# DoubleFaceAD: A New Datastore Driver Architecture to Optimize Fanout Query Performance

Shungeng Zhang
Louisiana State University
szhan45@lsu.edu

Qingyang Wang
Louisiana State University
qwang26@lsu.edu

Yasuhiko Kanemasa
FUJITSU LABORATORIES LTD.
kanemasa@fujitsu.com

Jianshu Liu
Louisiana State University
jliu96@lsu.edu

Calton Pu
Georgia Tech
calton.pu@cc.gatech.edu

## Abstract

The broad adoption of fanout queries on distributed datastores has made asynchronous event-driven datastore drivers a natural choice due to reduced multithreading overhead. However, through extensive experiments using the latest datastore drivers (e.g., MongoDB, HBase, DynamoDB) and YCSB benchmark, we show that an asynchronous datastore driver can cause unexpected performance degradation especially in fanout-query scenarios. For example, the default MongoDB asynchronous driver adopts the latest Java asynchronous I/O library, which uses a hidden on-demand JVM level thread pool to process fanout query responses, causing a surprising multithreading overhead when the query response size is large. A second instance is the traditional wisdom of modular design of an application server and the embedded asynchronous datastore driver can cause an imbalanced workload between the two components due to lack of coordination, incurring frequent unnecessary system calls. To address the revealed problems, we introduce *DoubleFaceAD*–a new asynchronous datastore driver architecture that integrates the management of both upstream and downstream workload traffic through a few shared reactor threads, with fanout-query-aware priority-based scheduling to reduce the overall query waiting time. Our experimental results on two representative application scenarios (YCSB and DBLP) show `DoubleFaceAD` outperforms all other types of datastore drivers up to 34% on throughput and 1.9× faster on 99$th$ percentile response time.

***Keywords*** distributed datastores, fanout queries, asynchronous, performance.

## 1 Introduction

Distributed datastores are widely used by modern web applications to improve system scalability and response time through data partitioning and fanout queries [16, 29, 46]. To speed up the tail response time, which has become a particular concern for latency-sensitive web-facing applications [17, 19, 20, 44], a common practice is to parallelize sub-operations (fanout queries) across multiple partitions [15, 17, 27, 43]. Since each partition only receives a limited partial workload, the system bottleneck is likely to shift from the original datastore tier to the upstream application server tier, or more specifically, the datastore drivers that interact with the downstream datastore cluster via a large number of fanout queries.

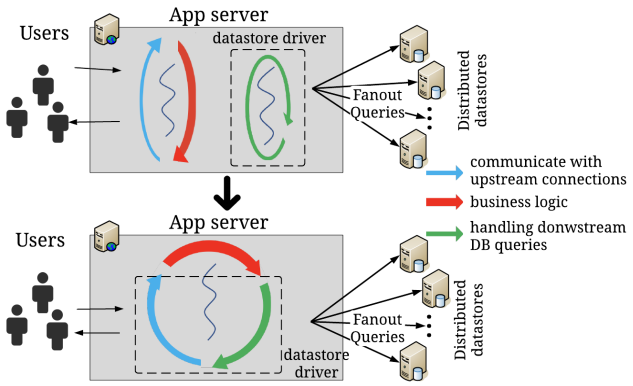



**Figure 1. The upper figure is the widely-adopted design where an application server and the corresponding datastore driver handle the upstream (HTTP requests) and the downstream (fanout queries) traffic separately. The lower figure is our `DoubleFaceAD` design where the application server and the datastore driver are integrated to handle traffic from both sides.**

A recent study [26] advocates the asynchronous event-driven architecture as a natural choice for the datastore drivers to handle highly concurrent fanout queries. One reason is that for some applications it is unnecessary to wait for all fanout queries to return before continuing execution and can often reduce the application completion time significantly [25–27]. The second reason, which we consider is more important, is to avoid multithreading overhead caused by processing highly concurrent fanout queries. Unlike a thread-based datastore driver using one-thread-per-connection to handle each fanout query, an asynchronous event-driven datastore driver can use only one or a few threads to handle all the fanout queries, significantly reducing the multithreading overhead [37, 46].

Though conceptually simple, taking advantage of the asynchronous event-driven architecture to construct high-performance datastore drivers to interact with scalable

Doi: 10.1145/3423211.3425684

distributed datastores is a significant challenge. This is because of the obscured request processing control flow inherited from the event-driven programming model [47, 55], making the performance of an asynchronous datastore driver not only lie in the design of the asynchronous drivers themselves, but also their interactions with other components in the system. For example, our experiments show that the asynchronous drivers of some popular datastores (e.g., HBase and DynamoDB) can have a significant performance drop under high concurrency workloads similar to their thread-based counterpart (see Figure 4). On the other hand, while the two latest asynchronous drivers for MongoDB do outperform their thread-based counterpart, their performance superiority is surprisingly reversed when fanout query response size changes (see Figure 5).

In this paper, we show that building high-performance asynchronous datastore drivers to communicate with the distributed datastores requires an integrated design of the application server and the datastore driver itself (see Figure 1). This is primarily due to the simplified event processing flow and the streamlined interaction between the two components. Concretely, the network events from both the upstream (the client) and the downstream (the distributed datastores) are handled by pluggable event handlers (business logic) that run on a few *shared* reactor threads. Such an integrated design is especially important in fanout query scenarios since the network events from the downstream can be much larger than those from the upstream, causing an imbalanced workload between the otherwise separate frontend and backend reactor threads. As we will show in Section 4, the imbalanced workload problem can incur frequent unnecessary system calls (e.g., *select()* in Linux) for separate reactor threads.

The first contribution of the paper is a quantitative evaluation (based on YCSB [12] workload and dataset) illustrating the unexpected performance variation of different datastore driver architectures under varying workload conditions. For example, we found that the asynchronous DynamoDB [4] and HBase [22] drivers have a surprisingly similar performance drop under high-level workload concurrency to their thread-based counterparts. While the two asynchronous MongoDB drivers perform well under high concurrency workload, their performance has very different sensitivity level to the change of fanout query response size.

The second contribution is a deep-dive analysis revealing the causes of the performance degradation of the four latest asynchronous datastore drivers. For example, the asynchronous drivers for DynamoDB and HBase essentially adopt the thread-based design but with an asynchronous interface, thus introducing high multithreading overhead when facing high concurrency workload. The default asynchronous driver for MongoDB is based on the latest Java asynchronous I/O library (AIO in JDK1.7) while the alternative one is built on top of a widely-used asynchronous network I/O framework Netty [38]. We found that the former one still uses a hidden on-demand JVM-level worker thread pool to process fanout query responses, causing non-trivial multithreading overhead when query response size is large (e.g., 20kB). For the latter one, we found an interesting imbalanced workload problem between the application server and the embedded datastore driver due to a lack of coordination between them, causing frequent unnecessary system calls.

As the third and the main contribution, we introduce a new Asynchronous Datastore driver architecture (`DoubleFaceAD`) that integrates the design of the application server and the datastore driver. Concretely, `DoubleFaceAD` manages both upstream and downstream connections using one or a few reactor threads to handle the network events from both sides. Such an integrated design maintains both high performance and high flexibility since business logic and datastore driver management functions can be pluggable event handlers that run on the same set of threads. We further introduce a fanout-query-aware priority-based job scheduling algorithm to reduce the tail latency of concurrent fanout queries. The experimental results show that `DoubleFaceAD` outperforms all other types of driver architectures up to 34% on throughput and 1.9× faster on $99th$ percentile response time.

In general, our results strongly suggest that the asynchronous design of datastore drivers has a potentially significant performance advantage over the traditional thread-based design when interacting with distributed datastore clusters through fanout queries. In fact, our work has a much wider impact on real-world modern cloud systems since fanout queries are not only common in distributed datastores, but also many large-scale distributed systems adopting microservices and serverless computing architecture [2, 6, 34, 48].

The rest of the paper is organized as follows. Section 2 illustrates the performance variation of three representative datastores drivers (DynamoDB, HBase, and MongoDB). Section 3 describes the unexpected multithreading overhead problem in the Java AIO-based asynchronous MongoDB driver. Section 4 explains the imbalanced workload problem in the Netty-based asynchronous MongoDB driver. Section 5 and 6 introduce and evaluate our new asynchronous datastore driver architecture, respectively. Section 7 discusses the cost and limitations of `DoubleFaceAD`. Section 8 summarizes the related work and Section 9 concludes the paper.

