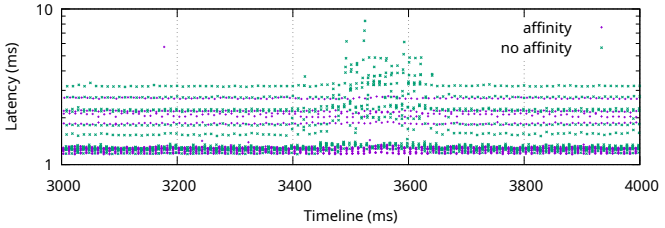


Fig. 7: Excerpt of the response times (Y axis) obtained with worker threads pinned to physical cores (purple dots) vs left free to migrate (green dots).

Figure 7 shows an excerpt of the data from the experiment, where we can see that the latencies experienced with core pinning (purple dots) are more stable and do not suffer from transients with higher values as in the case without core pinning (green dots). These happen quite often throughout the trace, particularly on session re-establishment, when a client thread closes the current session, and establishes a new TCP/IP connection, causing additional migrations of the worker threads. This results in an overall latency with an average of $1.5ms$ and a standard deviation of $2.2ms$, achieved with core pinning, versus a latency with an average of $1.9ms$ and a deviation of $4.2ms$, achieved without core pinning. The responsibility of thread migrations on these results is confirmed by reading the individual per-thread migration counters from `/proc/.../sched` at the end of the experiment, reporting 15 migrations only for the pinned set-up (1 migration per thread, needed when setting the affinity of the thread), versus 4601 migrations for the default configuration.

D. Single System: Impact of Poll/Accept Modes & DistWalk Overheads

In a variant of the above experiment run on the same machines, we launched the DistWalk node with 16 threads and core pinning enabled, in either of the 3 available accept modes: parent, shared and child (see Section III). The client has also been configured with 16 threads, each submitting 100 single-request sessions with a processing time of $1ms$, every $10ms$. We obtained $17ms \pm 19ms$ of end-to-end latency in child mode, slightly worse than $18ms \pm 24ms$ in shared mode, and much worse than $34ms \pm 22ms$ in parent mode.

This highlights the inherent limitations of the commonly used parent approach, with many threads.

To assess the impact on latency of different poll modes, we conducted another experiment using a node with a single worker thread. A client submits requests with a processing time of 0 ms, using a varying number of threads (from 1 to 100), each operating at a fixed request rate of 100:

```
node:   --nt=1 -c 2
client: --nt=<nthread> -C 0ms -r 100 -n 1k
```

The node is configured with varying poll and accept modes (excluding the "shared" accept mode, which works similarly to the "child" mode in this scenario). Figure 8 shows that the epoll modes (cyan data points) bring stable performance, whilst the more portable select and poll modes suffer from increased latencies as the number of connected sessions grows.

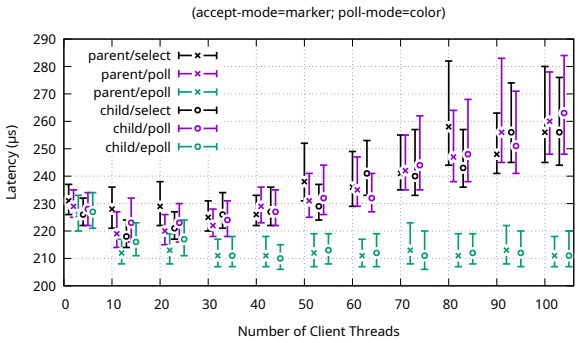


Fig. 8: Response time median (marker on Y axis) and 1st/3rd quartiles (lines) obtained with different combinations of poll and accept modes, at varying number of client threads (X axis).

The results from the single-threaded client experiment (first cluster of data points) can be used to infer the overhead introduced by DistWalk in the emulated distributed scenarios. By subtracting the ping latency of about $116\mu s$ from the latency of $230 \pm 10\mu s$, we obtain a per-request overhead of $114 \pm 10\mu s$.

E. Distributed System: Virtualization Overhead

Studying the overhead introduced by virtualization technologies, in terms of response latency is of utmost importance in emerging use-cases such as real-time virtualized applications. This subsection showcases how DistWalk can be used to emulate a workload across different orchestration platforms to analyze the virtualization overheads. These experiments use identical x86-64 machines, equipped with a i5-4590S CPU running at 3.00 GHz, 6MB L3 cache, and 16GB of RAM, and interconnected via a wifi router.

