How can Highly Concurrent Network-Bound Applications benefit from modern multi-core CPUs?

by

Lucas Crämer

Submitted to the Fakultät Print und Medien
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Mobile Media

at the

HOCHSCHULE DER MEDIEN - STUTTGART

February 2021

Author .....
Fakultät Print und Medien
February 28, 2021

Certified by .....
Prof. Walter Kriha
Erstprüfer
Hochschule der Medien

Certified by .....
Dr. Thomas Fankhauser
Zweitprüfer
Hochschule der Medien

Accepted by .....
Sibel Aktikkalmaz
Prüfungsverwaltung Hochschule der Medien

Erklärung

Hiermit versichere ich, Lucas Crämer, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: "How can Highly Concurrent Network-Bound Applications benefit from modern multi-core CPUs?" (Forschungsfrage: "Wie können netzwerkgebundene Anwendungen von modernen Mehrkernprozessoren profitieren?") selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.
Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO (7 Semester) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Ort, Datum	Unterschrift

Kurzfassung

Netzwerkgebundene Anwendungen wie Web Server und in-memory Datenbanken bilden das Fundament für moderne stark verteilte Systeme. Die Leistung von Netzwerkkarten und der dazugehörigen Infrastruktur steigt jährlich, indessen nimmt die Leistungszunahme von modernen Prozessoren pro Kern weiter ab. Um dieser Entwicklung entgegenzuwirken müssen diese Anwendungen horizontal und vertikal skalierbar sein. Parallele Algorithmen ermöglichen vertikale Skalierung auf modernen Mehrkern-Prozessoren. Diese Thesis betrachtet wie netzwerkgebundene Anwendungen auf modernen Mehrkern-Prozessoren skalieren können. Die dabei behandelten Themen reichen von Hardware und Betriebssystemen bis hin zu Algorithmen für dynamisches "Scheduling" im "User Space".

Als ein wesentlicher Teil dieser Thesis wurden Benchmark-Tests für die in-memory Datenbanken Redis (mit "I/O Threading"), KeyDB ("Redis fork" mit einer "multithreaded Event Loop"), Mini-Redis (unvollständiger Redis Server, der von "Work-Stealing Scheduling" Gebrauch macht) und Redis Cluster ("shared-nothing" Datenbank-Cluster) durchgeführt. Alle getesteten Datenbanken liefen auf einem (physischen) Server. Die Performance-Metriken waren der absolute Durchsatz und "Tail Latency". In diesen Benchmarks erzielt die "shared-nothing" Datenbank Redis Cluster den höchsten Durchsatz aller getesteten Anwendungen. Im Allgemeinen weisen die Tests darauf hin, dass je weniger Daten unter Prozessen geteilt werden und je weniger Mehraufwand die Parallelverarbeitung verursacht, desto höher ist der Durchsatz den diese Anwendungen bei homogener Last maximal erzielen können. Auf der anderen Seite zeigt diese Arbeit aber auch auf, dass das "shared-nothing" Paradigma, abgesehen von anderer Nachteilen, womöglich nicht die optimalste Strategie für die Senkung von "Tail Latencies" ist. Das Teilen von Daten zwischen Prozessen, zum Beispiel durch die Verwendung von "Single-Queue, Multi-Server" Queueing-Modellen und dynamischen "Scheduling" Strategien, kann bezüglich der "Tail Latencies" trotz Mehraufwand bessere Resultate erzielen, vor allem bei "skewed Workloads".

How can Highly Concurrent Network-Bound Applications benefit from modern multi-core CPUs?

by

Lucas Crämer

Submitted to the Fakultät Print und Medien
on February 28, 2021, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Mobile Media

Abstract

Highly concurrent network-bound applications like web servers and in-memory databases are the foundation of modern large scale distributed systems. While the performance of NICs and network infrastructure increases every year, improvements in the single-core performance of modern server CPUs seem to have plummeted in the past years. Highly concurrent network-bound applications need to counteract by scaling horizontally **and** vertically. One way to scale vertically in these applications is to implement parallel processing for multi-core scaling on modern CPUs. This thesis takes a look at how this can be achieved and the topics discussed range from hardware and operating system related issues to algorithms for dynamic scheduling in user space.
As part of this study benchmark tests were performed with the in-memory databases Redis (with I/O threading), KeyDB (Redis fork with a multithreaded event loop), Mini-Redis (incomplete Redis server that leverages work-stealing scheduling) and Redis Cluster (shared-nothing database cluster), which all ran on a single (physical) node. In the benchmarks the performance metrics tested were throughput and tail latency. The evaluated results demonstrate that the shared-nothing database Redis Cluster delivers the best results for throughput of all the tested applications. Generally, the tests indicate that the less data is shared in-between processes and the less overhead the algorithms for parallel processing introduce, the better the performance in regards to throughput in uniform load scenarios. However, this study also demonstrates that the shared-nothing approach, besides having other disadvantages, might not be the optimal strategy for lowering tail latencies. Sharing data in-between processes, e.g. by leveraging single-queue, multi-server models and dynamic scheduling strategies, which typically add overhead, can deliver better results in regards to tail latencies, especially when workloads are skewed.

Thesis Supervisor: Prof. Walter Kriha
Role: Erstprüfer

Thesis Supervisor: Dr. Thomas Fankhauser
Role: Zweitprüfer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past five years (January 2016 - January 2021) the average network traffic at the DE-CIX (Deutscher Commercial Internet Exchange) has more than doubled. Common microservice and fan-out architectures are growing more sophisticated and accumulate increasingly more data streams, worsening the effect of high tail latencies. Applications counteract by scaling horizontally and vertically. Average single core performance of CPUs did not increase as much in comparison. Software and hardware patches against side channel attacks like Meltdown and Spectre have contributed to further worsen this gap in the past years. However, the core count of CPUs monotonically increases every year.

For example, in 2019 AMD launched their second generation "EPYC" Server CPU series that offers up to 64 cores and 128 SMT "threads". In 2020 Ampere Computing introduced the "Ultra", an ARM based Server CPU targeted at cloud computing environments, with 80 CPU cores. In both cases the CPU core counts are per CPU socket, which means that in multi socket server systems the total CPU core count might even be a multiple of that.

Mid to high-end Amazon EC2 instances, such as the M5n and better, offer network bandwidths from 25 Gbps (burst) to 100 Gbps, thus making the network bandwidth disappear as the first bottleneck in the system for single-threaded highly concurrent network-bound applications in lots of scenarios.

One popular example is the open-source in-memory database Redis whose main logic was strictly single-threaded until Redis Version 6, which was released in 2020. This led to forks by cloud providers such as "Alibaba Cloud" and "Amazon Web Services" [9, 68] in 2018 and 2019 respectively and an open-source fork "KeyDB". These forks incorporate thread-level parallelism into Redis' main logic.

According to the Redis developers the preferred way for scaling Redis on a single node is the

strict shared-nothing approach with a partitioned Redis Cluster.

This thesis is going to demonstrate that one CPU core does not offer enough performance in certain cases to fully utilize the bandwidth provided by modern Network Interface Controllers (NICs), effectively bottlenecking the system. Therefore, highly concurrent network-bound applications need ways to utilize these cores efficiently for scaling vertically on a node.

In this thesis highly concurrent network-bound applications are defined as applications that establish multiple concurrent communication channels for end-to-end bidirectional communication with the goal of making and/or serving requests as performantly as possible. Common performance metrics are throughput and response times. These highly concurrent network-bound applications are typically driven by events in the network such as incoming HTTP requests, remote procedure calls (RPCs) or database queries, which work at the core of modern large scale distributed systems.

In essence, this thesis is supposed to prove that highly concurrent network-bound applications can benefit from modern multi-core CPUs under certain circumstances. It evaluates different architectures and paradigms for concurrency and parallelism in such applications based on research and benchmark experiments.

Highly concurrent network-bound applications are tightly integrated with lower level software stacks and the underlying hardware, so chapter 2 of this study elaborates on the current state of hardware and Linux that is relevant for network applications. The chapter takes a "deep dive" into event notification interfaces, the Linux network stack, NIC hardware, Data Plane Processing, lower level concurrent data structures and building blocks for state of the art synchronization primitives. It serves as the basis for reflecting about how to get the most performance out of the hardware and APIs provided by the operating system from a user space application developer's perspective.

Chapter 3 examines practical architectures and paradigms for concurrency and parallelism in network applications. It discusses the essence behind the shared-nothing and shared-something approaches and their implications on highly concurrent network-bound applications. Furthermore, the chapter addresses work distribution and scheduling strategies in the implementations of established and upcoming highly concurrent network applications and library abstractions, such as Redis version 6 with I/O threading, KeyDB, NGINX, libuv, Tokio and Glommio. Another section is devoted to Rust Futures and zero-cost async-await because it enables so-called "green threading" - a paradigm that is common in higher level concurrent programming. Rust futures make this paradigm easy to use for highly concurrent "low level"

implementations, while not sacrificing any performance. Rust futures are essential for Tokio and Glommio. SIMD is another important paradigm for parallelism on modern CPUs. So one section takes a quick glance at parts of network applications that can possibly benefit from SIMD.

A common belief in the developer communities is that in-memory databases are prone to be completely limited by the network bandwidth, which implies that there is no need for parallelism in the main logic. Chapter 4 and 5 about performance evaluations test whether this belief is true by using quantitative benchmark evaluations. These benchmarks test how some of the in-memory database solutions, which are discussed in detail in the chapter about architectures & paradigms, perform in regards to throughput and response time percentiles under high load in a variety of scenarios. The benchmark evaluations serve as the basis for verifying or refuting some of the research which was conducted on different approaches to concurrency & parallelism in highly concurrent network-bound applications.

Finally, the conclusion summarizes key findings with regard to the research question that I came across while researching for this thesis by discussing core principles when developing highly concurrent network-bound applications.

Due to the practice-oriented and recent nature of the research question, it is to be expected that this thesis also references lots of grey literature and blog posts among other more reliable sources.

Chapter 2

Current State of Linux and Hardware for Highly Concurrent Network-Bound Applications

In order to get the best performance out of modern CPUs in highly concurrent network-bound applications, it is important to take a look at how related APIs are handled on modern operating systems and where they historically came from. Since it is free, open-source and the predominant force in modern cloud infrastructure, this section focuses on the operating system GNU/Linux. However, some information given in this context also applies to other UNIX-like operating systems, but taking all of them into account is beyond the scope of this study.

The first and the biggest section in this chapter captures the current state of networking related APIs, such as “epoll”, “io_uring” and “UIO” in Linux, and discusses their implications on performance.

The second section picks up where the first subsection left off and serves as a quick introduction to paradigms for concurrency & parallelism within the network stack (data plane processing).

Security patches against side-channel attacks, such as Meltdown and Spectre, have a significant impact on applications that make frequent use of system calls. This is why the third section discusses the impact of these side-channel attacks on highly concurrent network-bound applications.

Concurrent queues are essential to the implementation of most highly concurrent network-

bound applications. So the fourth section discusses the “low level” implementation of a lock-free single-producer, single-consumer and a lock-free multiple-producer, multiple-consumer queue. Lock-free queues rely on atomic operations and so-called “memory barriers” to function correctly. As a consequence, the section also explains the implementation of these atomic operations and memory barriers on state of the art instruction set architectures.
Lock-free implementations are not viable in every scenario. When locking is required, it should be implemented as efficiently as possible. The last section in this chapter explains efficient “user space locking” with the Linux system call “futex”.

2.1 Concurrent Connections - From select to io_uring and DPDK

2.1.1 select, non-blocking and the Reactor Pattern

Linux like almost all modern operating systems ships with APIs for TCP and UDP networking, so-called socket APIs. This provides an abstraction over the concrete network stack and the hardware. The idea is that applications should not have to implement a TCP/IP stack or know how to communicate with every available networking controller. Networking in kernel space also provides memory protection and the kernel is able to handle resource allocation for multiple processes that need concurrent access to the network.
Within the Linux Kernel resides the networking subsystem. The kernel provides a set of well-defined system calls to enable networking functionality within a user space application. UNIX-like and POSIX-compliant operating systems typically implement the BSD sockets API for networking. As seen from the perspective of a server the basic operating principle of this API is to bind a socket, that is the combination of an IP address and a transport layer port, to a local socket address, then listen on that socket and accept incoming connections. However, this does not make multiple concurrent connections possible.
To achieve multiple concurrent connections, a naive approach is to assign a thread or a process to each new accepted connection, but implementing massive networking concurrency using thread-level parallelism is considered to be a pitfall with today’s systems [46, p. 1245][45]. The overhead of context switching between the threads results in bad performance once a certain amount of concurrent connections is reached. This is explained in greater detail in the chapter 3 about architectures and paradigms.
In this “thread-per-connection” approach the sockets are in blocking mode. This is also the

default behavior for networking sockets. Any operation on a blocking socket can theoretically “block” at any time. “Blocking” in this context means that in the case of reading, for example, the kernel buffer is not ready to be read from because no data has arrived at the socket yet. In the case of writing it means that the kernel buffer is not ready to be written to. The kernel effectively blocks the thread until the intended operation is ready. On highly concurrent systems this yields bad performance and might even impact the liveness of the system. If on the other hand this socket is in non-blocking mode, the intended operation returns immediately with an error and the value of “errno” is set to “EAGAIN” or “EWOULDBLOCK”. The “O_NONBLOCK” mask that is used to set descriptors which identify sockets in non-blocking mode, has no effect on descriptors identifying regular files. Now, another rather naive approach would be to put these file descriptors in non-blocking mode and poll each of them on a single thread until it is ready to perform the intended I/O operation. This solves the problem of the context switch overhead, but would just waste CPU cycles in many cases.[1]

To mitigate the need for applications polling manually on descriptors, modern operating systems usually ship with multiple I/O notification strategies. An example for an I/O notification selector is the select system call in UNIX-like and POSIX-compliant operating systems, whose history goes back to the 1980s. Nowadays, the select system call is considered to be deprecated for multiple reasons that are not the main subject of this thesis, but the basic interface has remained very similar in more modern event notification interfaces like "epoll" or "kqueue":

In “selectish” event notification interfaces the application registers its interest in particular events that it wants to receive notifications of with the associated file descriptors. For example, in the case of “select” an application can register a read interest with a file descriptor identifying a socket by adding it to the specific set for this event. There is a set of file descriptors for read, write and error events. The application needs to pass these sets of file descriptors as an argument to the select system call. The select system call either returns if at least one of the file descriptors becomes “ready”, which means that the operation associated with the interest can be performed without blocking, or if the specified timeout is exceeded. In order to notify the application of their readiness the sets of file descriptors are mutated in-place by the kernel. This is one of the many pitfalls of the classic select system call. They

[1] While manually polling file descriptors might not be efficient, polling for data in other cases, such as within drivers for highly performant I/O devices, can be more efficient than asynchronous interrupt based approaches as it is be explained later in the section on network drivers.

then need to be checked by the caller in user space for events. A “reactor style” application handles these events by calling a callback function that was registered earlier and is owned by each handle (file descriptor), a so-called Event Handler. The entity in the program which calls “select” (Synchronous Event Demultiplexer) and handles the events is called Initiation Dispatcher. The application typically repeats this process in a loop and this is called an event loop. One iteration is called an event loop tick [67].
The select system call has many problems. One of them, the in-place mutation of the “fd_set”s, was already mentioned earlier. It forces the application to always copy the “fd_set”s before passing them to the “select” call. Moreover, when using the “glibc wrapper” for “select”, an “fd_set” can only hold a maximum of 1024 file descriptors and no descriptor can have a greater value than 1023 [21].
In comparison to “select” the poll system call improves the ergonomics by fixing issues surrounding “fd_set”, otherwise its functionality is equivalent [20].