## 2 Background and Motivation

### 2.1 Server Connector Categories

Connectors are commonly used by modern internet servers to communicate with each other or with end-users. For example, a Tomcat application server uses an HTTP connector to communicate with end-users while uses a datastore driver to communicate with the downstream MySQL server. Concretely, the Tomcat server uses the HTTP connector to

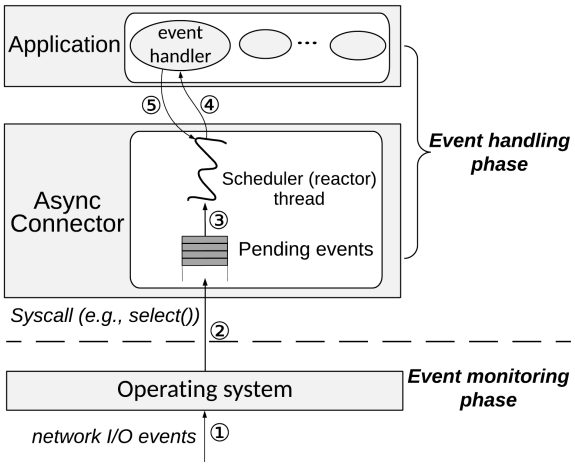



**Figure 2. Illustration of a general asynchronous event-driven connector architecture.**

(a) Software and hardware setup

| Software Stack | | Hardware Stack | |
|---|---|---|---|
| *Application Server* | netty 4.1.20 | *Server Type* | Lenovo ThinkServer RD640 |
| *Datastore Server* | HBase 2.1.0 | *Processor* | Intel Xeon CPU E5-2640, 2.50GHz, Hexa-core |
| | Amazon DynamoDB | *Memory* | 16GB |
| | mongodb-org-server 3.4.9 | *Disk* | SATA 500GB 7200RPM |
| *Datastore Driver* | hbase-client 2.1.4 | *Network* | 1Gbps |
| | aws-java-sdk-dynamodb 1.10.2 | | |
| | monogodb-driver-sync 3.6 | | |
| | monogodb-driver-async 3.6 | | |
| *Operating System* | RHEL 6.3 (kernel 2.6.32) | | |

(b) Experimental sample toplogy

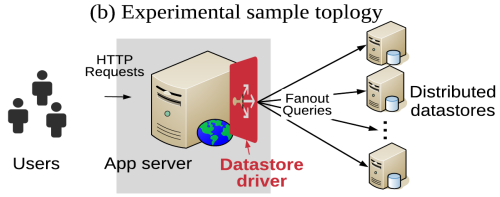



**Figure 3. Details of experimental setup.**

accept/close upstream network connections, read client requests and respond via the established connections; and if necessary, send queries to the downstream MySQL through the datastore driver. Such a scenario is prevalent in practice: an application server (via a frontend connector) manages the upstream connections while the backend connector manages the downstream connections separately.

**Thread-based connectors.** Thread-based connectors are used by servers adopting the traditional synchronous RPC-style request-response communication, in which the servers follow the one-thread-per-connection model. In this model, each established connection is handled by a dedicated worker thread until the connection close. In this case, concurrent requests from clients consume the same amount of worker threads in the server, bringing the well-known multithreading overhead such as scheduling and lock contention [51].

Asynchronous connectors are for servers that do not want to be blocked by busy waiting for responses [46] (e.g., from datastores). They can be categorized into two types.

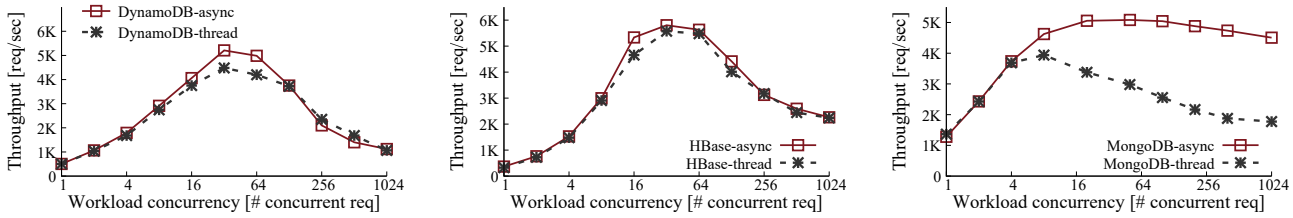
***Type-1 asynchronous.*** This type simulates the asynchronous behavior of the main thread of a server by delegating each asynchronous API call (e.g., sending out fanout queries) to a worker thread in a pre-defined thread pool [26]; each worker thread still uses the synchronous RPC request-response to communicate with other servers. Such a design is easy to implement, however, it brings the same multithreading overhead as the thread-based connectors.

***Type-2 asynchronous.*** This type employs an event-driven mechanism to manage connections and process all network I/O events using one or a few threads [28, 46, 55]. Figure 2 shows the interactions of an asynchronous connector with the application and the underlying operating system. The asynchronous connector is responsible for managing network I/O events from established connections by continually looping over two phases. The first phase is event monitoring, which determines connections with pending network I/O events (readable or writable). Concretely, the underlying operating system (OS) notifies the reactor thread of the asynchronous connector the occurrence of pending events through OS-specific event notification mechanisms such as *select*, *poll*, or *epoll*. The second phase is event handling, where the reactor thread repeatedly dispatches the connections with pending events to corresponding event handlers to process the business logic [28, 37, 51]. In practice, there are two typical designs of the second phase.

1. *Type-2a asynchronous.* This type is a single-threaded asynchronous connector which uses the same reactor thread to loop over the aforementioned two phases, for example, Node.js [1] and Lighttpd [33]. Such a design is especially beneficial for in-memory workloads because multithreading overhead is minimum [32].
2. *Type-2b asynchronous.* This type is to use a worker thread pool in the second phase to concurrently process connections with pending events. Such a design is supposed to efficiently utilize CPU resources in the case of transient disk I/O blocking or multi-core environments [37, 51].

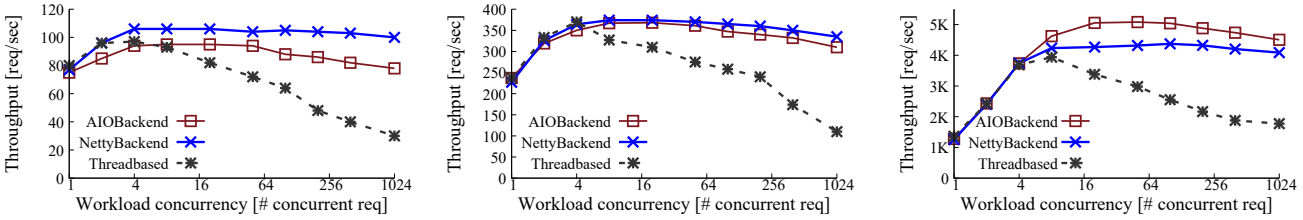
## 2.2 Experimental Environment

**Datastore drivers under study.** We study three representative NoSQL datastore drivers (DynamoDB, HBase, and MongoDB) that follow the aforementioned thread-based or asynchronous designs. Other than thread-based drivers, the asynchronous drivers for DynamoDB [4] and HBase [22] adopt the *Type-1 asynchronous* design. MongoDB has two asynchronous drivers. The default one adopts the *Type-2b* design, which is based on the latest Java asynchronous I/O (AIO) library and uses a JVM-level on-demand worker thread pool for event processing in the second phase. The alternative one adopts the *Type-2a* design, which is built on top of Netty [38], a widely used asynchronous event-driven network I/O framework. We will experimentally illustrate their performance differences and explain the root causes next.




(a) DynamoDB driver case. (b) HBase driver case. (c) MongoDB driver case.

**Figure 4. The performance impact of different datastore driver architectures (thread-based vs. asynchronous).**




(a) The 20kB response size case. (b) The 1kB response size case. (c) The 0.1kB response size case.

**Figure 5. Throughput comparison among different drivers for MongoDB under different workload concurrencies and fanout query response sizes.** The fanout factor is fixed to 5. Subfigures (a) and (c) show that the throughput order between `AIOBackend` and `NettyBackend` is reversed when the fanout query response size changes.

**Experimental setup.** Figure 3 shows our experimental setup. We compare the performance of a Netty-based application server equipped with the aforementioned datastore drivers communicating with the downstream datastores (e.g., DynamoDB, HBase, and MongoDB). The experiments were carried out by allocating a dedicated physical node to each server (including datastores [1]). We use a Netty-based application server because it is among the best to efficiently handle high concurrency workload [39]. We refer to the application server equipped with the thread-based and asynchronous driver for a specific data store as **XXX-sync** and **XXX-async**, respectively, where **XXX** can be either DynamoDB, HBase, or MongoDB. Since MongoDB has two asynchronous drivers, we refer the server with the Netty-based driver as **NettyBackend** and the one with Java AIO-based driver as **AIOBackend**. All the server implementations will be available in GitHub [54] upon publication.