For the sake of simplicity, the experiments use two nodes and one client; the latter is not virtualized to emulate the classic ingress traffic of virtualized infrastructures. The emulated workload consists of a `COMPUTE` operation on the first node, a `FORWARD`, and `COMPUTE` operation on the second one. The sequence is then executed 100 times with a processing duration of $100ms$ (on both nodes) and a rate of 10 requests/second to avoid congesting the deployment. The command line instructions for the client are the following:

```
--to node1 -C 100ms -F node2 -C 100ms -n 100 -r 10
```

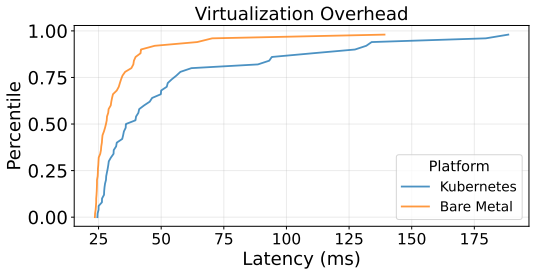


Fig. 9: Empirical CDF plots of the end-to-end response times obtained with a bare-metal versus Kubernetes deployment.

Figure 9 shows a plot of the Cumulative Density Function (CDF) of the latency comparing a bare-metal with a Kubernetes deployment. As expected, the Kubernetes latency distribution maintains a similar shape but it is shifted to the right and it has a higher variance, indicating degraded performance (the average latency is increased by $19ms$, and the 99th percentile by $78ms$).

F. Distributed System: Data Redundancy

Distributed databases are the most popular services offered by Cloud providers. A key property of these systems is data replication, which involves copying data across multiple sites to eliminate single points of failure and increase availability. Redundancy introduces the well-known challenges of maintaining consistency across replicas and the trade-off between consistency and availability of the CAP theorem [3]. Full consistency introduces high latencies, as it requires waiting for all nodes to replicate a data change. To address this, modern distributed databases offer various consistency models [19], generally defined by the number of replicas required to acknowledge a write operation.

In this subsection, we use DistWalk to study the latency implication of these consistency options. Figure 10 illustrates the experiment outcome over a deployment consisting of 4 nodes and a client. One of the nodes acts as a "gateway", spreading and gathering the forwarded requests on behalf of the user. The remaining ones perform the actual workload. This is the same sort of replication deployment used in MongoDB. The operation sequence consists of a `FORWARD` to 3 nodes followed by a sequential `STORE` operation of 10 kilobytes, instantly flushed on disk. The workload is then defined as 30 requests sent at a constant rate of 5 requests per second. The client command is:

```
--to node0 -F node1,node2,node3,<num_ack>
           -S 10k,sync -r 5 -n 30
```

where "nack" is the consistency level, or the number of acknowledgments, which is defined as the number of nodes the gateway has to wait before replying to the client. Each boxplot corresponds to a consistency level: the first group of

boxplots showcases a scenario under normal operating conditions, meaning the disk I/O is not oversaturated. Therefore, the latency difference is mainly due to network delays and the gathering overhead. The second group of boxplots showcases a scenario where one node is experiencing disk oversaturation, induced using the `fio` I/O tester tool. As expected, weaker consistency delivers an average latency comparable to the one observed under normal conditions (aside from negligible variations due to experimental error), whereas waiting for all the nodes results in a substantial performance hit, leading to a $\times 168$ increase in latency.

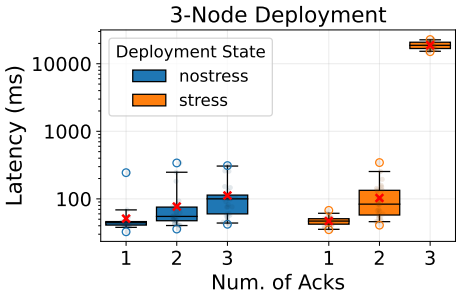


Fig. 10: Observed latency when `FORWARD`ing multiple `STORE` operations with varying acknowledgement levels.

G. Distributed System: Load Balancing

A key feature to achieve scalability is load balancing [4], which helps prevent any single server from overloading/underloading, thus enhancing performance, utilization, reliability, and availability. In this subsection, we use DistWalk to study latency implications of load balancing techniques by routing traffic to a cluster of nodes. As a side-effect, we also demonstrate DistWalk's composability by using a number of nodes in conjuction with IPVS, a layer 4 (i.e., connection-level) load balancer within the Linux kernel. Figure 11 explores 3 simple load balancing policies [13], Round-Robin (RR), Least-Connection (LC), and Shortest Expected Delay (SED), using a heterogeneous workload. More specifically, 4 clients emulate a latency-sensitive workload of 400 `COMPUTE` operations triggered at a constant rate of 50 requests per second. The compute duration is sampled from a uniform probability distribution ranging from $1ms$ to $10ms$. The command for a latency-sensitive client is:

```
--to lb -C unif:min=1k,max=10k -r 50 -n 400 --ns 5
```