2.1.2 epoll

“Epoll” is the further evolution of the select and poll system call on Linux. “Select” and “poll” are stateless, which means that the caller passes the file descriptors of interest with each new call and the kernel then checks each file descriptor for readiness concerning the specified events. This leads to a time complexity of *O(number of descriptors in the passed interest set)* for the select and the poll system call. This is inefficient, especially when a server has lots of concurrent connections and only very few events happen on these connections.
“Epoll” on the other hand is stateful. Before calling “epoll_wait”, the application sets up an “epoll” instance with “epoll_create”. This “epoll” instance is a kernel data structure which holds the associated interest and ready list. File descriptors can be added to the interest list by calling the “epoll_ctl” system call with the event masks that the application wants to be notified of passed as arguments. When an event of interest occurs on one of the monitored descriptors in the interest list, the Linux kernel adds the descriptor to the ready list. When the application calls “epoll_wait”, the ready list is returned to the caller. This results in “epoll” having a time complexity of *O(number of “ready” descriptors)*. It is the reason why “epoll” is generally more efficient than “select” and “poll”, especially in highly concurrent environments.
Level-triggered “epoll” shares the same semantics with “poll” and is the default. As an alter-

native to level-triggered “epoll” notifications, edge-triggered “epoll” notifications[2] are delivered only when events of interest occur on the monitored descriptor. That means in practice, a call to “epoll_wait” only notifies the caller(s) about a descriptor being ready once. After that, provided that no other events take place, there are no further notifications, even if there is still data left and ready to be read on that particular descriptor. Edge-triggered “epoll” can be more efficient than level-triggered “epoll”. It is recommended to use edge-triggered “epoll” with non-blocking descriptors and read from and write to a ready descriptor until it returns with “errno” set to “EAGAIN” [15]. The downside of this recommendation is that big reads or writes could “starve” other registered descriptors. However, in a multi-threaded context, edge-triggered “epoll” is particularly attractive because it avoids “thundering herd” wake-ups of multiple threads which are blocked by “epoll_wait”. When an edge-triggered descriptor becomes ready, only one of the blocked threads is “woken up”.

2.1.3 io_submit

The Linux kernel supports asynchronous I/O notifications, using the “io_submit” system call family. This API was originally created for asynchronous (unbuffered) file I/O and does not offer truly asynchronous networking I/O, but strangely enough, it does support stateful I/O notifications on sockets similar to “epoll” since Linux kernel version 4.18. “io_submit” supports adding multiple file descriptors with events of interest to the interest list by making only one system call similar to the kevent system call on BSD and Darwin. This is an advantage over “epoll” that could in theory reduce system call overhead.
“io_submit” does also allow system call batching for sockets. The application can add multiple read/write requests to a list and then pass this list to “io_submit”. The read/write requests are executed synchronously, requiring only one context switch to kernel space, which could reduce system call overhead even further [17, 52]. If the sockets passed to “io_submit” are non-blocking the process returns to user space after performing all non-blocking read-/write operations synchronously. The team behind Redis tried implementing this API for Redis because Redis’ performance is significantly impacted by system call overhead.[3] Salvatore Sanfilippo, the former lead developer, claims to have obtained bad performance by replacing “epoll” with “io_submit” [65]. Unfortunately, a Redis branch using “io_submit” was not publicly available for testing and to the best of my knowledge, Sanfilippo did not

[2] set with the “EPOLLET” mask
[3] This is especially true since the Meltdown/Spectre patches.

release any concrete numbers for verification.

2.1.4 io_uring and the Proactor Pattern

"io_uring" is on its way to become the new standard for asynchronous I/O operations in Linux. It was first introduced in upstream Linux kernel version 5.1 and thus is a relatively young API. Instead of improving the support for established Linux AIO, it introduces a completely new interface. Its goals are ease of use, extensibility, feature richness, efficiency and scalability. "io_uring" wants to achieve those goals by providing a generic, yet powerful, set of data structures and communication channels with the kernel to the user space application. When utilizing "io_uring" the user-space application places I/O requests (submission events) in the "SubmissionQueue" (SQ) and the Linux kernel processes the events and places the "result" in the form of completion events in the "CompletionQueue" (CQ). Then, they are ready to be retrieved by the user-space application. So in essence, "io_uring" is a relatively lightweight abstraction that implements two producer-consumer queues to establish a bidirectional communication channel between user and kernel space for I/O related operations. Unlike previous Linux AIO which only worked with unbuffered regular file I/O this interface works truly asynchronously with all kinds of I/O, most notably with regular buffered file I/O and networking I/O.

The "epoll" instance, which holds the interest and ready list and is essential to the popular "epoll" interface, is only mapped into kernel space. One consequence of this design choice is that the data structure can only be accessed via system calls, which can be comparable expensive in I/O heavy applications, especially since the Meltdown and Spectre patches. The "SubmissionQueue" and "CompletionQueue" of "io_uring" on the other hand are mapped into user and kernel space. Both are implemented as single-producer, single-consumer ring buffers and work completely lock-free. The section about concurrent queues goes into more detail about how such queues can be implemented. If kernel side polling for the "SubmissionQueue" is enabled, processing I/O without performing a single system call is possible as long as the application drives the I/O [14].

The architecture of "io_uring" makes the Proactor pattern a suitable fit for implementing highly concurrent network-bound applications. While the discussed Reactor pattern is designed on the basis of a Synchronous Event Demultiplexer such as "epoll", which can be queried for notifications on **operation readiness**, the Proactor Pattern is designed on the basis of an Asynchronous Operation Processor, which accepts operations, **asynchronously**

executes them and later notifies the completion dispatcher of their completion. In the Proactor pattern the completion triggers the dispatch of an associated Completion Handler [59]. This is similar to how an event triggers the dispatch of the Event Handler in the Reactor pattern. However, this Completion Handler, unlike the Event Handler, does not execute synchronous I/O operations, but instead queues more asynchronous I/O operations. In this pattern "io_uring" acts as the Asynchronous Operation Processor. With previous I/O selectors, such as "epoll", an Asynchronous Operation Processor had to be emulated with software in user space, which led to the Reactor pattern being the preferred choice in highly concurrent network-bound applications until now.

Benchmark tests for "io_uring" are promising and show significant performance gains over the "epoll" API for networking [80]. Like kqueue in 2000, "io_uring" might revolutionize I/O on UNIX-like operating systems, for the first time offering a "truly asynchronous" generic I/O interface, which actually works with all kinds of I/O and which can in theory remove the need to perform system calls for I/O operations completely. However, it might take some time for major applications to adopt it because the API differs so drastically from the established I/O selectors provided by current UNIX-like operating systems that it is not just a drop-in replacement.

2.1.5 Avoiding copies with MSG_ZEROCOPY and sendfile

MSG_ZEROCOPY

As the name suggests MSG_ZEROCOPY makes zero copy writes to a socket possible and enables truly asynchronous socket I/O on Linux. The kernel notifies the process when it is safe to reuse the previously passed buffer by placing completion notifications on the socket error queue. The queue has to be polled by the application. It is a trade off between per byte copy cost and the page accounting plus the completion notification overhead, which are required for the feature to work as intended. MSG_ZEROCOPY is supposed to pay off with writes over around 10 KB [18].

sendfile

The sendfile system call on Linux enables data transfers between two file descriptors within the kernel space, thus it skips the usual unnecessary copy of the buffer to the user space. Sendfile pays off in particular when large static "regular" files are transferred over a socket, such as it is often the case in web server applications. This approach is also known as

"zero-copy transfer" [46, pp. 1260–1262][22].

2.1.6 Linux Network Stack and User Space Network Drivers

In order to be able to discuss user space network drivers' improved performance, it is important to explain how data gets from the socket APIs provided by Linux to a network interface controller (NIC).

The Linux kernel stores a "socket" structure for each connection which is identified by a file descriptor. The "socket" has a "sock" structure which describes the corresponding INET socket. This "sock" structure owns a linked list of "sk_buff" (socket buffer) structures. The socket API pushes the socket buffers to the underlying protocol implementations which add lower layer data, such as port number and network addresses.

In the network subsystem the matching "net_device" (network device) structure is selected based on IP routing rules. This structure is at the core of the network driver layer and has the methods for acting upon the networking device. For example, for transmitting data the "sk_buff" is passed to the "hard_start_xmit" function, which is implemented by the device driver [30, 19].

From here on the implementations depend on the networking device, i.e. a specific NIC. So, as an example, this section describes how the driver for the Intel 8256x NIC device family works. However, this information may apply to a wider variety of NICs.

The NIC is connected via the Peripheral Component Interconnect Express (PCIe). The driver running on the host can communicate with the NIC via configuration registers which are mapped to physical RAM (MMIO). The PCIe Core on the NIC is responsible for bus communication, memory mappings and sending interrupts to the host's CPU. It has a direct memory access (DMA) Engine, which handles the receive and transmit data transfers as well as descriptor transfers between the host memory and a small amount of on-chip RAM for temporary buffering, which is filled and consumed by the Ethernet Transceiver (PHY). The driver initializes the configuration registers, so that the DMA Engine "knows" where the RX and TX descriptor queues are located in the host's physical memory. These descriptor queues are implemented as ring buffers and contain the descriptors with each one "pointing" to a specific packet buffer. The NIC driver consumes RX descriptors and the associated RX packet buffers and produces TX descriptors and associated TX packet buffers based on the input of the kernel. The roles are reversed for the DMA Engine [44].

Moreover, the NIC driver can either work in interrupt or poll mode. The NIC notifies the

driver of events by sending interrupts over the PCIe core. In interrupt mode the CPU acts upon these interrupts, e.g. it reads a RX descriptor off the RX descriptor queue, when a new packet arrives. Receiving and acting upon the interrupts is not for free though and creates overhead. When the NIC is busy and new packets arrive continuously, it can be more efficient to mask the interrupts and repeatedly poll the RX descriptor queue instead.

User space network drivers

User space network drivers are bypassing the kernel on Linux by utilizing the Linux UIO API. UIO maps the MMIO and interrupts to pseudo files, which the user space process is granted access to. To work around the problem of the virtual memory in user space, which does not guarantee that underlying physical addresses stay the same, applications make use of "hugepages" because the Linux kernel guarantees that their physical location does not change during runtime to enable safe communication with the NIC [69]. So the RX and TX descriptor queues and their associated packet buffers are entirely mapped into user space with "hugepages". By using these techniques all the kernel abstractions and features are bypassed effectively and the user space application gets direct access to the NIC.

The Data Plane Development Kit (DPDK) is an open-source project that enables such user space network stack and is currently managed by the Linux Foundation.

There are certain scenarios, where this approach has clear advantages over a kernel network stack:

Context switches to kernel space come at a cost, even more so since the patches against Meltdown/Spectre.[4]

There is also a considerable amount of "overhead" in the generic socket and networking APIs within the Linux kernel until the data is passed to the driver. The smaller the transmitted payloads are, the more substantial the associated overhead within the kernel is and at a certain point a hard limit is reached within the kernel, which, depending on the payload size, might be below what the NIC is actually capable of. In this scenario bypassing the kernel altogether in favor of a user space network driver and TCP/IP stack becomes attractive to make the most of the NIC's capabilities. Working with user space network drivers below this threshold is not recommended as they also have several disadvantages:

For one, the application's security deteriorates. Developers should program with extra care due to the process' ability to directly access the NIC.

[4]More on that in the section about the impact of side-channel attacks.

Apart from the operating system and the application, the user space driver also has to be updated during operation.
Depending on the NIC, the application might also have a monopoly on it. This is not a viable solution if multiple applications in the system are in need of concurrent access.
The application gets direct access to the ethernet frames, meaning that if the application wants to bypass the kernel completely, it needs to implement its own additional (TCP|UDP)/IP stack in user space and perhaps a socket API for convenience, too. OpenFastPath is an example for an open source TCP/IP stack which can run together with DPDK in user space [58]. It also provides convenience wrappers for the "BSD socket API" and popular I/O notification selectors such as "epoll". However, using those wrappers might result in worse performance.

2.2 Concurrency & Parallelism in Data Plane Processing

Data plane processing refers to the computation that is required for transmitting packets/frames from one interface to another in a computer network. User space drivers move data plane processing from kernel to user space. The subsection on the Linux network stacks already described the basic principle of how a NIC driver can communicate with a NIC. It also mentioned some of the advantages and disadvantages of using network drivers in kernel and user space. However, it did not go into detail about strategies to efficiently process received and transmitted packets.

2.2.1 Pipeline Model

In the pipeline model the package processing is divided into stages, with each stage requiring roughly the same computing power. Each stage is assigned to a thread. However, this model has a few disadvantages. For one, it is nearly impossible to perfectly divide the processing into stages that require the exact same amount of computing power. Depending on how well the process is divided into stages this might even significantly waste resources. On top of that, communicating between the pipelines might require memory accesses which could bottleneck the system. Finally, each packet is not core-local and due to the fact that state of the art CPU cores have "large" non-shared lower level caches nowadays, this might even further worsen the performance [36].

2.2.2 Run-to-completion Model

In the run-to-completion model the package processing is not divided into stages. Instead packets are assigned to each core/thread and each core/thread aims to handle the complete processing of those packets from beginning to end. The run-to-completion model addresses problems of the pipeline model. In theory, it wastes less resources and requires no communication between pipeline stages because there are none. This results in less memory accesses and the packets are kept core-local.
When dealing with dedicated hardware accelerators, such as hardware for checksum offloading or field-programmable gate arrays (FPGAs), or the CPU needs to wait on some other resource, the thread can make use of a cooperative multitasking approach. With this approach a task returns to a scheduler whenever it has to wait on a resource [36]. One problem of the run-to-completion model is possible contention on the NIC resources by the multiple threads. However, modern NICs have multiple descriptor queues which can be mapped in accordance to the number of threads, reducing contention.

2.3 Impact of side-channel Attacks

Research indicates that the Meltdown and Spectre security patches result in a significant slowdown of system calls. Above all, these patches enabled kernel page-table isolation (KTPI) and measurements to avoid indirect branch speculations. KTPI forces processes to maintain separate page tables for kernel and user space to fix Meltdown. Subsequently, this leads to translation lookaside buffer (TLB) misses, which results in a performance decrease by up to 63% on socket reads [4]. Spectre was apparently[5] patched using a software construct called "retpoline" to prevent branch-target-injection [78], which adds around 30 CPU cycles to each indirect jump or call and which slows down "epoll" by 72% [4]. Other security mechanisms which were patched in the Linux kernel such as "SLAB free list randomization" and "hardened user copy", whose aim is to reduce the effect of buffer overflow attacks within the kernel, as well as new features and kernel misconfigurations can also result in performance decreases with system calls. Overall, Ren's et al. paper "An Analysis of Performance Evolution of Linux's Core Operations" (2019) suggests that Redis', Apache's and NGINX's performance can be sped up by 56%, 33% and 34% respectively by mitigating and disabling various performance decreasing patches [4].

[5] "Apparently" because this seems to be architecture-dependent.

The gist of the paper is that users should check if they really need all those patches and if the Linux kernel is properly configured to their needs. Disabling patches and reconfiguring the kernel can lead to significant performance increases.
In conclusion, avoiding system calls in network bound applications whenever possible seems to be the right approach.

2.4 Lock-free Concurrent Queues and Memory Barriers

The data structure that drives efficient multiprocessor workloads is the ring buffer. In computer I/O the ring buffer data structures typically serve as “First In, First Out” (FIFO) queues in the common producer-consumer pattern. The ring buffer stores data in a single fixed-size array and treats it as if the ends were logically connected. The structure also stores a tail and a head “index” for this array. The tail is used by the producer and the head is used by the consumer for indexing the array. When the producer pushes a value onto the queue, it updates the tail and when the consumer pops a value off the queue, it updates the head. When the queue is full, old values can either be overridden or the process waits until a value is popped off the queue. A third possibility is that enqueueing simply returns with an error. When the queue is empty, the process can either wait until a value is pushed onto the queue or return with an error. These queues can be useful for many purposes. For example, the RX/TX descriptor queues that were already mentioned are implemented as ring buffers. The “SubmissionQueue” and the “CompletionQueue” of “io_uring” are implemented as ring buffers as well. And message queues for message passing between concurrent processes can be implemented as FIFO ring buffers, too.
There are different implementations with different thread-safety guarantees, which can be selected depending on the needs of the application. Single-producer, single-consumer (SPSC) Queues for the communication between two threads (1:1) are very common. One implementation of such a queue is the "WaitFreeQueue<T>" described by Herlihy [40, pp. 45–48]. This implementation is very simple, because it requires no locking at all and the possible concurrent access creates little overhead. The pushing onto the queue does not have to be synchronized because only one thread is allowed to produce values; neither does the popping of values off the queue because only one thread is allowed to consume values. But the pushing and popping both require reading head and tail values. Therefore, the load and the store operations performed on the head and tail have to be atomic.