We use JMeter [21] to generates HTTP requests (e.g., web search query) to access the application server, which will trigger fanout queries to a cluster of NoSQL datastores. These HTTP requests are categorized into three types: large, medium, and small, with which the downstream datastores respond with 20kB, 1kB, and 0.1kB of data for each fanout query, respectively. We choose these three sizes because they are representative workloads for realistic key-value stores [3]. For each HTTP request, the application server will simply assemble the responses from the fanout queries and send it back to the end-user. JMeter uses one thread to simulate each end-user. Each thread will immediately issue the next HTTP request once it receives the response from the previous request. Thus by specifying the number of threads in JMeter, we can precisely control the workload concurrency sent to the application server. For downstream datastores, we shard YCSB [12] dataset across 20 datastores; thus we can control the fanout factor from 1 to 20, which is a reasonable fanout factor range for distributed datastores [52]. Each datastore shard contains one-million 1-kB records and each contains a primary key and ten 0.1kB fields, thus the size of each shard is about 1GB which can fit into memory. Therefore, the downstream datastore will perform a scan query for a large response while a point-lookup query for a small response.

[1]We use Amazon DynamoDB [42] as our remote DynamoDB cluster while the HBase and MongoDB clusters are in our local cloud testbed.

### 2.3 Significant Performance Variations of Different Asynchronous Datastore Drivers

In this section, we show the significant performance variations of the same application server equipped with different datastore drivers. Our goal is to illustrate the importance of the problem. The detailed explanation is in Sections 3 and 4.

We first compare the performance of thread-based and asynchronous drivers of MongoDB, DynamoDB, and HBase as workload concurrency gradually increases from 1 to 1024, shown in Figure 4. In all cases, we set the fanout factor to 5 and fanout query response size to 0.1kB. The performance of all the three thread-based datastore drivers degrades dramatically at high workload concurrency, which is expected due to the well-known multithreading overhead as previous studies show [49–51, 55]. However, the three asynchronous drivers behave very differently. For example, both `DynamoDB-async` and `HBase-async` encounter the same level of throughput

drop at high workload concurrency as their thread-based counterpart while MongoDB-async (the default asynchronous MongoDB driver AIOBackend) achieves the expected high performance, for example, 140% higher throughput than its thread-based counterpart at workload concurrency 1024. The poor performance of DynamoDB-async and HBase-async is due to their *Type-1* design, which essentially uses a thread pool to simulate the asynchronous behavior of the main thread of the server, thus introducing non-trivial multithreading overhead at high workload concurrency.

More interesting phenomena are observed when we compare the performance of two asynchronous drivers for MongoDB (AIOBackend and NettyBackend) under different workload concurrency and fanout query response sizes, shown in Figure 5. AIOBackend adopts the *Type-2b* design while NettyBackend adopts the *Type-2a* design.

The first interesting observation is that the throughput of AIOBackend starts to drop as the workload concurrency increases beyond 64 especially in the 20kB query response size case, shown in Figure 5(a). Such a throughput drop is interesting because the same phenomenon does not occur in NettyBackend. In Section 3, we will explain that AIOBackend adopts the latest Java asynchronous I/O (AIO) library which uses an on-demand worker thread pool to process datastore query response, generating unexpected high multithreading overhead when facing large query response size and high workload concurrency.

The second interesting observation is the throughput order reversal between NettyBackend and AIOBackend when the datastore query response size decreases from 20kB to 0.1kB. For example, NettyBackend achieves 19% higher throughput than AIOBackend when workload concurrency is 100 in the 20kB case (see Figure 5(a)), however, under the same workload concurrency, AIOBackend achieves 15% higher throughput than NettyBackend in the 0.1kB case (see Figure 5(c)). The only change is the datastore query response size. In Section 4, we will explain that small datastore query response size could lead to imbalanced workload between upstream and downstream connections in NettyBackend, causing a large number of unnecessary system calls (e.g., *select()* in Linux) and thus high CPU overhead.

## 3 Unexpected Multithreading Overhead in Type-2b Asynchronous Driver

In this section, we study the root cause of why asynchronous AIOBackend (*Type-2b*) achieves lower performance than NettyBackend (*Type-2a*) at high concurrency workload as shown in Figure 5(a). What this study reveals is that the *Type-2b* asynchronous driver has inherent deficiencies in its basic design when it handles high concurrency of fanout query responses from the downstream datastores.

AIOBackend essentially adopts the *Type-2b* asynchronous design, which uses one or a few reactor threads to monitor network I/O events, and an on-demand worker thread pool

**Table 1. Both AIOBackend and Threadbased encounter more context switches and lock contention overhead than NettyBackend.** Fanout query response size is 20kB.

| | | AIOBackend | NettyBackend | Threadbased |
|---|---|---|---|---|
| Throughput [req/sec] | | 88 | 106 | 64 |
| Concurrent running threads [/sec] | | 22 | 3 | 48 |
| Context Switch [/sec] | | 4550 | 670 | 5338 |
| **Syscalls cpu usages** | **Locking (mutex)** | **7.7%** | **1.2%** | **35.3%** |
| | **thread initiation** | **6.0%** | **0.8%** | **1.1%** |
| Total CPU usage | | 100.0% | 100.0% | 100.0% |

to process query responses from the downstream datastores, as shown in Figure 6. Specifically, the frontend is a Netty-based reactor thread (application server) to handle HTTP requests from clients; the backend is an AIO-based MongoDB driver that interacts with the downstream datastore cluster. To improve the performance of event monitoring and handling, the AIO-based MongoDB driver exploits the asynchronous I/O supported in the latest Java Virtual Machine (JVM version after 1.7), which uses a JVM-level reactor thread to monitor the network I/O events and a JVM-level worker thread pool to handle those events (in our case, the events are fanout query responses). By default, the JVM-level worker thread pool adopts an on-demand strategy where a worker thread will be launched as needed and terminated after a certain period of idle time. Concretely, to process a new client request, the following four main stages are taken:

1. The *Frontend* reads and parses HTTP requests from clients, and then sends related fanout queries to the downstream datastores (①~③ in Figure 6);
2. The *Backend* JVM-level reactor thread monitors the availability of read events (fanout query responses) from the downstream datastores and wraps the read events into a task queue (④ in Figure 6);
3. The *Backend* JVM-level worker thread pool fetches and processes the read events from the task queue, and passes the intermediate results to the *Frontend* (⑤ in Figure 6);
4. The *Frontend* checks the completeness of the intermediate results from all the fanout queries regarding each specific HTTP request, and assembles the final response to return back to the clients (⑥ in Figure 6).

We found the unexpected multithreading overhead mainly comes from the third stage (⑤ in Figure 6). In this stage, the JVM-level worker thread pool fetches read events (fanout query responses) from the task queue for further processing. Since this stage does not involve network I/O (all the read events are ready), a few worker threads should be enough

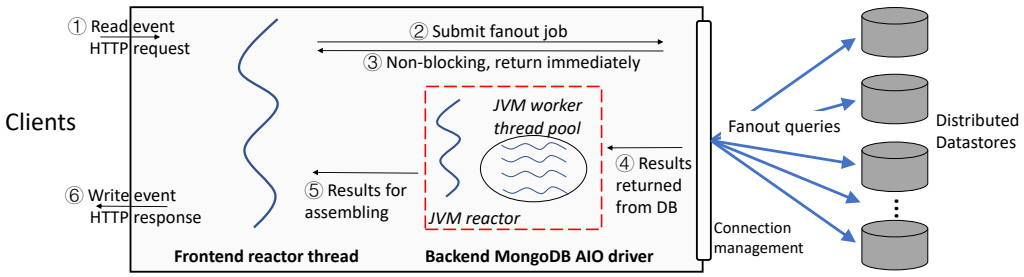



**Figure 6. Architecture of the asynchronous `AIOBackend` server.**

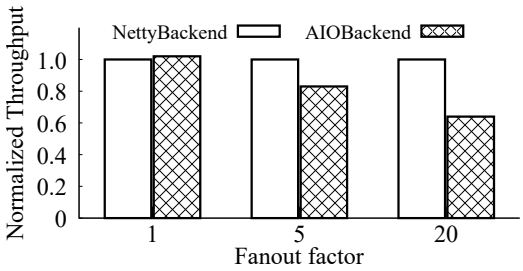



**Figure 7. The throughput of `AIOBackend` decreases as the fanout factor increases from 1 to 20 when the fanout query response size is large (20kB).**

to process all the fanout query responses as long as each response size is small (processing time is proportional to query response size). However, once the fanout query response size is large, more worker threads will be launched to concurrently process the query responses due to the on-demand scaling of the worker thread pool. In this case, the associated multithreading overhead (e.g., locking, scheduling) becomes large, especially when the workload concurrency is high.