where "lb" is the hostname where the load balancer is instantiated, and "ns" is the number of session re-connects. Client sessions are frequently closed and re-opened after 5 transmitted packets to allow for load balancing; Otherwise, the client's packets would always be routed to the same node. A fifth client emulates a saturating workload performing long-running computations in the range of $1s$ to $5s$, and at a lower rate of 5 requests per second. The corresponding command for the noisy client is:

```
--to lb -C unif:min=1m,max=5m -r 1 -n 15 --ns 5
```

Each group of boxplots corresponds to a different cluster size. As expected, RR performs considerably worse on smaller clusters on average, since distributing the requests equally amongst the available nodes is not ideal for heterogeneous workloads. SED performs well in every scenario since it selects the node with the shortest delay. However, the percentiles all show similar behaviors, as expected, because they are generally preferable for equally-performant nodes, which is not the case here due to the "noisy" client workload.

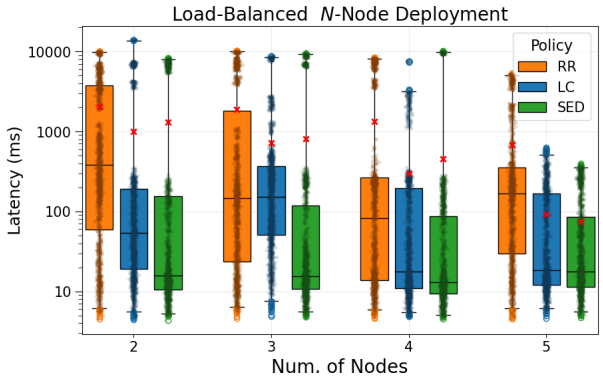


Fig. 11: Latency experienced by 4 latency-sensitive clients with noise from busy client.

H. Emulating an Apache Web Server with DistWalk

The Apache web server is a robust and flexible tool that offers three multi-processing modules (MPMs) to configure the pool of workers and determine how to dispatch HTTP requests: `mpm_prefork`, `mpm_worker`, and `mpm_event`. `mpm_prefork` uses a pool of processes, while the latter uses a pool of threads.

This subsection demonstrate how to emulate the timing behavior of an Apache web server using DistWalk. As a byproduct, the subsection also illustrates how to properly analyze a real workload, and translate it into DistWalk's terms for easy reproducibility. To achieve our goal, we analyzed a single-instance Apache deployment with 3 processes/threads, serving the default "Hello World" webpage. We used Apache Benchmark and `tcpdump` to gather comprehensive information on its compute, storage, and network activities over 5000 GET requests. From this data, we crafted the following commands (explained below):

```
node:   --nt 3 --am parent
client: --load-offset 0 -L 10918
         -C 1000 --ps 76 --rs 11192
         -r 85 -n 5000 --ns 5000
```

i) the `LOAD` operation of $\sim$10.9 kilobytes emulates the retrieval of the webpage from disk; ii) the load offset resets the file seek back to the beginning, emulating the file lookup; iii) the `COMPUTE` operation emulates the HTTP overhead, as DistWalk does not natively support this protocol (yet); iv) "ps" configures the send packet size, set to 76 bytes to match the payload size of a GET request from Apache Benchmark,

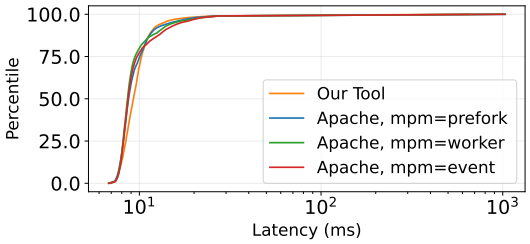


Fig. 12: Empirical CDF latency plot comparing the Apache web server and DistWalk's emulated counterpart.

as measured via `tcpdump`; v) "rs" is the reply packet size, accounting for the size of the webpage as well; vi) accept mode "parent" emulates the Apache's worker pool, since socket are handed off to threads in a similar way; and finally vii) the submitted traffic consists of 5000 requests.

Figure 12 validates the effectiveness of DistWalk in emulating Apache. DistWalk closely mirrors represent Apache's latency outputs, matching almost to the microsecond. The only discrepancy is in the worst-case latency, which appears to be $400ms$ less compared to Apache. In conclusion, we presented a reproducible methodology for using DistWalk to emulate the workload and latency of a simple web server.