When a program is running on one CPU core, on modern systems each load and store operation is guaranteed to be visible in the correct order (obviously), but depending on the memory model implemented by the CPU possibly **only** on that core. Modern processors usually do not execute the instructions of a program in strict sequential order for performance reasons and each CPU core typically has its own lower level cache(s). For these reasons, depending on the memory model implemented by the processor, the data produced by a thread can be visible in an arbitrary order to another concurrent thread by default. Compilers for "lower level" languages like C, C++ or Rust are also allowed to reorder the code for performance optimizations.

This could become a problem in Herlihy's "WaitFreeQueue<T>": The thread consuming the entries could see an incremented tail index before the data written to the array becomes visible, which might result in undefined behavior.

Herlihy's "WaitFreeQueue<T>" is implemented in Java. With the memory model implemented by the JVM (Java Virtual Machine), Herlihy's "WaitFreeQueue<T>" works under the condition that the head and tail are atomic or volatile variables [40, p. 63]. However, "lower level" languages, like the ones mentioned above, cannot rely on the memory model of a higher level virtual machine.

"Lower level" languages, like C, C++ or Rust, which are typically compiled to native machine code, are agnostic about the memory model of the instruction set architecture (ISA) that the code is compiled for. In order to be able to deal with various memory models on different CPU architectures, their "abstract machines" introduce so-called memory barriers. Loads and stores to atomic variables are typically supported by memory barriers. For other concurrent threads they provide different visibility of load and store operations and ordering guarantees before and after the access to the atomic variable. Compilers map these abstract memory barriers to the concrete memory model of the target architecture and restrict potential reorderings, which they would otherwise be allowed to perform on the instructions.

Depending on the CPU hardware, this mapping typically makes use of special load and store instructions. On Intel/x86-64 all load and store instructions by default offer the relatively strong "acquire-release" semantic, which is sufficient for the correctness of many algorithms. This is discussed in detail when the "lower level" implementation of Herlihy's "WaitFreeQueue<T>" is addressed. "Acquire-release" does not give any guarantees regarding global ordering of visibility in-between "independent" atomic variables in different concurrent threads. To achieve this global ordering, sequential consistency is required [63]. On

Intel x86-64 sequential consistency can be achieved using the “MFENCE” memory-fencing instruction, the “LOCK” prefix or special instructions with implicit “LOCK” prefix, like the exchange (XCHG) instruction [42]. Compare-and-swap (CAS) operations with “CMPXCHG” require such a “LOCK” prefix and are thus sequentially consistent by default. There is also no difference between weak and strong CAS on x86-64. According to the C++ specification, weak CAS is allowed to fail spuriously. This means it could fail although the result of the comparison is true. CAS is explained in more detail in a few paragraphs.

“ARMv8”, on the other hand, implements a weak memory model. Standard load and store operations with “LDR” and “STR” provide relaxed memory order. Sequential consistency, and thus “acquire-release” semantics, too, can be achieved by using the load-acquire register (LDAR) and the store-release register (STLR) instruction. Since ARM requires explicit load and store operations, there are no dedicated atomic read–modify-write operations. ARM has special “exclusive” load and store instructions for implementing such atomic operations. For exclusively loading “ARMv8” provides the load exclusive register (LDXR) and load-acquire exclusive register (LDAXR) instruction and for storing there is the store exclusive register (STXR) and store-release exclusive register (STLXR) instruction [6]. Interestingly, store operations on “ARMv8” can fail in case of contention on the cache line related to the exclusively loaded address and the status result is written to a third register. In case such a failure is not acceptable, the CPU has to loop until the read-modify-write operation is executed successfully. This is processor spinning exposed to the developers and sort of optimistic concurrency control within the CPU.[6]

Modern CPUs try to do as little stalling and make as few memory accesses as possible. They go to great lengths to only lock the bus when it is necessary. So they try to enable the memory guarantees requested by the application within the CPU caches by using complex cache-coherence protocols [40, pp. 473–476]. Still, these instructions and sharing memory between CPU cores due to coherence traffic in general often come at a significant cost. It is important to use the lowest barriers that enable the algorithm to work correctly.

To make Herlihy’s "WaitFreeQueue<T>" work on “lower level” languages, memory barriers are necessary. This rather simple SPSC queue only requires two barriers to function correctly. The store operation of the tail value has to be implemented with a release barrier after a value was pushed onto the queue. This guarantees that no reading or writing operations in the current thread can be reordered **after** this store. The load operation of the tail value

[6] As a result, there *is* a difference between weak and strong CAS operations on ARM and this is the reason why it is recommended to use weak CAS inside and strong CAS outside of loops.

in the consumer thread has to be implemented with an acquire barrier. The acquire barrier guarantees that no reading or writing operation in the current thread can be reordered **before** the load. The combination of both barriers is also called “acquire-release” protocol. It guarantees that before the atomic store, all write operations in the one thread are visible to other threads that acquire the same atomic variable. So these barriers provide sufficient guarantees for Herlihy’s "WaitFreeQueue<T>" to work correctly. All the other atomic load and store operations can be done relaxed, which means that ordering guarantees are not required before and after the access of the atomic variable. The code for a lock-free SPSC ring-buffer FIFO queue, which I have implemented in modern C++ and which is very similar to Herlihy’s "WaitFreeQueue<T>" can be found in the appendix. The implementation of “io_uring” described in an earlier section makes use of a similar queue to implement the “SubmissionQueue” and the “CompletionQueue”, which are used for communicating I/O requests and completions between kernel and user space.

Multiple-Producer, Multiple-Consumer Queue

However, a SPSC queue is not always sufficient. Algorithms with higher concurrency often require M:N communication channels and thus a multiple-producer, multiple-consumer (MPMC) queue or something in between like a multiple-producer, single-consumer (MPSC) queue. The trivial solution is to utilize the given SPSC queue and put a lock around the enqueuing, the dequeuing or both. This can be the best solution depending on the queue’s workload [50].

However, it is also possible to build entirely lock-free MPMC queues. All these implementations rely on compare-and-swap (CAS) operations:

Compare-and-swap operations are possible with most modern CPUs. They enable comparison between data stored in a memory location and another value, which was often read from that memory location previously. If both values are the same, the CAS operation stores a third new value. All of this is done in a single atomic operation with the caveat that the data CAS can operate on is limited in size. That is to say, on modern x86-64 CPUs the limit for atomic read-modify-write operations is set to 16 Bytes. Considering that the word size and thus the pointer size on x86-64 CPUs is already 64 bit (8 Bytes), this is fairly limited. It also implies that the array in a generic circular lock-free MPMC queue is limited to storing pointers so that multiple parallel accesses on the consumer and the producer side are possible. Consequently, the array has to be atomic.

The following paragraph explains the basic idea behind one implementation of such a MPMC lock-free queue. The queue is heavily inspired by the talk "An Interesting Lock-free Queue" that Tony Van Eerd gave at the CppCon in 2017 [79].

The main array of the queue stores a pointer together with an integer representing the generation of the pointer. In my implementation the generation is represented by an unsigned 64 bit integer, making the total memory footprint of the structure 16 Bytes. This means it is just within the boundaries of CAS operations on x86-64. The pointer in this structure can be null or it can store a valid memory address which points to another structure. Using CAS operations, multiple concurrent threads can exchange this generational pointer within the array. When one producer wants to push another entry into the queue, it simply iterates to the first generational pointer within the current generation that is null. Then, the thread initializes the new generational pointer and tries to store it in the specific location using a CAS operation, which succeeds if the previously stored old generational pointer did not change. If the operation fails, the thread repeats the whole process. The dequeuing works analogously, except that the thread iterates to the first generational pointer that stores a valid address and tries to replace it with a new incremented generational pointer, which points to null.

The generation is logically incremented with each new cycle through the circular array, so that threads always "knows" when a pointer has been updated, even if this pointer still points to the same address. So pairing of the generation with the pointer fixes the "ABA Problem" [40, pp. 223–238].

To speed up the iteration, which would otherwise be a costly search with a complexity of *O(n)*, the queue also stores atomic head and tail values for indexing similar to the SPSC queue. However, these values do not represent the exact position of the current head and tail pointer. They are rather an approximation. The true head and tail values are guaranteed to be more recent (larger) or equal to the stored head and tail values, resulting in eventual consistency. To determine this "happened-before-relation", the head and tail indexes need a generational stamp as well. With each queue/dequeue operation the executing thread tries to exchange the current tail/head for its own updated version of tail/head. It is important that all previous stores done by the thread become visible to all other producer and consumer threads before updating the tail/head values with a CAS operation. This optimization makes the best and common case queue/dequeue an *O(1)* operation, just like the SPSC queue, and in the worst and unlikely case an *O(n)* operation.

Disadvantages of this queue compared to the SPSC queue are that it only stores generic

pointers to structures, while the SPSC can store structures of arbitrary size directly within the array, resulting in a memory indirection and worse cache locality for the MPMC queue. The pointers to the allocated structures in the array also require further memory management in order not to be invalidated, while at least one thread is still accessing the memory the pointer points to. This could be done by a reference count, for example. It is important not to make a mistake, otherwise this could lead to undefined behavior. This queue requires multiple CAS operations, while the SPSC queue requires no CAS operations to work correctly. CAS operations can be comparatively expensive depending on the underlying CPU architecture. Overall, this MPMC queue has more overhead compared to the SPSC queue.
My implementation of this queue in modern C++ can be found in the appendix of this thesis.
Some of the key benefits that ring buffer data structures provide for I/O heavy workloads including highly concurrent network-bound applications are the following: They allow for *O(1)* access and can be implemented lock-free as demonstrated above. Furthermore, they can serve as FIFO queues for the common producer-consumer pattern. And what is important for modern CPUs: They are cache friendly because the data is stored in one contiguous chunk of memory. This is in accordance with what Bjarne Stroustrup says about data structures: Do not store data unnecessarily; keep data compact and access memory in a predictable manner [71].

2.5 Fast User Space Locking with futex

The previous section already implied that concurrent lock-free data structures might not always be the best fit for every problem. These data structures can become very complicated and the associated overhead that makes the data structure lock-free in the first place can become more expensive than using locks with non- or partially-concurrent data structures. When there is low contention on the lock, the inefficient part about "conventional" non-spinning mutex locks is the system call that is required for acquiring and releasing the lock. One solution to work around the system call is to use Spinlocks, but Spinlocks are considered to be a bad idea in user space for the most part [77]. Chapter 3 goes into more detail about this, when discussing busy-waiting in Redis' I/O threading implementation.
Fortunately, Linux offers a compromise: The futex system call makes it possible to develop synchronization primitives around an atomic variable in user space, so that not every operation upon the synchronization primitive requires a system call.

Mutexes can be implemented by utilizing the futex system call. When there is no contention, a mutex implementation with futex only requires a "cheap" atomic CAS operation in user space to acquire and release the lock, similar to a CAS Spinlock in user space. When there is contention and acquiring the lock at first try[7] fails, the implementation falls back to the futex wait system call[8], which takes the memory address of the atomic variable and the expected value as an argument and executes a "compare-and-block" operation in kernel space. This operation is required to prevent a thread from being blocked indefinitely. This would otherwise be possible because the lock could already be released concurrently in the "time frame" from the CAS operation to the system call. The "compare-and-block" operation makes this safe and the futex system call immediately returns to user space in this case. On the other hand, if and only if the atomic compare operation in kernel space succeeds, the thread is blocked [16].

This Mutex can be unlocked by atomically storing a specific value in the (atomic) variable, which is a comparatively efficient operation, as long as there are no (blocked) waiters. If there are (blocked) waiters, the futex wake system call has to be invoked, which wakes up a specified amount of blocked waiters.

This is the basic principle behind implementing Mutexes utilizing the futex system call from a high level perspective. Obviously, there is more nuance to real implementations, so I implemented the "mutex2" and "mutex3" algorithm that Ulrich Drepper introduces in the paper "Futexes Are Tricky" [35] in modern C++ with C++ memory barriers and included the source code in the appendix.

Nowadays, the implementations behind popular "general purpose" Mutex interfaces, such as the POSIX "pthread_mutex" or the "std::mutex" from the C++ standard template library, usually make use of the futex system call on Linux.

[7] This could also be implemented as a Spinlock-hybrid with multiple tries.

[8] Actually there is one multiplexed futex system call and the operation is just a parameter.

Chapter 3

Practical Architectures & Paradigms for Concurrency & Parallelism in Highly Concurrent Network-Bound Applications

This chapter examines architectures and paradigms for concurrency & parallelism in highly concurrent network-bound applications in great detail.

When developing highly concurrent network-bound applications, the major debate is about the shared-nothing versus the shared-something architectural paradigms. Decisions made in regards to the applied paradigm usually have the highest impact on the scalability and key characteristics of an application. So the first section discusses the essence of the shared-nothing and the shared-something approach as well as their implications on highly concurrent network-bound applications. One subsection introduces the shared-nothing "thread-per-core" model that has emerged in the past years to deal with modern high core count CPUs. Its aim is to deliver efficient multi-core scaling on these CPUs in particular.

Furthermore, the chapter addresses work distribution and scheduling strategies in the implementations of established and upcoming highly concurrent network applications and library abstractions, such as Redis version 6 with I/O threading, KeyDB, NGINX, libuv, Tokio and Glommio. The chapter tries to classify the techniques that these implementations make use of based on queueing models and work distribution algorithms. The consecutive section is devoted to Rust Futures and zero-cost async-await and picks up where the preceding sub-

sections about Tokio and Glommio left off. Futures and async-await enable so-called “green threading” - a paradigm that is common in “higher level” concurrent programming. Rust futures make this paradigm viable for highly concurrent “low level” implementations and are essential for the Rust Libraries Tokio and Glommio.
While the sections, which were already described, are mainly about incorporating thread-level parallelism into highly concurrent network-bound applications, the last section in this chapter “changes perspective” and discusses how such applications can benefit from SIMD.

3.1 Shared-Nothing vs. Shared-Something

3.1.1 Essence

Whether to use the shared-nothing or the shared-everything architecture or something in-between, deemed “shared-something” in this thesis, is a fundamental question when it comes to developing any highly concurrent server application.
Historically, shared-nothing architecture meant that a “high transaction rate multiprocessor system” does not share memory and disk I/O resources, while a shared-everything system shares both memory and disk I/O resources. Something in-between could be that processors only share either memory access or disk access [70] (shared-something). However, nowadays this concept is further abstracted to any kind of logical or physical resource that can be shared between nodes [29]. A node can store a finite amount of state and offers a finite amount of computational resources, with both hard limiting the amount of data a node can keep and the throughput it is able to process in practice.