We use `perf` [23] to demonstrate the non-trivial CPU overhead caused by multithreading in asynchronous `AIOBackend`. Table 1 compares some key performance metrics among the servers `AIOBackend`, `NettyBackend`, and `Threadbased`. The workload concurrency from clients is fixed to 100. The fanout factor is 5 and each fanout query response size is 20kB. As expected, the throughput of `Threadbased` is the lowest (64 req/sec) among the three because of the multithreading overhead caused by a large number of concurrently running threads (48). Especially, the mutex lock CPU overhead [2] involved in multithreading in `Threadbased` reaches 35.3%, significantly higher than that in the other two asynchronous servers. Nevertheless, the asynchronous `AIOBackend` server is also observed 7.7% CPU overhead in mutex lock. This is because the AIO-based MongoDB driver in `AIOBackend` adopts a JVM-level worker thread pool to process fanout query responses as previously explained. For example, the number of concurrently running threads in `AIOBackend` is 22 while such number is 3 for `NettyBackend` (the detailed

[2]Mutex lock contention occurs when multiple threads try to access a shared resource; the higher contention, the more CPU overhead.

**Table 2. `NettyBackend` wastes more CPU on *select()* system calls than `AIOBackend`.** The fanout factor is 5 and the DB query response size 0.1kB.

| | AIOBackend | NettyBackend |
|---|---|---|
| Throughput [req/sec] | 5042 | 4373 |
| # of `select()` [30s runtime] | **54K** | **155K** |
| `select()` CPU utilization [%] | **1.1%** | **8.1%** |

design is in Section 4), which helps explain the throughput difference of the two asynchronous servers.

`perf` also reports that 6.0% CPU resource of `AIOBackend` is used in initiating new threads (the thread initiation [36] factor in Table 1) while the other two servers do not have such CPU overhead. This is because of the on-demand scaling of the worker thread pool in the AIO-based MongoDB driver when handling fluctuations of fanout-query responses from the downstream datastores.

We further study the impact of different fanout factors on the performance of asynchronous `AIOBackend` and `NettyBackend`. To clearly show the different impact, we use `NettyBackend` as the baseline and compare the normalized throughput between the two servers, shown in Figure 7. The client workload concurrency still keeps 100 and the fanout factor of queries to downstream datastores varies between 1 to 20. While the throughput of `AIOBackend` and `NettyBackend` is almost the same when the fanout factor is 1, `AIOBackend` throughput decreases by 36% compared to `NettyBackend` when the fanout factor increases to 20. Such a performance degradation is due to the increase of the number of concurrent fanout query responses in the server. Thus more concurrent worker threads are launched in `AIOBackend` to process the increased fanout query responses, resulting in high multithreading overhead as explained in Table 1.

## 4 Imbalanced Workload Problem of Type-2a Asynchronous Driver

In this section, we study the design deficiencies of the asynchronous `NettyBackend` when it handles small size responses from the downstream datastores. `NettyBackend` adopts *Type-2a* asynchronous design, which essentially uses

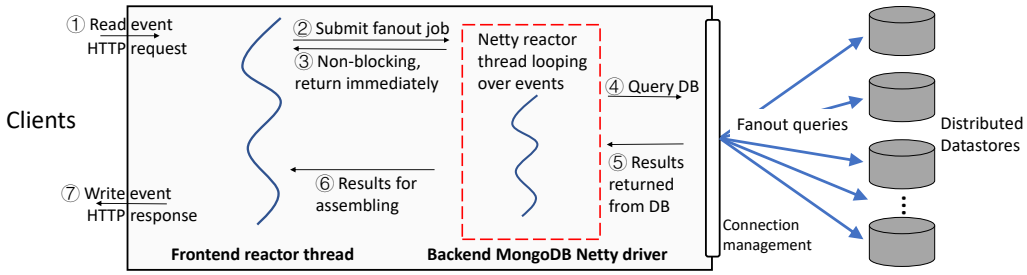



**Figure 8. Architecture of the asynchronous `NettyBackend` server.**

**Table 3. Imbalanced workload between Frontend and Backend reactor threads leads to inefficient system call *select()* in `NettyBackend`.** This table shows a 30-second measurement of system call *select()* in `NettyBackend`.

| # of Backend reactor threads | One | Two | Four |
|---|---|---|---|
| Throughput [req/sec] | 3537 | 4373 | 3397 |
| total # of *select()* (frontend+backend) | 248K (238K+10k) | 155K (87K+68K) | 366K (12K+354K) |
| total # of events (frontend+backend) | 401K (174K+227K) | 509K (289K+220K) | 390K (330K+60K) |
| events per *select()* (frontend / backend) | **0.7 / 22.7** | **3.3 / 3.2** | **27.5 / 0.2** |

only a few reactor threads to both monitor and handle network I/O events from the downstream connections, shown in Figure 8. Concretely, `NettyBackend` employs one reactor thread in the frontend connector to manage connections from upstream clients and another few reactor threads in the backend connector to manage connections to the downstream datastores. While such a design truly reduces the multithreading overhead, we found that the lack of coordination of event processing between the frontend and the backend reactor threads in `NettyBackend` could lead to a serious imbalanced workload problem between the two, causing frequent unnecessary system calls (e.g., *select()*).

We start our investigation by using `perf` to analyze some key system calls and their CPU usage in both `AIOBackend` and `NettyBackend` to capture any potential overhead, shown in Table 2. The workload concurrency for both servers is set to 100. The fanout factor keeps 5 while the fanout query response size reduces from 20kB to 0.1kB. Under this workload condition, the CPU usage of both servers reaches 100%, however, `NettyBackend` achieves about 15% less throughput than `AIOBackend`. `perf` reports that `NettyBackend` has 155K *select()* system calls during a 30-second experiment runtime, which is nearly 3 times of that (54K) in `AIOBackend`. System calls are generally expensive due to the related kernel crossing overhead [8, 45]. For example, the CPU usage of the system call *select()* in `NettyBackend` is 7% more (8.1% − 1.1%)

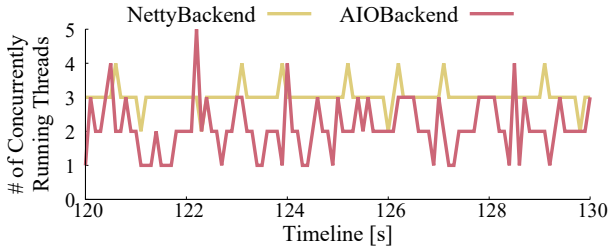



**Figure 9. Concurrent running threads comparison between `NettyBackend` and `AIOBackend`.**

than that in `AIOBackend`, contributing to about half of the throughput difference between the two servers.

We find out that the exceptionally high number of *select()* system calls in `NettyBackend` is due to the imbalanced workload between the frontend and the backend reactor threads. *select()* is invoked during the event monitoring phase when a Netty reactor thread performs an event occurrence checking on all established connections. Once returning a list of connections with the pending network I/O events, the reactor thread will iterate through the list and handle each event one by one. In general, the longer the list per *select()*, the more efficient of the reactor thread because it can process the same total number the events with fewer cycles. In the case of `NettyBackend`, the frontend and the backend reactor threads work independently to process events from upstream clients (i.e., HTTP requests) and downstream datastores (i.e., fanout query responses), respectively. Our experiments show that either the frontend or the backend reactor threads may make inefficient *select()* system calls due to the imbalanced workload, leading to high CPU overhead.

Table 3 shows the detailed profiling results by `perf` when we configure one, two, and four backend reactor threads in `NettyBackend` to process the fanout query responses from the downstream datastores. The three configurations are denoted as `OneCase`, `TwoCase` (the default setting), and `FourCase`, respectively. The number of frontend reactor threads is always set to 1 and the workload condition keeps the same as in Table 2. The results show `TwoCase` outperforms `OneCase` and `FourCase` by 22% and 44% in throughput, respectively. The throughput gap among the three servers can be explained by the CPU overhead resulting from different levels of *select()* system calls. For example, in `OneCase`, totally

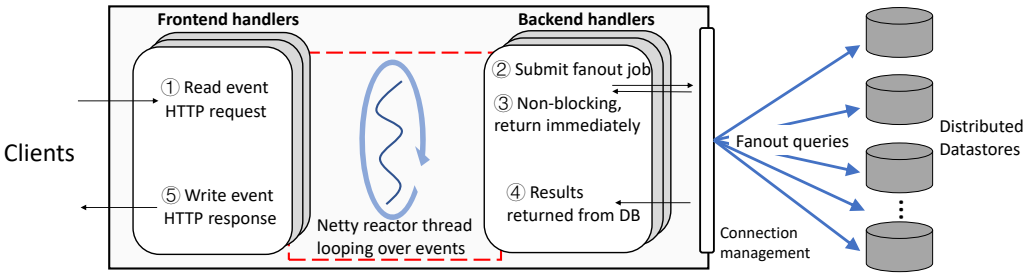



**Figure 10. Architecture of `DoubleFaceNetty`.** One (or a few) reactor thread is (are) responsible for managing both the events from the upstream clients (e.g., HTTP requests) and from the downstream datastores (e.g., fanout query responses).