V. Conclusions and Future Works

This paper introduces DistWalk, an open-source workload emulator for distributed systems. Thanks to its key features, DistWalk is capable of assessing the end-to-end timing behavior of complex applications and services consuming CPU, disk, and network resources across a geographically distributed infrastructure. Its versatility stems from the high configurability of its core components, the DistWalk client and node, and the ability to emulate graph-like interactions through an intuitive command-line interface. Moreover, DistWalk integrates seamlessly with existing tools such as load balancers, and orchestration platforms, without the need for ad-hoc configurations or third-party dependencies.

Regarding future works, the possible directions are manyfold. On the network side, we are planning to add support for additional networking options, protocols and primitives, including: security in data exchanges using TLS/SSL, enabling authenticated and encrypted communications; kernel bypass techniques like DPDK, that allow for offloading packet processing from the OS kernel to user-space processes, useful for emulating ultra-low latency and high-performance services, for which absolute performance is of utmost importance; richer networking topologies supporting fork/join points in addition to RPCs; and request exchanges through the HTTP(S) protocol to better emulate typical web servers and Service-Oriented Architecture (SOA) [15] traffic. Lastly, we plan to extend the current set of supported computational operations to expand the simplicity of the current approach, offering a richer set of building blocks for emulating more realistic distributed service patterns. For instance, we plan to introduce a variety of memory and computational stressors by integrating `stress-ng` to offer a broader variety of memory-intensive and CPU-intensive stress scenarios.

References

[1] R. Andreoli, R. Mini, P. Skarin, H. Gustafsson, J. Harmatos, L. Abeni, and T. Cucinotta. A multi-domain survey on time-criticality in cloud computing. *IEEE Transactions on Services Computing*, page 1–19, 2025.

[2] Remo Andreoli, Tommaso Cucinotta, and Daniel Bristot De Oliveira. Priority-Driven Differentiated Performance for NoSQL Database-as-a-Service. *IEEE Transactions on Cloud Computing*, 11(4):3469–3482, 2023.

[3] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343–477. Portland, OR, 2000.

[4] Timothy C. K. Chou and Jacob A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, (4):401–412, 1982.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[6] Andrea Detti, Ludovico Funari, and Luca Petrucci. µbench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

[7] Hua Fan, Aditya Ramaraju, Marlon McKenzie, Wojciech Golab, and Bernard Wong. Understanding the causes of consistency anomalies in apache cassandra. *VLDB Endowment*, 8(7):810–813, February 2015.

[8] Yu Gan et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 3–18, New York, 2019. ACM.

[9] Sascha Hunold and Alexandra Carpen-Amarie. Reproducible MPI Benchmarking is Still Not as Easy as You Think. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3617–3630, 2016.

[10] Joseph Idziorek, Alex Keyes, Colin Lazier, Somu Perianayagam, Prithvi Ramanathan, James Christopher Sorenson III, Doug Terry, and Akshat Vig. Distributed transactions at scale in amazon DynamoDB. In *USENIX Annual Technical Conference*, pages 705–717, Boston, MA, July 2023.

[11] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. The limit of horizontal scaling in public clouds. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 5(1), February 2020.

[12] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *IEEE International Symposium on Workload Characterization*, pages 1–10, 2016.

[13] Pawan Kumar and Rakesh Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM computing surveys (CSUR)*, 51(6):1–35, 2019.

[14] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[15] Kathryn B Laskey and Kenneth Laskey. Service oriented architecture. *Wiley Interd. Reviews: Computational Statistics*, 1(1):101–105, 2009.

[16] Juri Lelli et al. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[17] John Nagle. Congestion control in IP/TCP internetworks. *ACM SIGCOMM Computer Communication Review*, 14(4):11–17, 1984.

[18] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[19] William Schultz, Tess Avitabile, and Alyson Cabral. Tunable consistency in mongodb. *VLDB Endowment*, 12(12):2071–2081, August 2019.

[20] Nadia Ben Seghier and Okba Kazar. Performance benchmarking and comparison of NoSQL databases: Redis vs mongodb vs Cassandra using YCSB tool. In *International Conference on Recent Advances in Mathematics and Informatics*, pages 1–6. IEEE, 2021.

[21] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. Fogify: A Fog Computing Emulation Framework. In *IEEE/ACM Symposium on Edge Computing*, pages 42–54, 11 2020.

[22] Tianyi Yu et al. Characterizing serverless platforms with serverlessbench. In *11th ACM Symposium on Cloud Computing*, page 30–44, NY, 2020.

[23] Siyuan Zhou and Shuai Mu. Fault-Tolerant replication with Pull-Based consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation*, pages 687–703, April 2021.