3.1.2 Discussion

Although implementation details differ vastly depending on what flavour of shared-nothing, shared-something or shared-everything is actually meant, some general observations about the disadvantages of the approaches can be made: Shared-nothing architectures aim to have no contention on resources simply by not sharing them at all. Therefore, in theory, the throughput of an application using shared-nothing can increase linearly in parallel execution as discussed later in the section about the thread-per-core architecture. In shared-something architectures some resources are shared by definition, resulting in potential contention on these resources. This might force parts of the application to synchronize, which reduces the part of execution time that can actually benefit from parallel execution. Using Admahl’s law,

one can calculate the theoretical upper performance limit depending on the proportion of execution time that benefits from parallel execution and the amount of available executing nodes [40, pp. 13–14] e.g. CPU cores. Usually, a shared-nothing architecture requires almost no or very little synchronization between nodes, and therefore the theoretical upper performance limit is higher compared to the shared-something architecture. Shared-something architectures usually "pay a penalty" in Amdahl's law for requiring some sort of synchronization.
However, the efficiency of shared-nothing approaches stands and falls with the "delightfulness" of the problem that the implementation is supposed to solve. In this context "delightful" means that the state can be partitioned into non-overlapping collections so that all transactions on that state are locally sufficient [70], i.e. they can be executed on a single node without the involvement of another node. Depending on the nature of the aforementioned state and the application, partitioning state into non-overlapping collections so that transactions are locally sufficient can become a non-trivial problem to solve. One reason for this is that the amount of stored state should be roughly equal among all nodes. A shared-nothing architecture where, for example, 99% of the state is kept on a single node defeats the purpose and the benefits of shared-nothing. The same principle also applies to the computational resources of the node. A shared-nothing architecture where 99% of the computation is performed on a single node would make no sense either. The computational resources should be utilized about equally across all nodes. Shared-nothing databases, such as Redis Cluster, usually use "hash partitioning" [47, p. 205] to fulfill the goal of evenly distributing the keys and the state among the nodes. However, this does not prevent a highly skewed load distribution [1]:
If the load is not well-balanced, the nodes whose utilization exceeds that of the other nodes could become hotspot nodes. This is why the shared-nothing architecture might be more prone to hotspot problems depending on data access patterns. The following paragraph provides an example of this:
Given are a shared-everything database running on a single node and a shared-nothing database running on three equivalent nodes A, B and C. The shared-everything database and the shared-nothing database can achieve a maximum throughput of 250k requests and 300k requests per second respectively. In the shared-nothing database a single node can handle 100k requests per second. The data in the shared-nothing database is partitioned in hash slots. One key, which is located in node A of the shared-nothing database, becomes hot. This key is requested 120k times per second. Consequently, node A is running at its hard limit and is forced to drop requests. Within the same time frame another key becomes hot and is

requested 100k times per second. Coincidentally, this key is located in node A, too. Node A can not keep up with serving 100k additional requests per second, because its hard limit is already reached, so it is forced to drop even more requests. While node A is failing to serve all of its requests, node B and C of the shared-nothing database might just be serving 5k requests per second each and thus are underutilized. Although it has a lower absolute maximum throughput than the shared-nothing database, the shared-everything database could in theory serve all the requests perfectly fine in this example. This is an extreme case and a thought experiment, but skewed workloads are not unusual. For example, when a celebrity on a social media site causes a "storm of activity", millions of followers might want to access the celebrity's profile, which could internally be represented by an ID that is the key to the profile content in the database. In conclusion, one partition has to deal with the whole workload. Skewed workloads have to be tackled at application level for most databases [47, p. 205], e.g. by making the data set more finely grained.

Getting back to the example with the shared-nothing database, it would be the best hypothetical case if the keys and the access to the keys were uniformly distributed among the nodes.

The hotspot problem mentioned above was named as one of the reasons why "Alibaba Cloud" decided to implement a multithreaded fork of Redis despite the possibility of data partitioning via Redis Cluster [9].

This problem is closely related to queueing theory. Shared-something architectures normally make use of the single-queue, multi-server model, whereas shared-nothing architectures usually make frequent use of a multi-queue, multi-server model.

"Non-delightful" transactions on shared-nothing systems are either not possible at all or they require a transaction protocol and message passing between the nodes. Generally, depending on the latencies between involved nodes and the frequency and execution cost of "non-delightful" transactions, the performance ranges from acceptable to too slow for a viable system.[1]

The metastate, which contains the information on which node a specific piece of state is stored, also has to be stored somewhere and has to be distributed.

As shared-nothing architectures usually involve more logical nodes than shared-everything architectures, they can introduce more administrative overhead to distributed systems. In conclusion, if the data and the access patterns meet the requirements for the shared-nothing

[1]Interestingly in shared-something systems the complexity is in the field of concurrency & parallelism, while in shared-nothing systems the complexity shifts to problems related to distributed computing.

architecture, its advantages outweigh the benefits of shared-something approaches and it should be preferred. The benefits of the shared-nothing architecture are not limited to "truly" distributed environments. Applications running on modern servers with multi-core CPUs can also benefit from the shared-nothing architecture, which will be demonstrated in the section "Thread-Per-Core" and in the performance evaluations for this study.

3.1.3 "Work around" non-delightful Transactions - A Case Study Of Transactions in Redis Cluster

The in-memory database Redis enables transactions via Lua scripts. Clients can send Lua scripts, which can then be evaluated on the Redis server with specified parameters. Lua scripts are executed transactionally and provide strict serializable isolation guarantees on one Redis server instance [32]. Redis transactions do not make use of any client interaction within the script, which makes them particularly efficient. Redis Cluster consists of multiple Redis server instances arranged in a shared-nothing database cluster, where keys are evenly distributed among nodes via hash slots. Redis Cluster does not support distributed transactions. That means transactions in Redis Cluster can never cross the boundary of a single node and thus only involve keys which are all stored on the same Redis server instance. However, to make transactions in Redis Cluster possible, the client can force the involved keys into a single hash slot on a single node by utilizing a concept that Redis calls "hash tags" [62]. These "hash tags" ensure that only a substring of the key is hashed for the hash slot by putting it into braces. So, for example, if the keys "foo" and "bar" are involved in a transaction, the substring "baz" can be appended in braces. Only "baz" is hashed for the hash slot and this ensures that "foo{baz}" and "bar{baz}" end up on the same node, making the transaction possible. The downside of this approach is that it may contribute to worsening the hotspot problem that was described in the previous section.

3.1.4 Thread-Per-Core and Amdahl's law

To fully utilize the core count of modern multicore CPUs in I/O heavy applications, the thread-per-core architecture was developed. The goal of this architectural approach is to embrace the fact that CPU core speeds do not increase as much anymore, whereas core counts on CPUs increase monotonically. So its focus is on utilizing all available CPU resources. The principle is simple: Create exactly one thread in the application for each available CPU core. The idea is that assigning exactly one thread per core reduces the amount of expensive

context switches in the system.

The thread-per-core model works especially well with the shared-nothing architecture because in theory it can scale linearly with the CPU core count in the system:

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

$$\text{Given } p = 1$$

$$S_{latency}(s) = s$$

If needed, shared-nothing thread-per-core can make use of "pipe()-style" message passing for inter-thread communication. The time spent on passing messages between processes (or threads) reduces the variable p. Communication between processes should thus be used sparsely.

Shared-something architectures often need synchronization on the shared resources. Depending on whether this synchronization reduces the proportion p of the time, which profits from parallelization, and depending on how much it is reduced, the theoretical speedup of the program is limited. Therefore, the application does not scale indefinitely on an infinite amount of CPU cores. In reality, this means that with each added CPU core the CPU cores are utilized less efficiently and at some core-count this is not a viable scaling strategy anymore.

The paper "The Impact of Thread-Per-Core" compares a shared-nothing thread-per-core in-memory database with the common multithreaded in-memory database "memcached" running on the same hardware and utilizing the same amount of threads. The paper shows that a shared-nothing thread-per-core approach with configured interrupt request (IRQ) affinity can reduce tail latencies by up to 71% for in-memory databases [37]. When IRQ affinity is enabled, only specific cores in the kernel are responsible for packet processing and receiving interrupts from the NIC.

For inter-thread communication the database described in the paper makes use of a bounded and lock-free circular SPSC queue.

3.2 Work Distribution and Scheduling in Highly Concurrent Network-Bound Applications

Highly concurrent network-bound applications are supposed to take advantage of multicore CPUs, so they need methods for distributing the workload to multiple operating system processes/threads and consequently to the CPU cores. This implies the need for concepts and algorithms for distributing and scheduling the workload.

The workload in highly concurrent network-bound applications is primarily characterized by small tasks, which are issued by a large amount of mostly independent connections and require rather "little" computation and long waiting times in-between I/O operations in most cases. A task in such a system is usually a unit of work which can be executed concurrently. The goal is to achieve high performance by multitasking between tasks efficiently. This section discusses how to achieve parallelism in highly concurrency network-bound applications and points out the strengths and weaknesses of different work distribution and scheduling strategies.

The most trivial scheduling strategy for a network application is to use blocking I/O and assign an OS-thread to each established connection as described earlier. This strategy already enables parallelism. With this approach, it is up to the operating system to unblock and schedule the probably mostly blocked threads and thereby serve the concurrent connections. However, the problem with this approach is that concurrency is enabled via thread-level-parallelism. Processes/threads should not be used to enable plain concurrency, which does not necessarily require parallelism, for multiple reasons: Repeatedly creating operating system processes/threads and lots of context switching, which is required whenever data arrives on one socket, are too expensive for highly concurrent network-bound applications. On modern hardware, one context switch requires about 1-2 microseconds [7]. On a 2 GHz CPU core, 2 microseconds equal 1000 CPU cycles. This might not sound like a lot, but in theory, one 2 GHz CPU core has only about 1600 CPU cycles overall to process 1 kilobyte of payload, if the CPU core wants to make full use of the capabilities of a 10 Gbps NIC. 1000 CPU cycles just for a context switch is quite demanding in comparison.

So one thread should handle more than just one connection. The best performance is typically achieved by keeping operating system processes/threads busy with computation that is ensured to make progress and mapping each thread to a CPU core. I/O selectors provided by the operating system and user space network stacks, such as the ones discussed at length in chapter 2, give the application the ability to serve several connections concurrently in an efficient manner, while being agnostic about parallelism within the application in the first place. So the question of how to schedule the tasks driven by the I/O selector and the related processing for several cores efficiently[2] remains.

This section is based on information which can be extracted from the source code and documentations of several open source multithreaded/multi-processed highly concurrent network-

[2]Efficiency is not well-defined in this context. It depends on the requirements for the application.

bound applications and library abstractions, namingly “NGINX”, “Redis” and “KeyDB”, “libuv”, “Tokio” and “Glommio”.

NGINX is the “Swiss Army knife” of high performance HTTP based applications and a very popular web server, caching server, reverse proxy and load balancer [55], which can often be found at the top of corresponding benchmarking results. KeyDB is a multithreaded fork of the popular in-memory database Redis [72]. libuv is a library that provides asynchronous evented I/O, which was originally developed to drive Node.js' event loop [74]. Tokio is a runtime that is focused on asynchronous operations. It serves as a basis for the popular actor framework ACTIX [11] and Deno [33], that is Ryan Dahl's[3] new implementation of a Node.js like JavaScript runtime for server applications. Glommio is a thread-per-core I/O library based on the new Linux asynchronous I/O API “io_uring” [34], which is still in its early stages.

Nowadays, most highly concurrent network-bound applications achieve parallelism by integrating processes/threads into the Reactor Pattern in some way or other. The Reactor Pattern was already mentioned in the second chapter in the first section about I/O notification selectors, but its key concepts will briefly be reviewed as a reminder:

In the Reactor Pattern, a handle is an entity of interest, for example a socket. Notifications of specific events on the descriptor, for example read-readiness, can be registered with an I/O notification selector, such as “epoll”, which is usually provided by the operating system. This I/O selector blocks until an event of interest occurs on at least one of the registered handles (file descriptors). A handle has an associated callback function (Event Handler) for each event that can occur on the entity. An Initiation Dispatcher dispatches the callbacks associated with each handler and then goes back to calling the I/O selector (Synchronous Event Demultiplexer) [67]. This is called an event loop and one iteration within the loop is an event loop tick. This approach is efficient because the thread which runs the event loop either processes events and thus makes progress or it is blocked if there are no events to process. In some documentations, this is also referred to as “event-driven architecture”. The main program logic is completely single-threaded and driven by the event loop in such an architecture. In theory this architecture is rather simple. However, in practice there are usually a few obstacles that one needs to know about for further context:

While the reactor architecture works well for networking operations, file operations on UNIX-like operating systems are technically always “ready” and setting them to non-blocking mode

[3]Ryan Dahl developed Node.js in 2009.

makes no difference as explained earlier. They can potentially always block the thread. So running them on the same thread as the event loop is risky and can potentially reduce the performance severely because the event loop could be prevented from processing events and making progress. The solution is to utilize separate operating system processes/threads for handling file I/O requests. This means that even an application with just a single event loop instance running on one thread requires operating system threads for file I/O. NGINX, KeyDB, libuv and Tokio all make use of this approach for dealing with file I/O.
Often it is desired that UNIX signals can be received via the I/O notifications. While this is possible via the "signalfd" system call [23] on Linux nowadays, this behavior can also be achieved by using the "self-pipe trick" [46, pp. 1370–1372].
Waking the event loop thread using another process/thread or communicating custom events can be achieved by utilizing the UNIX "pipe" system call. To do so, a pipe has to be created. The read end has to be registered with the I/O selector and the write end can be used in another process/thread for sending arbitrary data.

3.2.1 Producer-Consumer - Redis I/O threading

Redis' approach to multithreading differs vastly from the approaches of the other mentioned applications and libraries. Redis used to be driven by a completely single-threaded reactor until Redis version 6. By default Redis is still single-threaded, but since version 6 it is possible to manually enable I/O threading. The idea is that the event loop still runs on only one thread, but instead of handling all synchronous reads and writes to the clients' sockets on the thread that runs the event loop, reads and writes are queued and postponed to be processed by multiple I/O threads. The schematic in figure 3-1 shows a visual representation of Redis' I/O threading.
In simplified terms, the main thread, which runs the event loop, is the only producer in the system and the I/O threads implement multiple consumers for fan-out socket I/O operations. The main thread waits synchronously for the consumer threads to finish. The reasoning behind this design choice is that under full load Redis spends the majority of the computation time on I/O operations specifically reading from and writing to sockets [66]. By benchmarking GET and SET operations, I could verify that with payloads up to the smaller kilobyte range Redis spends about 80% of computation time on reading from and writing to TCP sockets under full load. Redis I/O threading is supposed to benefit from parallelizing this part by utilizing a producer-consumer model, which distributes the read and write tasks across

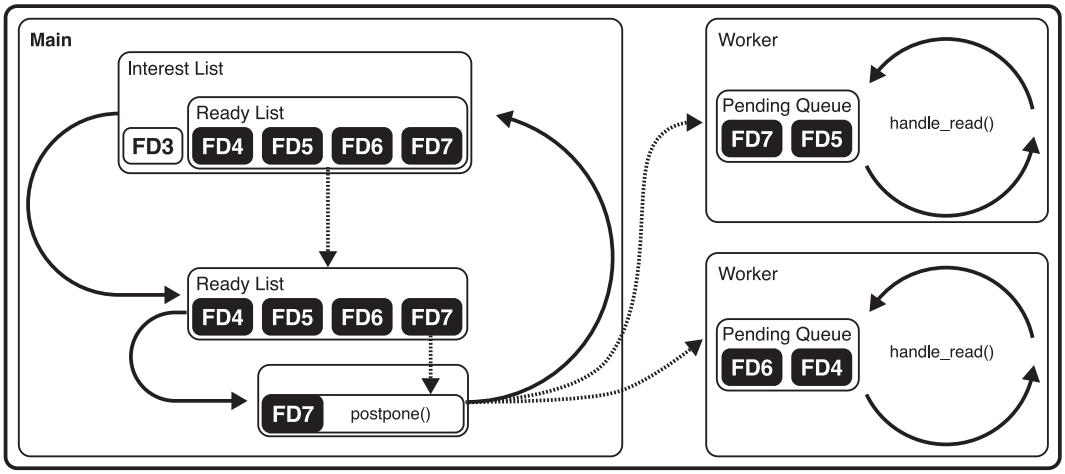


Figure 3-1: Simplified schematic of producer-consumer I/O threading in Redis.

consumer threads. The number of I/O consumer threads is specified in the configuration file or with an option flag at startup.