248K *select()* system calls occur in `NettyBackend`, among which 238K are invoked by the frontend reactor thread while 10K by the backend reactor thread, suggesting imbalanced workload between two sides. More importantly, the frontend reactor thread only processes 0.7 events per *select()*, suggesting low efficiency of the frontend reactor thread because it processes zero event in many *select()* system calls (thus called "spurious" *select()*), purely wasting system resources. On the other hand, once we increase the number of backend reactor threads to four (`FourCase`), the *select()* system calls invoked by the backend reactor threads become extremely inefficient: only 0.2 events per *select()*, suggesting even more frequent "spurious" *select()* and worse throughput compared to the other two cases.

Large amount of "spurious" *select()* system calls occur due to two reasons. First, both the frontend and the backend reactor threads in `NettyBackend` continuously loop over the event monitoring and event processing phases even if there are no pending events, leading to "spurious" *select()* system calls [3]. Second, the ratio of frontend and backend reactor threads (assuming they have the same priority) determines the chance which side grabs the shared CPU core during each CPU scheduling. For example, in the `FourCase` in Table 3, the four backend reactor threads (compared to one frontend reactor thread) will likely grab the shared CPU core even if some backend reactor threads have no pending events to process, leading to "spurious" *select()* system calls and high CPU overhead.

Readers may assume that `TwoCase` should be the optimal configuration since the events per *select()* for both frontend and backend are nearly the same (the last row in Table 3), indicating a balanced workload between the frontend and the backend. However, such a balance is only statistically true over a long-running period, it does not hold in real-time. Figure 9 shows the number of concurrently running threads of the `TwoCase` `NettyBackend` and `AIOBackend` during a 10-second runtime. `TwoCase` shows relatively stable three running threads (one frontend and two backend reactor threads) while the running threads in `AIOBackend` varies significantly with the timeline, indicating the effectiveness of the on-demand scaling strategy of the worker thread pool in balancing workload between the frontend and the backend connector in `AIOBackend`. Such a balancing workload in `AIOBackend` makes it achieve 15% higher throughput than the `TwoCase` `NettyBackend` as shown in Table 2, indicating a static allocation of reactor threads in either the frontend or the backend may not achieve the optimal performance.

[3]Netty will set a small timeout (in a few milliseconds) between consecutive loops. Our server log confirms that a large amount of *select()* system calls return a few or even an empty set of connections with pending events.

## 5 `DoubleFaceAD`: A New Driver Architecture

So far we have studied two problems of typical designs of asynchronous datastore drivers when dealing with fanout queries: 1) the multithreading overhead due to the use of thread pools (*Type-1* and *Type-2b* asynchronous); 2) the unnecessary system call overhead caused by imbalanced workload between frontend and backend reactor threads (*Type-2a* asynchronous). We found that such two problems not only exist in distributed datastore drivers, but also in many other modern cloud systems such as microservices applications where fanout queries are also common scenarios. We summarize this representative asynchronous software in Table 4.

To address the above two problems, we introduce a new datastore driver architecture, `DoubleFaceAD`, with a design goal of achieving both high throughput and low tail response time. Specifically, `DoubleFaceAD` integrates the frontend and the backend by managing both upstream connections (from clients) and downstream connections (from datastores) through the same set of reactor threads to reduce the CPU overhead (see Section 5.1). We further introduce a fanout-query aware priority-based scheduling inside `DoubleFaceAD` to mitigate the tail response time (see Section 5.2).

### 5.1 Integrated Design of `DoubleFaceAD`

The main idea of `DoubleFaceAD` is to use the same set of limited reactor threads to handle events from both the upstream clients and the downstream datastores, shown in Figure 10. Concretely, when a `DoubleFaceAD`-based server receives an HTTP request, the following four stages are involved:

1. A reactor thread reads and processes the HTTP request, and then sends related fanout queries to the downstream datastores asynchronously (①~③ in Figure 10);
2. The same reactor thread monitors the occurrence of read events (fanout query responses) from the downstream

**Table 4.** Summary of representative software sharing the design deficiencies of datastore drivers studied in this paper

| Type | Name | Datastore model | Note |
|---|---|---|---|
| $Type-1$ | Amazon DynamoDB driver [4] | Document & KV-oriented | Simulate the async behavior of the main thread of a server by delegating each async API call to a pre-defined worker thread pool. It has the same multithreading overhead as the thread-based design. |
| | Apache Hbase driver [22] | Column-oriented | |
| $Type-2a$ | Apache Cassandra driver [13] | Document-oriented | Drivers in this type use the same reactor thread to loop over both the event monitoring and the event processing phase. Multiple copies of reactor threads can be launched, and each reactor thread manages a subset of connections (i.e., the $N$-copy model [53]). |
| | Azure Cosmos DB driver [5] | Document & Column & KV & Graph-oriented | |
| | Redis driver [41] | KV-oriented | |
| | MongoDB driver (Netty-based) | Document-oriented | |
| $Type-2b$ | MongoDB driver (AIO-based) | Document-oriented | The main reactor thread monitors the occurrence of pending events (the first phase), and a worker thread pool for event processing (the second phase). |

connections, and handles each query response (④ in Figure 10);

3. The same reactor thread checks the completeness of the intermediate results from all the fanout queries for each specific HTTP request and assembles the final response to return back to the client (⑤ in Figure 10).
4. Multiple reactor threads are launched to fit a multi-core environment (follow the N-copy model [53])

The above integrated design of `DoubleFaceAD` makes two significant changes compared to the previously discussed `AIOBackend` and `NettyBackend` (see Section 3 and 4). First, in `AIOBackend`, the fanout query responses will be handled by an on-demand scaling thread pool, which brings a non-trivial multithreading overhead (e.g., mutex lock and thread initialization) when the concurrency of fanout query responses is high. `DoubleFaceAD` resolves this problem by letting a limited number of reactor threads (e.g., one per CPU core) manage both the event monitoring and handling, thus reducing the multithreading overhead and avoiding the thread initialization overhead during runtime.

Second, unlike `NettyBackend` which uses separate reactor threads to manage upstream and downstream connections (see Figure 8), `DoubleFaceAD` uses the same set of reactor threads to manage connections from both sides. Concretely, a reactor thread repeatedly loops over two phases: The first phase (event monitoring phase) determines a set of connections with pending events; the second phase (event handling phase) iterates over each of the connections that have pending events; the type of connections (either upstream or downstream) is determined based on the context associated with each event handler. Such design avoids the imbalanced workload problem between the frontend and the backend reactor threads.

We note that the integrated design of `DoubleFaceAD` does not necessarily sacrifice software maintenance flexibility for high performance. In `DoubleFaceAD` architecture, both business logic and datastore driver management are pluggable event handlers running on top of the same set of reactor threads. These handlers are logically divided into Frontend and Backend for the events from the upstream clients and the downstream datastores as shown in Figure 10. Thus, developers can independently upgrade either the Frontend business logic or the Backend connection management by changing the corresponding event handlers.

### 5.2 A Fanout-Query Aware Priority-based Scheduling

Modern web-facing applications typically have strict latency requirements due to its business impact [15, 17, 50]. For example, Instagram wants to reduce the $99.9th$ percentile response time of its production Cassandra datastore clusters within 20ms to provide a smooth experience for millions of Instagram users [19]. Here we introduce a fanout-query aware priority-based scheduling algorithm to minimize the tail response time of `DoubleFaceAD`-based servers.

Our scheduling algorithm is motivated by an observation from realistic workloads for distributed datastore clusters: the fanout queries to downstream datastores may not respond at the same time due to the variety of each shard.[4] On the other hand, the completion of a client request usually requires assembling all its fanout query responses before sending the final response to the client. Remember that a reactor thread in `DoubleFaceAD` processes all the fanout query responses (events) from different clients' requests in a batch mode after each event monitoring phase, so it is natural that only partial fanout queries for a specific client request get responses from the downstream datastores in the current batch due to the variety of each data shard. Thus the processing of incomplete fanout query responses from a specific client request may significantly delay the processing of fanout query responses from other client requests, causing overall long waiting time. Figure 11 illustrates this process.

The main idea of our fanout-query aware scheduling algorithm is to reduce the overall waiting time of client requests by first handling those fanout query responses that can assemble a complete response to a client request in the current batch, shown in Figure 12. For example, the client requests with fanout factor 3 and 8 can be finished in the current batch at timestamp $t_{req1}$ and $t_{req2}$, respectively. We schedule

[4] Dean et al. in his "the tail at scale" paper [15] reported a similar observation on the variety of VMs causing the tail latency in Google's clusters.

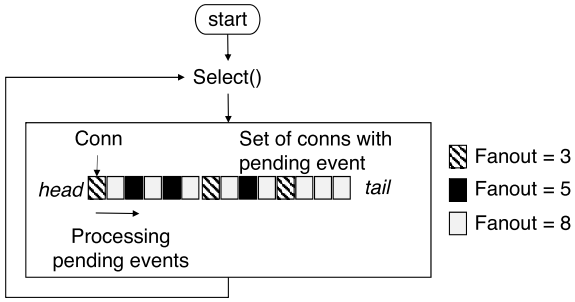



**Figure 11. The completion of the client request with fanout factor 3 is delayed by events of other uncompleted requests.**

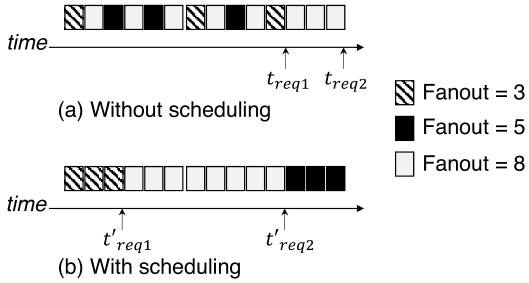



**Figure 12. Our fanout-query aware priority scheduling is able to minimize the average waiting time for all requests.**

the events of these two requests first. In this case, the completion time of such two requests is reduced to $t'_{req1}$ from $t_{req1}$ and $t'_{req2}$ from $t_{req2}$, respectively (see the striped and the grey boxes in Figure 12(b)). In the case of the client request with fanout factor 5 (black boxes), only partial fanout query responses (3 out of 5) are present, suggesting that this client request cannot be finished in the current batch. Thus we schedule the corresponding events the last.

In the next section, we experimentally show the effectiveness of our introduced DoubleFaceAD and the associated fanout-query aware priority-based scheduling algorithm.

## 6 Experimental Evaluation

To validate the effectiveness of our proposed DoubleFaceAD architecture, we compare the DoubleFaceAD-based server with aforementioned asynchronous AIOBackend (*Type-2a*) and NettyBackend (*Type-2b*) using various synthetic and real-life workload and datasets.

### 6.1 Experimental Results on YCSB Dataset

We start our evaluation on the synthetically generated YCSB [12] dataset (see Section 2.2 for details). We use two workload generators: JMeter [21] and RUBBoS [11]. JMeter is for stress testing that generates workload with precisely controlled workload concurrency for the target application server; the RUBBoS workload generator is to simulate real-world workload traffic (the request rate follows a Poisson

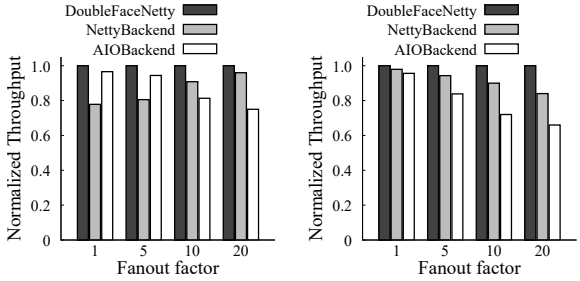



**Figure 13. Normalized throughput comparison among the three servers.** DoubleFaceNetty performs the best under various fanout factors and fanout query response sizes.

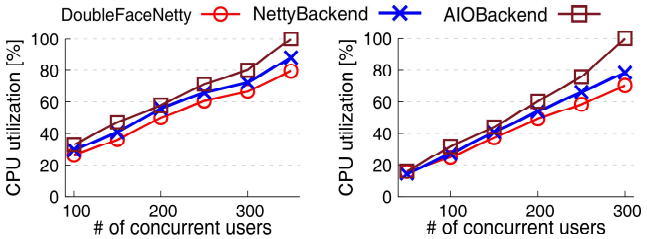
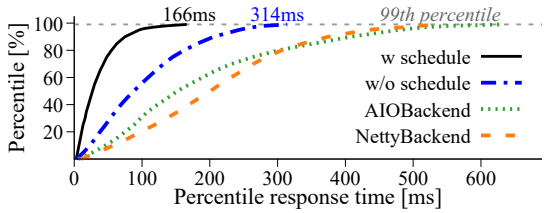


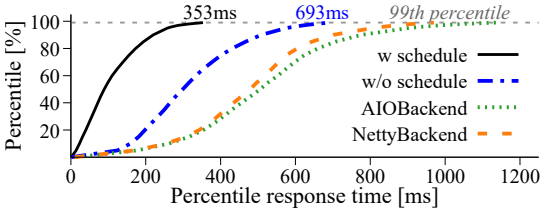

**Figure 14. Overhead analysis by comparing the CPU util. at the same workload.** The fanout factor keeps 20 and the fanout query response sizes are 0.1kB for (a) and 20kB for (b). DoubleFaceNetty has the lowest CPU overhead, especially at the high workload levels.

distribution with the mean determined by the number of end-users) to interact with the target application server.

We first use JMeter to validate the efficiency of our DoubleFaceAD architecture in terms of throughput. For a fair comparison, we implement a DoubleFaceAD-based server on top of the same asynchronous Netty framework as used in the other two servers (AIOBackend and NettyBackend), named DoubleFaceNetty. The client workload concurrency always keeps 20. We evaluate the impact of different fanout factors (from 1 to 20) and query response sizes (0.1kB vs. 20kB) on the performance of the three servers. Figure 13 shows that DoubleFaceNetty (the baseline) always achieves the best performance among the three servers under various workload conditions. For example, at 0.1kB query response size, Figure 13(a) shows DoubleFaceNetty achieves 20% higher throughput than NettyBackend when the fanout factor is 1, due to the avoidance of the imbalanced workload problem that causes significant *select()* system call overhead in NettyBackend (see Section 4). On the other head, as the fanout factor increases to 20, DoubleFaceNetty achieves 25% higher throughput than AIOBackend due to reduced multithreading overhead (see Section 3). When the query response size increases to 20kB and the fanout factor is 20, Figure 13(b) shows DoubleFaceNetty achieves 34% higher throughput




(a) The fanout factor of $Lfan$ and $Sfan$ requests is 5 and 3.




(b) The fanout factor of $Lfan$ and $Sfan$ requests is 7 and 1.

**Figure 15. Percentile response time comparison among different servers running on the YCSB dataset.** `DoubleFaceNetty` with scheduling can significantly reduce the tail response time compared to the other three cases.

than `AIOBackend` since large fanout query response size exaggerates the multithreading overhead in `AIOBackend`.

We then use the RUBBoS workload generator to evaluate the CPU overhead of `DoubleFaceNetty` compared to other servers (`AIOBackend` and `NettyBackend`). We set the fanout factor to 20 in all cases and evaluate the overhead over two fanout query response sizes: 0.1kB vs. 20kB. Figure 14 shows that `DoubleFaceNetty` always has the lowest CPU utilization among the three servers at different levels of concurrent users. For example, at 350 of concurrent users workload in Figure 14(a), the `DoubleFaceNetty` CPU utilization is 80%, which is 20% and 8% less than that in `AIOBackend` and `NettyBackend`, respectively. As the fanout query response size increases to 20kB, Figure 14(b) shows that `DoubleFaceNetty` consumes 30% less CPU utilization than that in `AIOBackend` at 300 of concurrent users workload due to the exaggerated multithreading overhead in `AIOBackend`. This set of experiments show that `DoubleFaceAD` can mitigate or avoid both multithreading overhead and the imbalanced workload problem when handling highly concurrent fanout queries.

We further use the RUBBoS workload generator to validate the effectiveness of our fanout-query aware priority-based scheduling algorithm on mitigating the tail response time. The generated HTTP requests can be categorized into two classes: the $Lfan$ requests with a large fanout factor (e.g., 5) and the $Sfan$ requests with a small fanout factor (e.g., 3). The percentage of $Lfan$ and $Sfan$ requests are 50/50. We use the same level of workload (i.e., 2500 concurrent users) to test all the four servers (`AIOBackend`, `NettyBackend`, `DoubleFaceNetty` with or without scheduling). These servers produce the same amount of throughput (around 230 reqs/sec) but at different levels of CPU utilization (i.e., from 70% to 90%) due to the variety of CPU overhead.

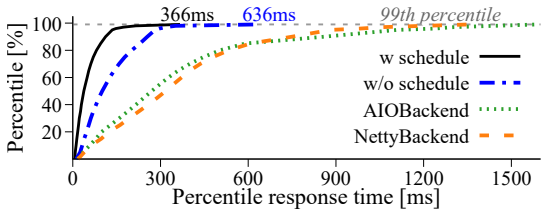



**Figure 16. Percentile response time comparison among different servers running on the YCSB dataset with large data shard size** (i.e., ×10 times the size of that in Figure 15(a)). The fanout factor of $Lfan$ and $Sfan$ requests is 5 and 3. `DoubleFaceNetty` with scheduling still outperforms the other three cases in tail latency.