However, there is a caveat regarding the implementation: The I/O consumer threads are not always active. Instead, they are "woken up" by unlocking a corresponding mutex for each consumer on the main thread when a certain load threshold is reached. If the I/O threads are active and the workload is below the threshold, the consumer threads are blocked by locking the corresponding mutexes on the main thread. When they are awake, the threads are busy-looping and waiting for work by checking if the length of their work queue is non-zero. The main thread distributes clients with pending read or write tasks evenly among these separate queues on each event loop tick. When the work is distributed, the main thread sets the length of each of the consumer queues atomically. This signals the consumers to stop spinning and start processing I/O. After that, the main thread processes its own slice of clients. When the main thread is finished, it synchronously waits until all consumers have finished their work. To find out if all consumers have finished their work, the main thread "busy-checks" if the total sum of the queues' length is zero. The pseudocode for Redis' I/O threading is displayed in figure 3-2. The threads are busy-waiting because waking them up by unlocking the mutex on each event loop tick would be too expensive. Spinning threads do not have to be woken up but they have the disadvantage of "burning" CPU cycles, especially if there is no or little work to do. So the I/O threading is only activated when there is enough work to process. It is advisable to never set the number of I/O threads higher than the number of available CPU cores. The operating system does not "know" if a thread is just busy-looping or actually making progress. It could interrupt the thread running the event loop to schedule

an equal time slice to a thread that is just spinning and waiting for work from the main event loop thread, which could potentially degrade performance. This could also happen if there are more available CPU cores than I/O threads. However, this is unlikely if there is no other significant load on the system. To rule this problem out completely, it is also possible to pin the threads to specific cores. For the reasons mentioned above, Linus Torvalds advises against the usage of spinlocks in user space altogether [77].

```
while !shutdown:
    handle_pending_reads_using_threads(global_read_queue)
    handle_pending_writes_using_threads(global_write_queue)
    # using stateful event notification interface e.g. epoll
    events = stateful_poll_wait(stateful_poll_fd, timeout)
    for event in events:
        if event.fd == listener_fd:
            event.fd.accept()
        if event.mask & READABLE:
            # will postpone read if I/O threads active
            connections[event.fd].handle_read_event()
        if event.mask & WRITABLE:
            # interest is registered (rarely) if write was previously blocking
            # due to level-triggered notifications
            connections[event.fd].handle_write_event()

# producer side (main thread)
handle_pending_writes_using_threads(global_write_queue):
    i = 0
    # distribute work
    for write_task in global_write_queue:
        thread_queues[i++ % thread_count].push_back(write_task)
    # "fan-out" - give the go to the worker I/O threads
    for thread_id = 0; thread < thread_count; thread_id++:
        thread_pending[thread_id].atomic_store(thread_queues[thread_id].length)
    # meanwhile main thread (id = 0) handles its slice
    while write_task = thread_queues[0].pop():
        handle_pending_write(write_task)
        thread_pending[0]--
    # "join" - busy-waiting for the other threads to finish their work
    while sum(thread_pending) != 0
```

```
# consumer side (spinning worker threads)
thread_create(
    # note: in the real implementation this thread can be blocked
    # by a mutex that is controlled by the main thread
    while True:
        # busy-waiting for write task from the queue
        while thread_pending[thread_id].atomic_load() == 0
        # thread handles its slice
        while write_task = thread_queues[thread_id].pop():
            handle_pending_write(write_task)
            thread_pending[thread_id]--
, thread_count - 1)
```

Figure 3-2: Simplified pseudocode for producer-consumer I/O threading in Redis.

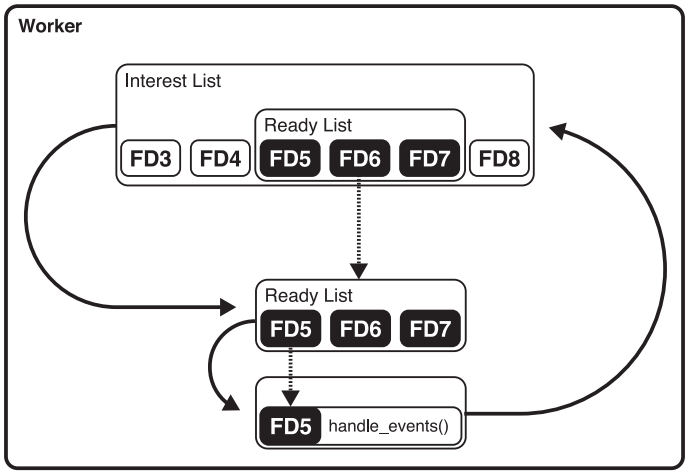


Figure 3-3: Simplified schematic of one worker in KeyDB's multithreaded event loop(s).

3.2.2 Multithreaded Event Loop(s) - NGINX, KeyDB, libuv and co.

The general concept for scaling to multiple cores in NGINX, KeyDB, libuv and Tokio is running the reactor logic with the event loop on multiple operating system threads/processes. There are, however, subtle differences in the technical implementations regarding the usage of "epoll" and other I/O notification systems. The scheduling of the work also differs and for some of the implementations this is not the only paradigm for achieving parallel processing besides file I/O.

To put it simply, the NIC can be seen as the producer and the threads running the event loop as the consumers. The interface between them is a (stateful) I/O selector provided by the operating system. In the case of "epoll", the "epoll" instance created with "epoll_create" can be seen as the work queue. By calling "epoll_wait" the threads running the event loop pop multiple tasks off of that work queue. The first distinction between the implementations is that this "main" work queue, which distributes the work on each event loop tick, can either be shared or non-shared. KeyDB and NGINX have a work queue and therefore an "epoll" instance per worker thread/process on Linux, while Tokio shares the "epoll" instance.[4] The schematic in figure 3-3 shows a visual representation of a single worker thread in KeyDB's multithreaded event loop. The corresponding pseudocode is displayed in figure 3-4.

Conceptually, both approaches have advantages and disadvantages. The reason why both approaches are in use boils down to queueing theory: A single queue, which is shared across

[4]Redis, which forms the basis for KeyDB fork, does not use edge-triggered but level-triggered notifications and so does KeyDB. Level-triggered notifications are complicated to use with one shared "epoll" instance. As mentioned previously, edge-triggered notifications are preferred in this use case. This might also have influenced the decision to implement one "epoll" instance per worker thread in KeyDB.

workers, leads to a better distribution of the work and consequently to potentially lower latencies and higher throughput. In theory, the single-queue, multi-server is deemed superior over the multi-queue, multi-server system [3]. In a single-queue, multiple-server scenario, a comparably demanding task does not actively prevent successive work in the queue from being processed. The “blocking” effect is also known as “head-of-line blocking” [47, pp. 17–18]. In the case of “epoll” with edge-triggered notifications, things are not that simple though. Calling “epoll_wait” pops multiple events with associated tasks off of the “work queue”. A reactor style application then processes a whole batch of tasks associated with the events synchronously. So one demanding task within that batch can still “head-of-line-block” successive tasks from being processed, but subsequent tasks, which are added after popping the tasks off of the ready queue[5], are not directly affected by the processing time required for a demanding task in the batch. In most server applications not all workloads, which are generated by those batches, are equal, but in some use cases a high load in comparison to the "batch size" may make this problem insignificant.

Leveraging a shared work queue across multiple threads still tackles the problem of evenly distributing work during runtime to some degree and is therefore already a form of dynamic work assignment. Threads which have completed their work call the stateful I/O selector to pick up a new batch of work from the shared queue, which supports threads that are still busy and processing events.

On the other hand, the principle “one queue per worker thread” eliminates the contention on the queues and generally renders the need for sharing any state of connections with other threads obsolete because each worker only receives the notifications for its own subset of connections. As each worker serves its own subset of connections, there is the need for a concept for assigning these connections to the threads. The assignment of the connections is static in the case of KeyDB, so the algorithm is rather simple: Saturate the threshold of C connections on each of the N worker threads. When the threshold is saturated on all workers, each connection is “round-robin” distributed, so that it is assigned to one of the worker threads with the fewest connections. This threshold can and should probably be optimized for the use case in production.[6] Once a connection is assigned to a specific worker, it cannot move to another worker. So this kind of work distribution is not influenced by the actual load on the workers. This creates a similar problem as discussed in the section about shared-nothing architectures:

[5] So basically which are added **after** calling “epoll_wait”.

[6] For my benchmarks I set this threshold to one.

If the work is almost uniform randomly distributed among the connections, this static work assignment is probably ideal because it creates no overhead during runtime after the connection was assigned to the worker thread. However, if that is not the case, work might be distributed unevenly, which could lead to worse throughput and latencies, especially in high load scenarios.

NGINX has the ability to start multiple worker processes and analogous to KeyDB, each worker process has a reactor with its own stateful system selector instance (e.g. epoll instance). Connections are distributed by the kernel in NGINX. The listener socket is duplicated on "fork()" and therefore shared among the processes. In scenarios with smaller load in Linux this leads to the kernel handing most new connections to the same worker process because Linux unblocks the last thread that called "epoll_wait". This worker process accepts and serves the requests, resulting in comparably high load on that one worker process. To achieve fair scheduling, it is possible to use the "SO_REUSEPORT" option on Linux [24]. "SO_REUSEPORT" makes it possible to create a new socket that listens on a port which is already in use. Thus a new socket structure in the kernel is created. The kernel "round-robin" queues incoming connections onto socket structures. So when each worker has its own socket structure, connections are distributed "round-robin" as a consequence. However, this can produce a higher variance on response times compared to the conventional "dup" on the same socket structure [53]. Sharing the listener socket structure corresponds to a single-queue, multi-server model, while creating a socket structure for each worker corresponds to a multi-queue, multi-server model. So the results are in line with the theoretical superiority of the single-queue, multi-server model.

Creating multiple event loops in "libuv" also results in multiple "epoll" instances and therefore multiple work queues similar to KeyDB's and NGINX's approach. However, the library libuv also offers another paradigm for introducing parallelism to the Reactor Pattern: As mentioned previously, all of these programs/libraries use threads for dealing with file I/O. These threads are usually implemented as thread pools. Thread pools have the advantage of setting an upper limit for running threads and threads within the pool do not need to be created and destroyed repeatedly, which is an expensive operation in the context of high performance server applications [40, pp. 370–371]. Depending on the threads' stack size, these advantages usually come at a relatively low memory usage trade-off. "libuv" makes it possible for the application to queue arbitrary work onto these thread pools, using the "uv_queue_work" function. "uv_queue_work" takes a data pointer for sharing state, a work callback, which

```
thread_create(
        while !shutdown:
            # process pending writes from queue
            # on EAGAIN install write handler
            handle_pending_writes(event_loops[thread_id].write_queue)
            # using stateful event notification interface e.g. epoll
            # each worker thread runs its own event loop with a dedicated
                stateful_poll instance
            events = stateful_poll_wait(event_loops[thread_id].stateful_poll_fd,
                timeout)
            # synchronous event demultiplexing
            for event in events:
                if event.fd == event_loops[thread_id].listener_fd:
                    # distributes connections to matching thread
                    event.fd.accept_on_thread()
                if event.mask & READABLE:
                    # when query is executed push result into the pending write queue
                    event_loops[thread_id].connections[event.fd].handle_read_event()
                if event.mask & WRITABLE:
                    # interest is registered (rarely) if write was
                    # previously blocking due to level-triggered notifications
                    event_loops[thread_id].connections[event.fd].handle_write_event()
, thread_count)
```

Figure 3-4: Simplified pseudocode for KeyDB's multithreaded event loop(s).

can contain "blocking" computation, and a completion callback. In this context "blocking" refers to computations that make the event loop wait long enough, and thus "blocks" the event loop from processing events and making progress, which is the reason why it should be offloaded. This could be "real" thread blocking computations, such as an external library, which makes use of blocking file descriptors, but also something computation-intensive like request sanitizing, for example.

3.2.3 Work Stealing Scheduling - Tokio

In comparison to KeyDB, NGINX and libuv, the asynchronous runtime Tokio has a more sophisticated architecture:[7]

Tokio is written entirely in Rust and differs from the other approaches that were mentioned in the way that it embraces concurrent programming by leveraging the concept of lightweight

[7]Tokio's runtime has a modular design and offers different runtime strategies. Depending on the use case, it can be built with the different feature sets. This thesis generally refers only to the multithreaded Tokio runtime with enabled network functionality, which is Tokio's default.
It should also be noted that Tokio is still a relatively young project and the API was only recently stabilized with the 1.0 version in December 2020. From my experience, a lot of information on implementation details one can find on the internet is outdated and should be taken with a grain of salt. The code in the mainline is the only source that can be trusted. Tokio is probably the most complex and ambitious project of all the projects addressed in this thesis, so it can be quite hard to grasp the essential logic that drives Tokio by taking a quick glance at the source code.

Rust futures. Other “higher level” asynchronous runtimes, such as the runtime of the popular Go programming language, often use fibers[8] instead. “Futures” deserve their own section, but to put it simply, they act as a proxy for computation that has yet to happen. In Tokio a task logically represents a chain of non-blocking futures that is scheduled cooperatively in user space by “yielding” back to the runtime. Tasks are a unit of work that can be run concurrently. The Go equivalent of a Tokio Task would be the “goroutine”. This concept is also known as “green threading” or N:M threading. Similarly to “goroutines", Tokio Tasks can communicate via channels [76]. Tokio is included in this section because the futures within its tasks are state machines generated at compile time, which are dynamically scheduled at runtime by a relatively lightweight scheduler, bringing the performance of applications developed with Tokio to a level comparable to what “system-level” implementations, such as the previously mentioned applications and library abstractions in this section, offer. This section focuses on the reactor and scheduling in Tokio. The reactor uses one instance of a stateful I/O selector such as “epoll” with edge-triggered notifications and shares it among the worker threads. So Tokio leverages the single-queue, multi-server approach, which is superior to the multi-queue, multi-server approach in theory, in regards to “epoll”. However, the difference is that the thread, which receives the batch of ready events by calling the I/O notification selector, does not necessarily dispatch the associated callbacks directly. Instead the thread calls the “wake” method of the ready descriptors. The call to “wake” pushes the associated task into the run queue of the worker which provided the “waker”. Each worker thread has its own bounded run queue for runnable tasks. When this queue is full, work is pushed onto a global multiple-producer, multiple-consumer queue. By default, the Tokio runtime spawns one of these worker threads per CPU core, so Tokio also embraces the thread-per-core model. Each worker pops off tasks of its own run queue and processes them by polling the top-level future of each task until the queue is empty. When the queue is empty, the worker tries to “steal” half the tasks of every other worker onto its own run queue, starting with a random worker. If the worker succeeds in “stealing” half the tasks of at least one of the other workers, the worker continues the processing of tasks. If it does not succeed, it falls back to checking the global queue for tasks. In either case, after the worker successfully “stole” tasks, it goes back to processing the new tasks in its own run queue before calling the I/O notification selector. In case no worker has any tasks that could be stolen, the worker calls into this selector for new I/O notifications immediately and starts the next event loop tick. As a side effect,

[8] Also known as stackful coroutines

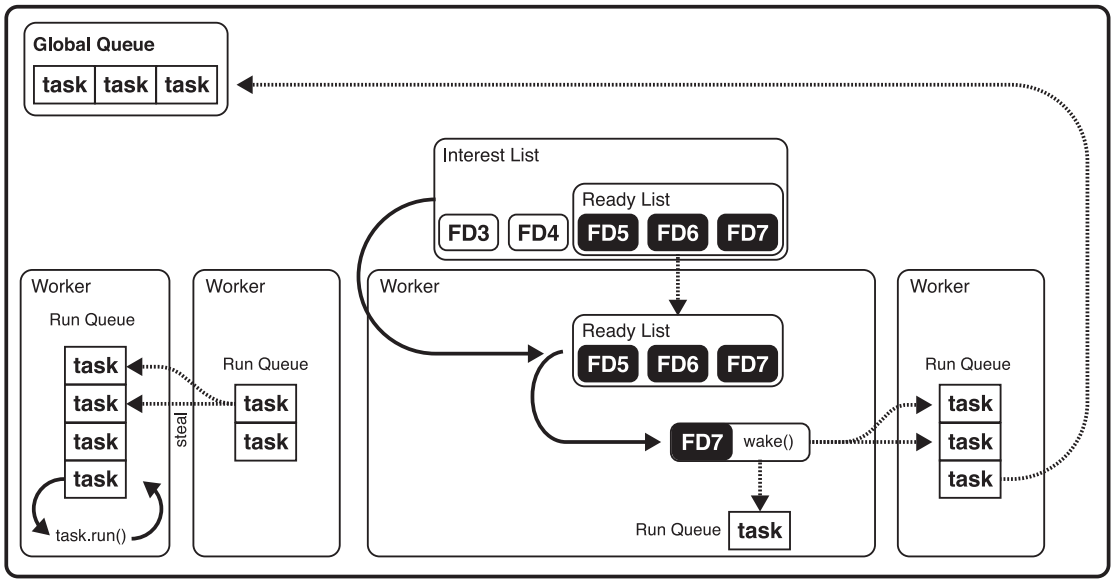


Figure 3-5: Simplified schematic of Tokio's main components and the different worker states.

tasks are generally processed within one global event loop tick. Figure 3-5 shows a simplified schematic of Tokio's main components and the different worker states. By implementing this work stealing dynamic scheduler, Tokio tackles one of the problems described earlier:
The I/O notification selector returns a "batch of work" under load, so even though the shared "epoll" instance with edge-triggered notifications is essentially a shared queue, workers receive multiple tasks from that queue at once. A scheduled task, which requires relatively long computation, can still block tasks scheduled for later execution within the run queue of a worker thread (head-of-line blocking). One solution to this problem could be to push **all** the tasks onto a single shared queue and let N consumer worker threads pop off one task at the time. However, the Tokio developers ruled this solution out due to relatively high contention on a single queue [50]. That is the reason why in the Tokio implementation each worker has its own local run queue to work around the contention on a global queue. To counteract the disadvantages of a local queue, underutilized workers with an empty work queue "steal" from workers which still have runnable tasks. This "work stealing scheduling" counters underutilization, potentially decreases queueing delays the "stolen" tasks might face otherwise, and generally contributes to more uniform load distribution among worker threads. The goal of work stealing is to optimize for busy workers that make progress without generating too much overhead in the common case [40, p. 380]. So making the process of workers popping tasks off of their own local queue as cheap as possible is key to avoid bottlenecks that are caused by the scheduling. However, it is acceptable if "stealing" from another worker's queue is a more

demanding operation in comparison. The queue implemented by Tokio is a bounded circular SPMC queue, which is very similar to the SPSC queue presented in the section about concurrent queues and memory barriers. The main difference is that because this SPMC queue implementation allows for concurrent dequeuing, it requires an atomic CAS operation in the pop method to make sure that the updated head is only "released" and the task is returned, if no other thread has concurrently updated the head.