Figure 15(a) shows the percentile response time comparison among four servers when the fanout factor for $Lfan$ and $Sfan$ requests are 5 and 3, respectively. This figure shows `DoubleFaceNetty` with scheduling achieves significantly lower tail response time than the other three servers. For example, the 99$th$ percentile response time of `DoubleFaceNetty` with and without scheduling is 166ms and 314ms, respectively, showing a more than 1.9× speedup. The speedup is even greater when compared with `NettyBackend` and `AIOBackend` (540ms and 630ms). Figure 15(b) shows a similar percentile response time comparison but with a wider fanout factor gap between $Lfan$ and $Sfan$ requests (7 and 1, respectively). We observed that the gap of percentile response time between `DoubleFaceNetty` with and without scheduling becomes wider in Figure 15(b) than in Figure 15(a) (compare the solid black and the dashed blue lines), suggesting our scheduling algorithm can achieve better performance improvement when the fanout factor gap between requests becomes larger.

To evaluate the impact of large shard size of datastores on our scheduling algorithm, Figure 16 shows the percentile response time comparison among the four servers when we increase the shard size of each datastore from the original 1GB to 10GB. Each shard still fits into the memory and the average fanout query response size keeps the same as that in Figure 15(a). The only difference is that the average response time from the downstream datastores to the upstream datastore drivers increases from 0.12ms to 0.18ms. We can see that `DoubleFaceNetty` with scheduling still outperforms the other three servers in tail latency, suggesting the robustness of our scheduling algorithm.

### 6.2 Experimental Results on DBLP Dataset

To test how sensitive of `DoubleFaceNetty` on the variety of dataset, we further validate the effectiveness of our scheduling algorithm on a real-life DBLP dataset [31], where the dataset contains more than 7M co-author pairs (tuples), each tuple is about 30kB, significantly larger than each record in the previous synthetic YCSB dataset. The DBLP dataset is evenly distributed among the 20 downstream MongoDB servers. So each shard is about 20GB. We use the RUBBoS

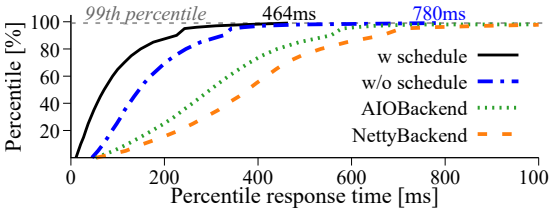



**Figure 17. Percentile response time comparison among different servers running on the DBLP dataset.**

workload generator to generate a similar workload as described in Section 6.1.

Figure 17 compares the percentile response time of the same four servers as before but on the new DBLP dataset. The fanout factor for $Lfan$ and $Sfan$ requests are set to 5 and 3, respectively. We observe that the performance gain of our scheduling algorithm on `DoubleFaceNetty` is less than that in the YCSB dataset case by comparing Figures 15(a) and 17. This is because the response of fanout queries on the DBLP dataset is much heavier than that on the YCSB dataset (30kB vs. 0.1kB), which reduces the weight of performance improvement led by our scheduling algorithm. On the other hand, we observed the tail latency of `AIOBackend` is better than that in `NettyBackend`, which is due to the increased response size of fanout queries, causing non-trivial multithreading overhead in `AIOBackend` (see Section 3).

## 7 Discussion

From a system performance scalability point of view, event-based implementations such as Nginx web server [35] are often considered to have more stable scalability at high levels of concurrency when compared to thread-based counterparts such as Apache. This performance scalability advantage is often attributed to lower multithreading overhead of asynchronous event-driven servers [14, 28, 37, 51, 55].In our experimental studies (Figure 4, as well as subsequent figures and tables), we have confirmed the limitations of thread-based implementations and better scalability of asynchronous designs. Furthermore, we found that some asynchronous approaches are more scalable than others for workloads consisting of fan-out queries in web-facing applications.

Our experiments showed that the performance of different asynchronous approaches varied in non-obvious ways when workload variables change. Two important factors that cause performance problems are: 1) fanout query parallelism and 2) fanout query response size. The first class of performance problems consists of unexpected multithreading overhead problem for *Type-2b* asynchronous drivers (Section 3). For example, `AIOBackend` incurs higher overhead than `NettyBackend` (Table 1 at high query response size (20kB)). However, the situation is reversed at low response size (0.1kB), when `NettyBackend` consumes more CPU due to unnecessary select syscalls (Table 2). These factors also interact with each other: at 20kB response size, `NettyBackend` has better scalability when fanout query parallelism increases (Figure 7), but `AIOBackend` has better scalability at 0.1kB response size (Figure 5).

The second class of performance problems consists of workload imbalance between Frontend (app server) and Backend (datastore driver), resulting in less efficient select syscalls for *Type-2a* asynchronous drivers (Section 4). Concretely, an improper ratio of frontend and backend reactor threads in `NettyBackend` (Table 3) can cause significant throughput drop due to inefficient execution of syscalls. Since the execution of frontend and backend reactor threads vary independently, both factors 1 (fanout query parallelism) and 2 (fanout query response size) can cause workload imbalance due to fixed allocation of reactor threads. This problem appears to be more complex than a parameter tuning issue. For example, Figure 13(a) shows a good balance between Frontend and Backend when increasing fanout query parallelism from 1 to 20 in `NettyBackend` for response size 0.1kB, but imbalance for response size 20kB (comparing `NettyBackend` and `DoubleFaceNetty` throughput in Figure 13(b)). We note that `AIOBackend` addresses this problem by adopting an on-demand worker thread pool in Backend.

`DoubleFaceAD` design is able to reduce the multi-threading overhead by combining the threads that handle both Frontend and Backend (Section 5). In addition, `DoubleFaceAD`'s combination of threads is able to avoid the imbalance between Frontend and Backend and the associated inefficient syscalls. Figure 13 shows that `DoubleFaceAD` achieves the highest throughput when compared to `NettyBackend` and `AIOBackend` in a variety of fanout factors (from 1 to 20), and query response sizes of 0.1kB and 20kB.

Our experimental results indicate that the two factors shown in Table 5 appear to contribute to the lion's share of performance variability of asynchronous approaches. Currently, we are investigating other factors that may have potential impact on the performance scalability of asynchronous drivers. For example, Figure 16 shows preliminary results of shard size (1GB vs. 10GB) on query response time tail latency, where `DoubleFaceAD` continues to achieve the best response time tail latency. Another factor with potential impact is the computation intensity of business logic, which may affect the overall system performance since it changes the CPU load of Frontend and may impact the workload (im)balance between Frontend and Backend of the asynchronous server. More generally, the interactions between CPU load and overall throughput and response time will be a subject of future research.

## 8 Related Work

Previous research on improving asynchronous datastore driver performance can be categorized into three classes:

**Optimizing OS support** mainly focuses on mitigating system call overhead [30] and simplifying the application program interfaces (APIs) of network I/O for asynchronous programming [24]. For example, TUX [30] integrates both

**Table 5. Comparison of performance degradation in synchronous approaches due to two important factors: imbalanced workload and multi-threading overhead.**

| Factors | NettyBackend | AIOBackend | Thread-based | DoubleFaceAD |
|---|---|---|---|---|
| Fanout factor size (i.e., fanout query parallelism) | Imbalanced workload problem (e.g., Figure 13(b)) | Unexpected multithreading overhead problem (e.g., see Figure 7) | Multithreading overhead problem | Avoid both problems (e.g., see Figure 13) |
| Fanout query response size | Imbalanced workload problem (e.g.,comparing Figure 13(a) and 13(b)) | Unexpected multithreading overhead problem (e.g., see Figure 5) | Multithreading overhead problem | Avoid both problems (e.g., see Figure 13) |

the event monitoring and event handling into the kernel space to avoid kernel-user space crossing overhead caused by system calls, but sacrifices the development flexibility of business logic, which usually runs in the userspace. Han et al. [24] present MegaPipe, a new API for efficient and scalable network I/O for message-oriented workloads such as key-value stores, which is intended to provide lightweight sockets for asynchronous event-driven programming.

**Tuning software configurations** to improve the performance of asynchronous Internet servers has also been studied before. For example, Pariag et al. [37] show that carefully tuning the number of concurrent established TCP connections and exploiting the non-blocking *sendfile()* system call can improve the maximum achievable throughput of an asynchronous event-driven $\mu$Server. Brecht et al. [9] further improve the performance of $\mu$Server by modifying the accepting strategy of new connections based on different workload characteristics.