One more issue Tokio tackles is the granularity of the tasks: A uniform and comparatively fine granularity of tasks reduces the complexity of fair scheduling and makes the runtime more predictable. So the team behind Tokio suggests that developers insert yield points, which give control back to the scheduler, in between relatively demanding computations in their applications and libraries if possible [64].

Alternatively, computationally intensive or blocking computations can be offloaded to a thread outside of Tokio's thread pool with the "spawn_blocking" function. The result of the blocking computation can be non-blocking asynchronously awaited until completion within a Tokio task that runs on a normal worker thread.[9]

In Tokio each "await" within a future on another asynchronous future is also a potential yield point. The worker drives the chain of futures of each task in the run queue until it cannot continue making progress. As a result in certain conditions the worker may continuously poll a task's futures for a comparably long time until one future in the chain is not ready anymore.

For example, if there is a large socket buffer in the kernel that is ready to be read from and the read buffer in user space is relatively small in comparison, a loop within the future of the task, which reads from the socket, does not break and yield to the worker's scheduler until the socket buffer is drained (EAGAIN). To reduce the effect of such a "not-yielding" task, which can lead to "head-of-line blocking", the Tokio runtime assigns an operation budget to each task per "event loop tick". This operation budget is a simple integer, which is decremented with each poll on a task's future. When this counter is zero, the task automatically yields to the worker's scheduler. The Tokio authors claim that adding this operation budget reduced response time percentiles to a third of their original values in some cases [51].

The pseudocode in figure 3-6 demonstrates the main logic that drives the Tokio runtime.

The Tokio runtime is supposed to be a compromise between fair scheduling, whose goal is to distribute load uniformly among all the available cores, and high throughput performance.

[9]The effect is similar to the "uv_queue_work" function in libuv.

```
thread_create(
        while !shutdown:
            # poll on each (probably) ready future
            # for progress until the run queue is empty
            while task = run_queue.pop():
                task.future.poll(&context)
                # as long as there is budget remaining and
                # a task exists in the "lifo_slot" keep polling.
                budget = INITIAL_BUDGET
                while lifo_task = lifo_slot.take():
                    if !budget--:
                        run_queue.push_back(lifo_task)
                        break
                    lifo_task.future.poll(&context)
            # try stealing work from the other workers
            if try_steal_into(&run_queue):
                # if successful continue
                continue
            # check for work in the global queue
            if task_global = global_queue.pop():
                # if successful continue
                run_queue.push_back(task_global)
                continue
            # using stateful event notification interface e.g. epoll
            # reactor with shared stateful_poll instance
            # on event on fd wake the corresponding tasks
            events = stateful_poll_wait(stateful_poll_fd, timeout)
            for event in events:
                # wake will push each associated task into the
                # run queue of the responsible worker
                global_scheduled_io_ressources[event].wake()
, thread_count)
```

Figure 3-6: Simplified pseudocode for Tokio's work stealing scheduler.

Multiple benchmarks of ACTIX-web, which makes use of Tokio, against other HTTP frameworks can be found on the internet and it seems to perform very well [5]. Some benchmarks even claim that ACTIX's throughput is two times higher than that of equivalent Go implementations and that its performance in response times is also better [81].
In a uniform load scenario the maximum throughput of Tokio is probably slightly lower compared to approaches where work distribution is rather static, like in KeyDB, due to the user space scheduling overhead. However, when the load produced by connections is non-uniform, Tokio should in theory yield improved tail latency results, which can mainly be attributed to the work stealing scheduling strategy and the limited task operation budget per tick. In some cases overall throughput might increase as well due to a better utilization of the available CPU cores.
Tokio's developers have implemented an incomplete Redis server (and client) named Mini-Redis [75]. This application is rather a "toy", which serves to demonstrate some of the capabilities and features of Tokio. As it comes in handy, the benchmarks in chapter 4 include Mini-Redis, but the results should be taken with a grain of salt because deep optimizations are certainly not the highest priority in a "showcase application".

3.2.4 Thread-Per-Core and io_uring - Glommio

Glommio is a young thread-per-core I/O library, which is written in Rust just like Tokio and also ships with a runtime focused on asynchronous computation and based on lightweight Rust futures. It should be noted that Glommio is currently in its early stages of development. It is the only library mentioned in this chapter that already takes advantage of the new asynchronous I/O API "io_uring", which is available only for Linux at the moment[10]. Unlike Tokio, Glommio embraces the shared-nothing architecture. So similar to KeyDB each worker runs a completely independent event loop. In Glommio, the way this event loop works vastly differs from how it works in the previously mentioned implementations because of the truly asynchronous programming paradigm of the "io_uring" API. The reactor registers three "io_uring" instances per CPU core in Glommio: The main ring, which should be used for the main I/O operations, the latency ring for low latency I/O operations and the poll ring, whose main focus is polling fast storage devices, such as NVMe flash memory drives, for data. So Glommio relies on the multi-queue, multi-server model, which, in theory, is the inferior model, but has the advantage of no contention on the queues in practice .

[10] January 2021

This is important because the goal of Glommio's shared-nothing thread-per-core architecture is almost linear scaling and contention would hinder the achievement of this goal. [11]

Glommio's runtime is similar to Tokio's in that both rely on cooperative scheduling. Developers can create task queues with varying "shares". The "shares" are used to proportionally assign CPU time under load. The more "shares" a queue has, the more CPU time is given to the queue. The function "yield_if_needed" should be deployed within large tasks to yield control back to the scheduler. "yield_if_needed" checks if there is pending data to process at the latency ring. Since "io_uring" maps the submission and completion rings into user **and** kernel space, this operation is a lightweight integer comparison of head and tail values at the "latency completion ring". No system call is required. Latency sensitive events in this latency ring can be prioritized over events in the main ring. When there are no events on either of the rings, Glommio registers notifications for the latency ring's file descriptor with the main ring and the thread can safely be blocked by calling "io_uring_enter". It is up to the developer to properly configure the tasks with regard to computation time and latency requirements. The previously mentioned applications and library abstractions do not actively prioritize certain tasks over others, but rather try to complete all tasks with the lowest latency possible. This is a paradigm shift in Glommio. Finding the right latency requirements for a use case in a potentially large distributed fan-out architecture in the first place and mapping these use cases to the proper underlying tasks in Glommio is a non-trivial problem. In the case of Glommio the concept of encapsulation is partially abandoned in exchange for a better performance.

In shared-nothing architectures like Glommio, state is usually partioned. Consequently, in a database application, which is split by key ranges, the state where each key range is stored[12] also has to be kept somewhere. In Redis Cluster this state is stored on the clients and each node has its own independent listener port because it is a "standalone" Redis server instance. Consequently each Redis' Cluster client establishes a connection with each node in the cluster. In the shared-nothing thread-per-core database, which is mentioned in the section about the thread-per-core paradigm, the state is stored on the database server and queries are redirected to the responsible thread via user space message passing. The advantage of this approach is that the server can listen on a single port, thus clients establish only a single

[11] Modern x86-64 Server CPUs usually have some sort of simultaneous multithreading (SMT), which results in two vCPU cores per CPU core. On a x86-64 CPU with enabled SMT one CPU core would be responsible for six "io_uring" instances in this architecture. It would also be interesting to see how the kernel handles the large amount of rings in a system with a high CPU core count and the effect of enabled IRQ affinity.

[12] In real implementations this is probably rather a hash slot than a key range.

connection to the database.

A database developed with Glommio could also listen on a single port by creating a separate non-shared listener socket for each thread using the "SO_REUSEPORT" in Linux [24]. "SO_REUSEPORT" makes it possible for each listener socket to have its own queue, although the listener sockets "listen" on the same port. As discussed in the section about NGINX, incoming connections are distributed "round-robin" onto the queues of these dedicated socket structures. However, the Glommio database would not use message passing to redirect queries. Instead, Glommio's developer suggests to make use of "extended Berkeley Packet Filter" (eBPF) programs in the kernel to process and route each packet to the corresponding socket structure and thread, which is responsible for the key [31]. In essence eBPF programs can "hook" into designated code paths in the kernel and execute custom logic. The eBPF code itself runs safely inside of a virtual machine [38]. So the process of routing to the corresponding socket is handled entirely within the kernel. Therefore no redirection in user space is required.

3.3 Rust Futures and zero-cost async-await

Highly concurrent network-bound applications try to fully capitalize on the provided hardware. However, "lower level" languages such as C and C++, which are closer to the hardware, do not ship with any abstractions or ecosystems that support the implementation of high concurrency applications. Furthermore, they are usually considered rather "unsafe" to work with for reasons that are beyond the scope of this study. So entire language ecosystems have been built on the premise of simplifying concurrent programming, such as Erlang or Golang. They all give up some control over the hardware for improved ergonomics and safety, such as a garbage collected runtime[13] and higher level concepts like "fibers" for green-threading, which usually create overhead. The Rust ecosystem provides a proof of concept that highly concurrent and highly performant close-to-hardware code does not have to be "ugly" or "unsafe". In fact, one of Rust's core features besides a performance similar to C/C++ and "memory-safety at compile-time" is "fearless concurrency". In Rust one of the fundamental building blocks of concurrency, which is especially relevant for network applications, is the concept of futures and zero-cost async-await.

Futures are data structures that act as a proxy for the result of a computation that is not

[13]Garbage collection can be particularly problematic, because it can have a negative impact on tail latency results.

yet complete [28]. Rust's "future ecosystem" is implemented as zero-cost abstractions and should be considered when building high performance applications, such as highly concurrent network-bound server applications. In practice, the "Future trait"[14] provides the possibility to implement efficient concurrency that "asynchronous runtimes", such as Tokio or Glommio, can rely on.

When Rust futures are polled they either resolve the result of the computation or return "pending", which signals that the computation is not ready yet. Consequently, each future has to implement a poll method, which is the function that drives the future's progress and that either returns the enum type "Poll::Ready" containing the result[15] or "Poll::Pending" [27].

Rust's futures are the foundation for Rust's async-await syntax. Rust's async-await syntax is syntactic sugar for composing one "big" future out of multiple futures. Each "await" represents a poll on a future in the chain and a possible yield point.

"Async" functions are transformed at compile time into functions which return a future. The returned future is essentially a generator, which is created at compile time and then turned into an optimal state machine. This state machine contains all the necessary data within the states and the logic for acting upon that state for driving the futures. Conceptually, the state machine is a union type over all possible states and therefore its memory footprint is roughly the size of the largest state. Mapping variables within an "async" function to the states efficiently requires a liveness analysis.

The basic concept behind the liveness analysis is that for each variable at each yield point it ensures within an "async" function, that the variable is stored within one or multiple states, if it can prove that a value written to the variable before the yield point[16] might be read after the state transition. If two variables within an "async" function are in use at the same time, the memory that stores the two variables within the states must be non-overlapping [54].

However, the async-await syntax hides all of that complexity from the programmer and by chaining futures makes composing asynchronous logic trivial, while not sacrificing the performance of the application by leveraging zero-cost abstractions and compile time optimizations.

Futures in Rust are "lazy" and cannot make progress without being polled. This strategy is optimal for enabling concurrency on a single processor [8]. The executor, which itself might be driven by a reactor like in Tokio, is responsible for polling the top-level future owned by a

[14] "Interfaces" in other imperative programming languages are similar to Rust's "traits".

[15] "Enums" in Rust can store entire structures, so these "enums" can serve as union types, too.

[16] Each yield point is a possible state transition.

task at the right time by calling the poll method and passing a "context". However, nothing happens if a future is polled when the computation is not ready yet, apart from wasting CPU time. A needless poll just returns "pending". The context that is passed as an argument to the poll function of the future contains a "waker". This waker is responsible for notifying the executor once the corresponding task is ready to be polled and is passed with each poll because futures can be moved to different tasks with a different waker. When the executor is notified it has to poll the top level future of the task at least once to ensure that it can continue to make progress [26].

Network applications usually deal with streams of data. Streams in Rust's future ecosystem represent a series of values that are produced asynchronously, so they are essentially futures that can yield multiple asynchronous results [12]. One use case for a stream could be a TCP connection.

3.4 SIMD for Network Applications

Multiprocessing is not the only way to achieve parallelism on modern CPUs. As the name suggests SIMD (single instruction, multiple data) allows the computer to execute the same operation on multiple data points in parallel. Modern instruction set architectures (ISAs), such as x86-64 and ARMv8, offer a set of SIMD instructions and corresponding registers for applying the same operation to a vector of packed integer or floating point values with a single instruction. This approach is more efficient than executing the same instruction multiple times sequentially and therefore leads to better performance and higher energy efficiency [2, 39].

SIMD instructions were added to Intel x86 CPUs in the form of the MultiMedia eXtension (MMX) in 1997. These instructions were originally targeted, at high demanding multimedia applications, such as 3D applications and videos, to speed up the vector math. In the following decades Intel added several iterations of Streaming SIMD Extensions (SSE) and later Advanced Vector Extensions (AVX) to their x86 CPUs, with each major iteration adding more and wider registers as well as more instructions [43]. Latest generation Intel CPUs have 32 512-bit wide AVX-512 registers per CPU core.

Modern compiler backends such as LLVM or the GCC backend can do their best to substitute operations on normal arrays with SIMD instructions automatically. In the case of Clang and GCC such "auto-vectorization" optimizations are applied on optimization level three (flag

"-O3") by default or manually with flag "-ftree-vectorize" enabled [25, 13].

The best way to make sure that certain logic in the application truly benefits from SIMD is to optimize the code in assembly "by hand" or alternatively use "SIMD intrinsics". This might also require rearranging the data structure layout. The best case for SIMD computation is the structure of arrays (SoA) processing method as the structure's data can be loaded directly into SIMD registers, while the more traditional array of structures (AoS) arrangement often requires loading data from non-contiguous memory into a SIMD register [41]. Modern vector architectures support non-contiguous loads, so-called gathers, but they come along with a trade-off in performance compared to contiguous loads [39].

Almost all x86-64 CPUs in use today support AVX2 and can therefore load one half of a cache-line into a 256-bit wide register and compare the values of two registers at 32 bytes each with one instruction in parallel, for example. The application of SIMD is not limited to conventional vector math. This provides the opportunity for significantly improving performance on string or buffer operations, such as protocol and query parsing or table scans, which are common in network applications. In the past years, research has been done regarding efficient vectorizing of such algorithms.