**Applying novel scheduling algorithms** to reduce tail latencies in distributed datastores when handling highly concurrent has been studied recently [7, 10, 18, 40]. For example, Rein [40] identifies the bottleneck fanout queries (the slowest one) and schedules the smaller-bottleneck fanout queries first to reduce head-of-line blocking and improve tail latency. Cho et al. [10] implement *Moolle* in the production distributed graph service at LinkedIn, which schedules the optimal number of fanout queries per request (i.e., the fanout factor) by removing the duplicate and the unnecessary fanout queries to curtail tail latency.

To our best knowledge, the development of asynchronous servers and datastore drivers so far are independent of each other. While separating the two sides may follow the classic modular design principle in software engineering, it also brings significant overhead under recent new workload scenarios such as fanout queries on distributed datastores and microservices applications. Taking advantage of the merits of asynchronous design on both sides, our DoubleFaceAD provides an integrated design of an application server and a datastore driver as a whole package while maintains the flexibility of application-level business logic development.

## 9 Conclusion

Compared to thread-based servers with high throughput and short response time requirements (e.g., in web-facing applications), event-based design and implementations are considered to have better performance scalability at high levels of concurrency. Our experimental studies on some of the latest datastore drivers (e.g., DynamoDB, HBase, and MongoDB) confirm the limitations of thread-based implementations. Furthermore, we found unexpected performance degradations in some asynchronous approaches, including multi-threading overhead and unnecessary syscalls.

Our experiments show that two factors cause the performance degradation of asynchronous approaches. First, *Type-2b* asynchronous drivers (e.g., AIOBackend) that adopt thread pools incur non-trivial multithreading overhead when the fanout query response size becomes large (Section 3). Second, *Type-2a* asynchronous drivers (e.g., NettyBackend) that adopt event queues may suffer from imbalanced workload between Frontend and Backend, causing unnecessary system calls (Section 4), when the query parallelism and response sizes vary.

We introduce the DoubleFaceAD approach, with a fanout-query-aware priority-based scheduling algorithm (Section 5), and a few threads that handle both Frontend and Backend events. Experiments show that DoubleFaceAD is able to reduce multi-threading overhead and avoid unnecessary syscalls. Further experiments (Section 6) also show that DoubleFaceAD reduces the response time long tail problems that plague many web-facing servers. Ongoing efforts to study the potential impact of other factors such as shard size and cost of business logic indicate that DoubleFaceAD may be generally applicable in emerging high performance and high efficiency middleware based on microservices to support serverless computing.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Amy Tai, for their feedback on improving this paper. This research has been partially funded by National Science Foundation by CNS (2000681), RCN (1550379), CRISP (1541074), SaTC (1564097) programs, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies mentioned above.

## References


[1] 2018. Node.js. https://nodejs.org/. Accessed: 2018-05-07.

[2] Atul Adya, William J Bolosky, Gerald Cermak, John R Douceur, Marvin M Theimer, and Roger P Wattenhofer. 2006. Serverless distributed file system. US Patent 7,062,490.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[4] AWS. 2019. AWS SDK for DynamoDB. https://github.com/aws/aws-sdk-java/tree/master/aws-java-sdk-dynamodb. Accessed: 2019-05-05.

[5] Azure. 2018. Java SDK for SQL API of Azure Cosmos DB. https://github.com/Azure/azure-cosmosdb-java. Accessed: 2018-03-05.

[6] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[7] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 195–212.

[8] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, et al. 2008. Corey: An Operating System for Many Cores.. In *OSDI*, Vol. 8. 43–57.

[9] Tim Brecht, David Pariag, and Louay Gammo. 2004. accept () able Strategies for Improving Web Server Performance.. In *USENIX Annual Technical Conference, General Track*. 227–240.

[10] SungJu Cho, Andrew Carter, Joshua Ehrlich, and Jane Alam Jan. 2016. Moolle: Fan-out control for scalable distributed data stores. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 1206–1217.

[11] Object Consortium. 2005. *RUBBoS benchmark*. Accessed: 2018-05-07.

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

[13] Datastax. 2018. Datastax java driver for apache cassandra. https://github.com/datastax/java-driver. Accessed: 2018-03-05.

[14] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 145–160.

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[17] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (2018), 65–72.

[18] Diego Didona and Willy Zwaenepoel. 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 79–94.

[19] Instagram Engineering. 2018. Who moved my 99th percentile latency? https://instagram-engineering.com/open-sourcing-a-10x-reduction-in-apache-cassandra-tail-latency-d64f86b43589. Accessed: 2018-03-05.

[20] LinkedIn Engineering. 2015. Open-sourcing a 10x reduction in Apache Cassandra tail latency. https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency. Accessed: 2018-03-05.

[21] Apache Software Foundation. 2018. Apache JMeter™. http://jmeter.apache.org. Accessed: 2018-03-05.

[22] The Apache Software Foundation. 2019. Apache HBase Java Client. https://github.com/apache/hbase/tree/master/hbase-client. Accessed: 2019-05-05.

[23] Brendan D. Gregg. 2018. *perf*. Accessed: 2018-05-07.

[24] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O.. In *OSDI*, Vol. 12. 135–148.

[25] Md E Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S McKinley. 2015. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *ACM SIGPLAN Notices* 50, 4 (2015), 161–175.

[26] Arun Iyengar. 2017. Providing Enhanced Functionality for Data Store Clients. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 1237–1248.

[27] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 129–141.

[28] Maxwell N Krohn, Eddie Kohler, and M Frans Kaashoek. 2007. Events Can Make Sense.. In *USENIX Annual Technical Conference*. 87–100.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[30] Chuck Lever, Marius Aamodt Eriksen, and Stephen P Molloy. 2000. *An analysis of the TUX web server*. Technical Report. Center for Information Technology Integration.

[31] Michael Ley. 2009. DBLP: some lessons learned. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1493–1500.

[32] Chuanpeng Li, Kai Shen, and Athanasios E Papathanasiou. 2007. Competitive prefetching for concurrent sequential I/O. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 189–202.

[33] Lighttpd. 2018. lighttpd: fly light. https://www.lighttpd.net/. Accessed: 2018-05-07.

[34] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. 2019. JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms. In *International Conference on Cloud Computing*. Springer, 287–302.

[35] NGINX. 2018. nginx. https://nginx.org/en/. Accessed: 2018-05-07.

[36] OpenJDK. 2018. The HotSpot Group. http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2018-03-05.

[37] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R Cheriton. 2007. Comparing the performance of web server architectures. In *ACM SIGOPS Operating Systems Review*, Vol. 41. 231–243.

[38] The Netty Project. 2018. Netty. http://netty.io/. Accessed: 2018-03-05.

[39] The Netty project. 2019. Netty project: Adopters. https://netty.io/wiki/adopters.html. Accessed: 2018-03-05.

[40] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostic, and Sean Braithwaite. 2017. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 95–110.

[41] reddison. 2018. Reddison: State of the Art Redis client. https://github.com/redisson/redisson. Accessed: 2018-03-05.

[42] Amazon Web Services. 2018. Amazon DynamoDB: Fast and flexible NoSQL database service for any scale. https://aws.amazon.com/dynamodb/. Accessed: 2019-05-13.

[43] N. B. Shah, K. Lee, and K. Ramchandran. 2016. When Do Redundant Requests Reduce Latency? *IEEE Transactions on Communications* 64, 2 (Feb 2016), 715–722.

[44] Huasong Shan, Qingyang Wang, and Calton Pu. 2017. Tail attacks on web applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1725–1739.

[45] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.. In *Osdi*, Vol. 10. 1–8.

[46] Akshitha Sriraman and Thomas F Wenisch. 2018. $\mu$Tune: Auto-Tuned Threading for {OLDI} Microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 177–194.

[47] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. 2003. Why Events Are a Bad Idea (for High-Concurrency Servers).. In *HotOS*. 19–24.

[48] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 133–146.

[49] Qingyang Wang, Hui Chen, Shungeng Zhang, Liting Hu, and Balaji Palanisamy. 2019. Integrating Concurrency Control in n-Tier Application Scaling Management in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 855–869.

[50] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. 2017. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 207–217.

[51] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[52] Mindi Yuan, David Stein, Berenice Carrasco, Joana MF Trindade, and Yi Lu. 2012. Partitioning social networks for fast retrieval of time-dependent queries. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. IEEE, 205–212.

[53] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazieres, and M Frans Kaashoek. 2003. Multiprocessor Support for Event-Driven Programs.. In *USENIX Annual Technical Conference*. 239–252.

[54] Shungeng Zhang. 2018. Application server implementations. https://github.com/sgzhang/AsyncFanout. Accessed: 2019-03-05.

[55] Shungeng Zhang, Qingyang Wang, and Yasuhiko Kanemas. 2018. Improving asynchronous invocation performance in client-server systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 907–917.