The hash table[17] is a standard data container that has a use case in almost every network application. For example, a single Redis server instance instantiates 95 hash tables on startup and the core database is implemented as a hash table.

A few years ago Google noticed that significant portions of the execution time in their applications is spent with processing hash tables. So Google started implementing their own family of more optimized hash tables named "swiss tables". Before that Google mainly used the hash tables from the C++ standard template library such as "std::unorderd_map". In typical hashmap implementations linked lists or tree structures are used to chain entries when hash entry collisions occur. This leads to cache misses on typical operations. The "absl::flat_hash_map" of the Google "swiss tables", on the other hand, is implemented as a flat probing table with a seperate array, which contains a byte of metadata for each entry in the table. Each metadata byte contains 7 bits of the 64 bit hash code of the key of the corresponding entry and 1 bit which is reserved to signal if the entry is empty, full or deleted. The other 57 bits of hash code are used to find the start of the "bucket chain" in the probing table. The slice in the metadata array, which contains the metadata of the corresponding entry, fits entirely into the first level cache and is scanned using SSE instructions with 16

[17] Also known as hash map or dictionary

candidates in parallel [10]. The find operation on the hash table returns the matching candidate or candidates. Because seven bits of the hash code already match, the requested entry is likely among the candidates. Google claims that the migration to the optimized "swiss hash tables" reduced the percentage of fleet-wide CPU utilization by half a percent, while RAM was reduced by about one percent [48].

Network applications often spend a considerable amount of time parsing data exchange formats such as the JavaScript Object Notation (JSON). Historically, JSON parsing was mainly done character by character in most implementations. The library "simdjson" changes this paradigm and instead implements a JSON parser that makes use of SIMD and branchless programming techniques to parse whole vectors of bytes on a single core in parallel. On a modern Intel x86 Skylake CPU "simdjson" is able to parse multiple gigabytes of data on one core. This results in a 2.1 times speedup when parsing the "Twitter JSON benchmark" compared to other popular open-source implementations [49]. [18]

Bloom filters are probabilistic data structures that can be used to efficiently check if a value is (probably) present in a set of values. They have many possible purposes in network driven applications, such as databases. For example, they are useful to speed up look-ups in sorted string tables (SSTables). Polychroniou's et al. 2014 paper "Vectorized Bloom Filters for Advanced SIMD Processors" describes the results of tests conducted to find out how the use of SIMD on Bloom filters affects the performance. The tests demonstrated a speed up of "bloom filter probing" by 1.4 to 3.3 times using Intel AVX2 SIMD instructions [57].

Network applications using SIMD can definitely benefit from parallelism on a single core. Generally, when data is processed in a loop and iterations are not directly dependent, vectorization utilizing SIMD might be profitable. However, it should be noted that this often requires a complete change in the way programmers think about algorithms and data structures. Algorithms heavily relying on SIMD are often hard to comprehend at first glance because programmers are usually not used to this model. One good example of this is the "simdjson" project.

[18] Redis and KeyDB do not utilize SIMD for protocol and query parsing at this point in time. There is no information suggesting that NGINX makes use of SIMD for parsing HTTP. Especially in NGINX, SIMD might be a great opportunity for potential performance improvements.

Chapter 4

Performance Evaluations - Methodology

4.1 Approach

4.1.1 Overview

While chapters 2 and 3 have a rather qualitative take on parallelism in highly concurrent network-bound applications, this and the following chapter's main focus is on gathering and evaluating quantitative data by using experiments to distinguish key characteristics of some of the implementations that are described in the chapter about architectures & paradigms. The goal is to verify some of the research findings presented in chapter 3 by performing experiments and examining the results. In the next section the essence of the research in regards to the selected applications is reviewed. Quantitative benchmark experiments are the tool of choice for testing the potential real world performance of a system.

First of all, the benchmarks in this thesis are supposed to demonstrate that even the throughput of so-called network-bound applications can be bottlenecked by a single thread of execution. Secondly, the benchmark results should reflect the differences in how various highly concurrent network-bound applications incorporate parallelism into their implementation. It is important to note that benchmarks are always biased to a certain degree because they test a finite amount of scenarios.

Further subsections discuss the selection of the candidates, the quantitative metrics, which build the basis for the performance evaluation, and the extensive pre-testing.

4.1.2 Candidates

In this study the selected application type of choice is in-memory databases. Common belief among developers is that in-memory databases are typically prone to be limited by the network bandwidth and therefore parallelism within the main logic is not necessary. In-memory databases do not require large amounts of CPU intensive calculations and are not limited by the performance of other resources such as the speed of the secondary storage, which might affect the performance of other highly concurrent network-bound applications like web servers.

The candidates of choice are the in-memory databases Redis [61], KeyDB [72], Mini-Redis [75] and Redis Cluster.

Redis is among the candidates because it is the most popular open source in-memory database and it just recently introduced (optional) parallelism into the main logic. Before that there was a controversy in the developer communities about the necessity of parallelism for Redis. Redis without I/O threading is a plain single-threaded reactor. With I/O threading enabled, Redis' main logic could be described as a synchronous producer-consumer reactor: On each event loop tick, read and write operations are distributed to consumer threads and the main thread synchronously waits for them to finish. This approach is supposed to perform the worst due to the overhead associated with initializing the I/O threading, the expensive task distribution and the inefficient busy-waiting of the I/O threads.

KeyDB, the open-source multithreaded Redis fork, represents the multithreaded event loop model with a dedicated "epoll" instance per worker thread and static client assignment. This approach should produce the best throughput results among the shared-something approaches due to the multithreaded event loop and static client assignment, which has almost no runtime overhead.

Mini-Redis makes use of Tokio and therefore represents an in-memory database with work-stealing scheduling within a lightweight asynchronous runtime that is driven by a single "epoll" instance. Since Mini-Redis is rather a "showcase" application that is developed by the team behind Tokio, Mini-Redis is the "wild card" among the candidates. However, according to the research, Tokio's work stealing scheduler and task budgets should yield the best results in tail latencies.

Redis Cluster represents the strict shared-nothing approach. There is no state shared between the nodes in the cluster during runtime. Even the state of the hash slots is stored on the clients. For the purpose of simplification, in this context thread is equal to node and a node

is an operating system process that runs a single-threaded Redis server instance. To make a fair comparison, a Redis Cluster with N nodes is always compared to one of the other applications with N threads, if not stated otherwise.

In uniform workload scenarios Redis Cluster should provide the best throughput results. There are no shared resources, which means that there are no locks on the core hash tables, client structures or elsewhere. There is no contention on socket or "epoll" structures within the kernel and no explicit coherence traffic because there is no shared memory. Minimizing differences, which are not of interest for this thesis, is an important requirement, so all of the selected candidates have to support the same "Redis Serialization Protocol" (RESP) and this is the reason why the second most popular in-memory database "memcached" is not included in the tests. A single-threaded Redis instance serves as the point of reference for the following benchmarks.

4.1.3 Performance Metrics

As already hinted in the previous subsection, the performance metrics for the experiments in this study are throughput and response time percentiles.

Throughput is usually measured in operations per second (OPS). In a highly concurrent network-bound application an operation is usually a query or a request. Other more specific terms to describe OPS are queries per second (QPS) and requests per second (RPS). They are often used interchangeably.

However, I decided to measure in bytes per second (B/s) for the following reasons: In the upcoming benchmark all applications have to process the same queries with the same payloads and they use the same protocol (RESP), which implies the same protocol overhead. For this artificial benchmark, specific QPS numbers are not of interest anyway. Instead, it is more interesting to demonstrate how close the candidates can get to the hard limit of the network, which is defined by the NIC and the network bandwidth. Measuring in bytes per second makes it easier to compare the candidates in regards to this.

High response times percentiles are the performance metric to look out for in large distributed fan-out architectures. They directly affect the user-experience of a service. In sequential execution response times add up and even in parallel execution the slowest request sets the hard limit for the response time of the main request. Response times are measured in percentiles and high percentiles are known as tail latencies. Common percentiles are the P95, P99 and P99.9 percentile. They represent thresholds at which 95%, 99% and 99.9% of requests

are faster than that particular threshold. Amazon describes response time requirements for internal services in terms of the P99.9 percentile [47, p. 15]. In-memory databases are usually leaf nodes in such distributed fan-out architectures and services aggregate multiple queries from such databases for one request. So this study even measures up to the P99.99 percentile.

4.1.4 Pre-Testing

Before performing the real benchmarks I wanted to take a closer look at Redis to explore how much computation is required proportionally for internal operations. I developed a small library for monitoring the Redis event loop. This library enables saving event loop ticks and measured time frames within an “event loop monitor” at runtime. Each monitor saves its data inside of a vector in memory during runtime. These monitors are inserted at certain points of interest, effectively creating a modified version of Redis. For example, these points of interest include pending writes to a socket, reading a query from a socket or the execution of commands. When the redis-server is stopped, the content of these monitors is saved to CSV files.[1]

These monitors suggest that Redis spends about 80% of time within I/O operations during an event loop tick under a high load of SET and GET commands as long as the payloads are in the smaller than 10 kilobyte range.

Roughly 7% [2] of time within an event loop tick is spent on the execution of the commands.[3] Write operations with payloads of 100kb or more are substantially more expensive in Redis. The histogram in figure 4-1 displays the time that is required for executing the "writeToClient" function with varying payloads in Redis. This information is interesting for the final benchmarks because “throwing in” requests with larger payloads into the request mix likely contributes to skewing the task granularity.

In KeyDB the access to the core hash table is guarded by a global lock. It is a custom implementation in assembler in the form of a hybrid Spinlock named “fastlock”, that calls “futex wait” after spinning for a lot of iterations and that is optimized for the common case of short lock intertervals. Mini-Redis does also lock the access to the core hash table, but Mini-Redis uses a “std::sync::Mutex”, which is implemented with futex on Linux.[4]

The measurements can be utilized to approximate p, which is the proportion of execution

[1] I did not use this version but instead an unmodified version of Redis for the “memtier” benchmarks.

[2] 6% for GET requests, 8% for SET requests

[3] This was tested on the final benchmark setup.

[4] Interestingly, the core hash table in Mini-Redis is a “std::collections::HashMap”, which is the Rust port of the “absl::flat_hash_map” developed by Google that is discussed in the section about SIMD in chapter 3.

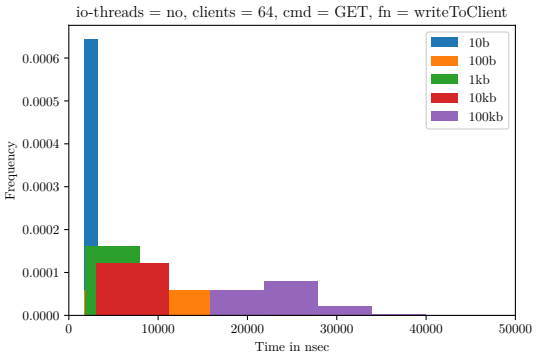


Figure 4-1: Histogram of time required for executing "writeToClient" function with varying payloads in Redis.

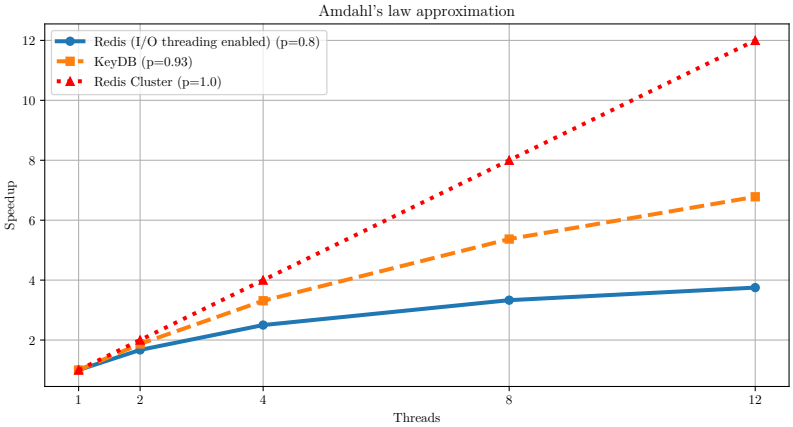


Figure 4-2: Approximate speedup of Redis with I/O threading (p=0.8), KeyDB (p=0.93) and Redis Cluster (p=1) according to Amdahl's law.

time that the part benefiting from multithreading originally occupied. Therefore, the measurements can give some approximate information about the potential speedup that can be expected by the different multithreaded approaches in Redis and KeyDB according to Amdahl's law (see fig. 4-2). This helps for reasoning about the gathered data later on. [5] If the results are very different from the described potential speedup, there is probably something wrong in the test scenario.

[5] While KeyDB and Redis share major code parts, there are still enough differences, that this approximate speedup cannot serve as a definitive hard limit.

4.2 Methods

4.2.1 Overview

This section describes the methods used for gathering the benchmark results.
It deals with the hardware and infrastructure selection, the implementation of the scripts, which are used for automating the execution of the benchmarks, the selection and the configuration of the benchmarks, as well as the configurations used for the candidates.

4.2.2 Hardware and Infrastructure

Cloud computing providers are the leading force in IT infrastructure for distributed computing. Amazon Web Services (AWS) was chosen as the provider for the benchmarks as it is the most popular cloud provider.
Two AWS "m5zn.6xlarge" general purpose EC2 instances were selected each with 24 vCPU cores[6], 96 GB RAM and full 50 Gbps network bandwidth that had Ubuntu 20.04 LTS (Linux Kernel 5.4) pre-installed. The benchmarks were run over a real network within the same AWS Virtual Private Cloud (VPC) and the same AWS Availability Zone (us-east-2b).
The instance type "m5zn.6xlarge" was selected because it helps simulate a realistic scenario with its strong single core performance and relatively high network bandwidth.[7] It has *full* access to 50 Gbps bandwidth, utilizing the Elastic Network Adapter (ENA). That is important because EC2 instance types with "up to ... Gbps" network bandwidth burst performance in certain scenarios, which could lead to distorted benchmarking results. Also headroom in bandwidth comes in handy for demonstrating CPU bottlenecks. The benchmarks should not get too close to fully saturating this bandwidth. The maximum memory bandwidth of the selected instance type is about 70GB/s according to the STREAMS 5.1 memory benchmark.

4.2.3 Scripts

I developed shell scripts that run the benchmarks on all candidates. Amongst other things these scripts aim to configure environments that are as similar as possible for all candidates. So I set each application up to deliver its best performance, at least to my knowledge. The shell scripts are available in the github repository[8] for this thesis. The "main script" runs a "benchmark script" for each application. Redis, KeyDB and Mini-Redis share one benchmark

[6] 2nd Generation Intel Xeon Scalable Processor Cascade Lake
[7] A potential CPU bottleneck should not be provoked artificially, e.g by testing on a "free tier" EC2 instance.
[8] URL: https://github.com/lc0305/bachelor_thesis

script. For Redis Cluster I created a new script because it needed multiple adjustments.
The "benchmark script" for Redis, KeyDB and Mini-Redis performs the memtier benchmark on each thread (1, 2, 4, 8, 12) and client (24, 96, 192, 384, 768) configuration. These configurations were selected because the EC2 instances that were used for the tests had 24 vCPU cores. Using all available cores for the benchmark makes it unlikely that it becomes the bottleneck. So the benchmark was allowed to make use of all the available 24 cores for each run, while the database server was only allowed to use 12 cores, which corresponds to a maximum configuration of 12 threads for the databases. Running the benchmark with a fixed amount of 24 threads meant that the lowest client configuration was 24 clients (one client per thread). The other client configurations were multiples of 24 and went up to 768 clients because I assumed that in-memory databases with more than 768 concurrent clients were unlikely. In the following sections the 24 client configuration is referred to as *low concurrency*, the 192 client configuration as *medium concurrency* and the 768 client configuration as *high concurrency*.
The script iterates over these configurations. With each thread configuration the script starts the properly configured database server on the other target server instance via remote SSH. After each startup, the database serves 2M requests for "warm up" without performance measurements enabled. When the warmup finishes, the script performs the actual "memtier benchmark" runs with 10M requests each on the remote database server for each client configuration. When all runs for a thread configuration are finished, the script "kills" the remote database server. The database server restarts with a new thread configuration in the next iteration.
Redis Cluster requires at least three nodes, so that is why the thread/node configurations in the cluster script start at four nodes. Other noteworthy changes in the cluster script are that it sets up and destroys a whole cluster for each mentioned iteration on the remote server via SSH.

4.2.4 Memtier benchmark configuration

"Memtier" is a popular benchmarking tool for performance testing NoSQL key-value databases [60]. The "memtier benchmark" can spawn multiple threads and each thread independently generates requests for one or more "simulated" client connections. When responses for the generated requests are received, the benchmark sends new requests according to the configuration. This process repeats in a loop and breaks when a certain limit is reached. This limit

could be a discrete request count or a "real" time frame. The benchmark records the throughput (in operations per second and kilobytes per second) and the response time percentiles for the requests. Results are saved to files persistently when the benchmark is finished. The "memtier benchmark" is being developed by the "Redis Labs Team".

In this study, the "memtier benchmark" is configured to send SET and GET queries with a 1:2 ratio and uniform random distribution. 99% of the SET and GET values are configured to have a data size of 100 bytes. The 100-byte requests represent requests with a small payload. Based on the pre-testing, varying payloads below the one kilobyte threshold does not make a big difference. The other 1% of SET and GET values in this configuration have a data size of 100 kilobytes and represent a larger payload.

The pre-testing demonstrated that write operations to the clients are substantially more expensive with the benchmark configured to data sizes of 100kb compared to data sizes below 1kb. Larger data sizes might also lead to Redis and KeyDB installing the write handler:

Redis adds clients with pending writes to the pending writes queue and processes the clients in this write queue before calling "epoll_wait". The write handler is only installed when the non-blocking write returns with "EAGAIN" **or** when the server has sent more than 64kb of data to a client **and** the client has more pending data in the reply buffer within one event loop tick.[9]

A small amount of more demanding requests does not immediately saturate the NIC, but does contribute to skewing the overall task granularity, which is desirable for a benchmark that is also testing tail latency performance. Unfortunately, my tests indicate that these "large" requests are grouped together.

Furthermore, it is not possible to configure non-uniform client load in "memtier" to my best knowledge. However, developing an improved benchmark for NoSQL key-value databases is beyond the scope of this thesis.

In this study "memtier" is configured to write the main data, including the average throughput, to a JSON file and response time percentiles for High Dynamic Range (HDR) Histograms to one more separate file.

[9] A socket is almost always ready to be written to, so when registering write interest with level-triggered notifications, like Redis does, "epoll_wait" always returns immediately. The solution would be to constantly deregister the write interest again with "epoll_ctl" after finishing a reply. However, this was probably ruled out as being too inefficient by the developers.

4.2.5 Redis (Cluster) and KeyDB configuration

Redis and KeyDB have persistence with weak durability enabled by default. This has usually low impact on performance, but can make the Redis process "fork" for log rewriting of the append-only fashion (AOF) files. Since I did not want any contending processes during my benchmark, I disabled AOF and RDB persistence completely by setting the "–save ‘ ‘" and "–appendonly no" arguments.

By default, Redis I/O threading only fans out the write operations to the consumer threads. However, I wanted to measure the full multi-threaded potential of Redis, so I enabled threaded read operations with "–io-threads-do-reads yes".

I did not enable Redis I/O threading for the nodes in Redis Cluster.

Furthermore, I added the "–min-clients-per-thread 1" flag to KeyDB to get rid of the threshold that is described in chapter 3, which KeyDB uses by default for distributing connections to the worker threads.

4.2.6 Mini-Redis configuration

Some minor adjustments are required to make the Mini-Redis Server work with the benchmark tests in this study. Because it is a "showcase application" the "maximum connections" parameter is not configurable, but instead hard coded, so I manually set it to 10k in the code. Mini-Redis had to be recompiled for every thread configuration because the thread count is handled by a Tokio Macro at Mini-Redis' current stage and is not configurable during runtime. In this study, the Mini-Redis server is not linked against the default allocator, but instead against "jemalloc", because both Redis and KeyDB build with "jemalloc" on Linux by default. In-memory databases make frequent use of the allocator and I tried to reduce the influences of differences between the candidates that are not of interest for this thesis.

4.3 Analysis

4.3.1 Overview

This section describes the approach to analysing the data. All the relevant quantitative evaluations are done within "Jupyter-Notebooks". The notebooks are available in the github repository[10] for this thesis.

The goal of the analysis is to show how high concurrency network-bound applications *can*

[10]URL: https://github.com/lc0305/bachelor_thesis

benefit from multi-core CPUs and capture obvious trends in the gathered data. For more comprehensive results, further benchmarks with varying hardware and software configurations would have to be done. So the upcoming results should not be misinterpreted as definitive assessments for the tested candidates.

4.3.2 Throughput

For the evaluation of the throughput it is of interest how much average (mean) throughput (in MB/s) the different candidates produce in regards to varying levels of concurrency and parallelism. The level of concurrency is measured in concurrent clients. The level of parallelism is measured in the amount of threads or processes that the candidate makes use of. In a two-dimensional plot one of these two variables has to be constant, so each combination is plotted twice. For simplification only the 24 (low concurrency), 192 (medium concurrency) and 768 (high concurrency) configurations are selected as constants for the plots if the thread count is the variable.
Heatmaps were used to make substantial differences in throughput results of each client-thread configuration compared to a single-threaded Redis instance visible. The single single-threaded Redis instance serves as the point of reference for all benchmarks.

4.3.3 Response times

A High Dynamic Range (HDR) Histogram is plotted based upon the response time percentile data generated by “memtier” for each client-thread configuration until the P99.99 percentile. Thus the slowest 1k requests of the 10M requests for each plotted combination are discarded. HDR Histograms make differences in tail latency performance visible. Multiple heatmaps capture substantial differences between the selected candidates compared to the point of reference, using the measured P95, P99, P99.9 and P99.99 response time percentiles.

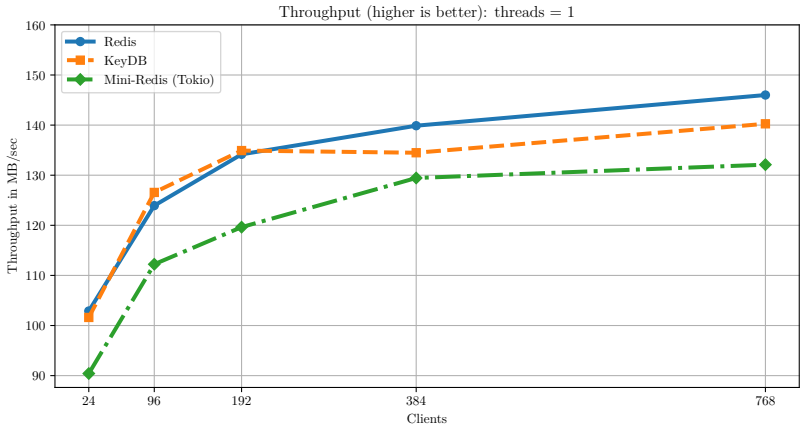
Chapter 5

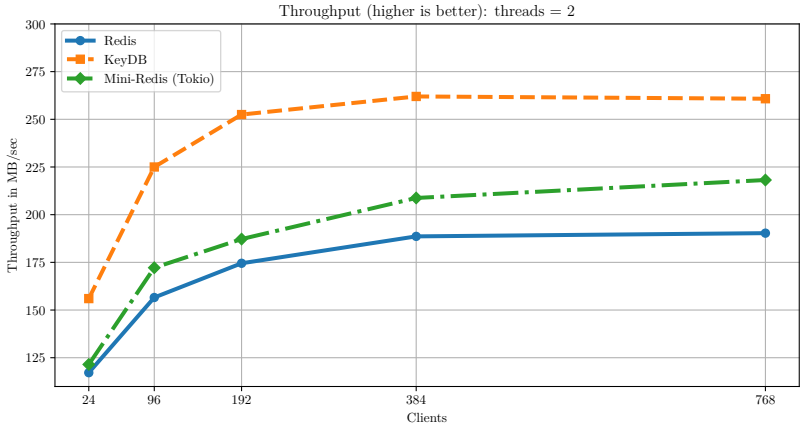
Performance Evaluations - Results

5.1 Throughput

5.1.1 Plots

Throughput: Threads = 1 - 4





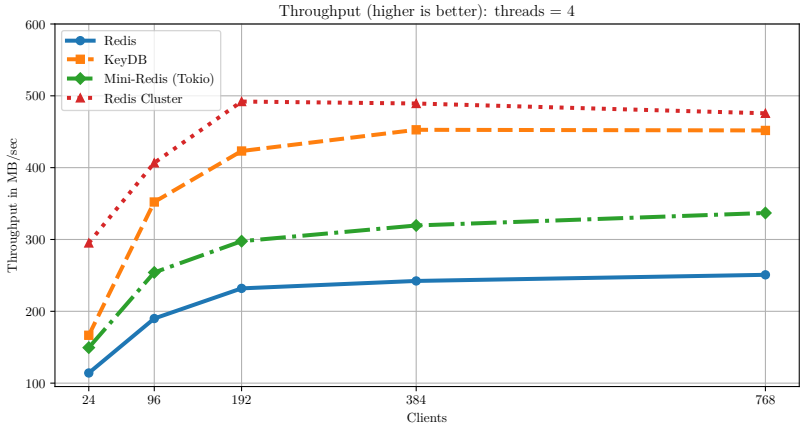
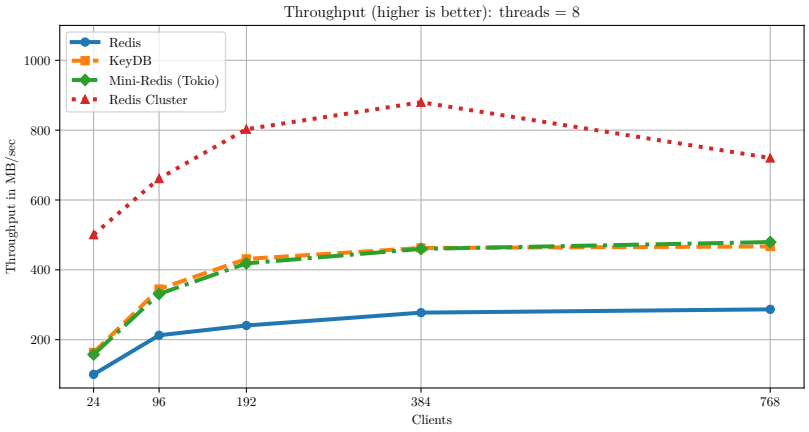


Table 5.1: Throughput with varying client configurations and 1-4 threads.

Throughput: Threads = 8 - 12



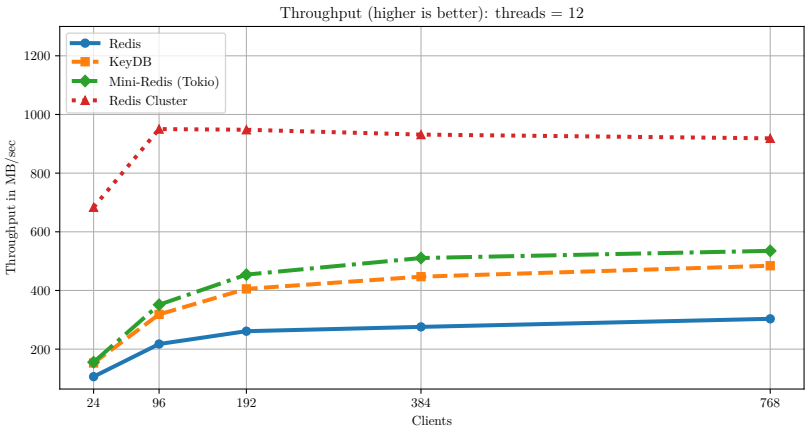
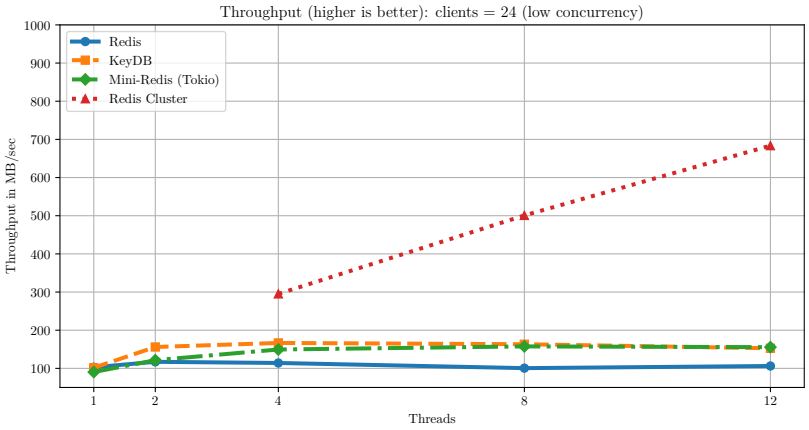
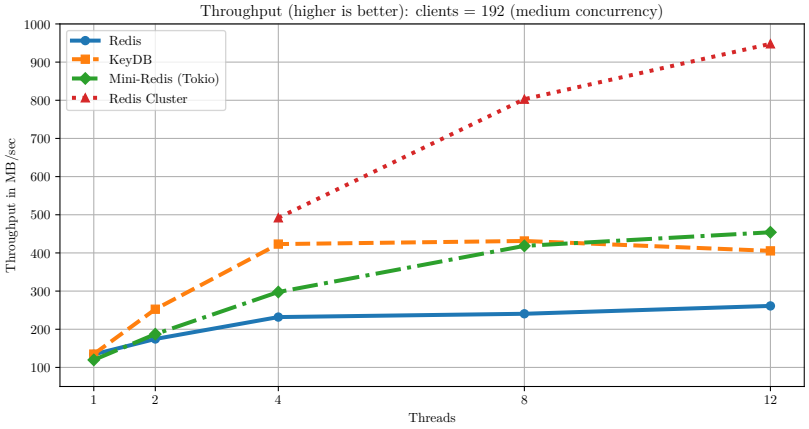


Table 5.2: Throughput with varying client configurations and 8-12 threads.

Throughput: Clients = 24 - 768





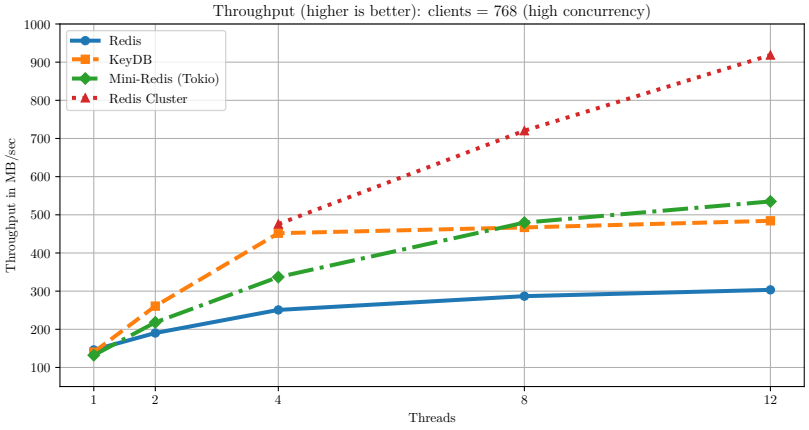


Table 5.3: Throughput with varying client configurations and 1-12 threads.

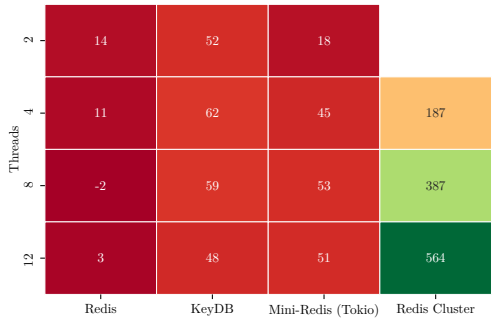
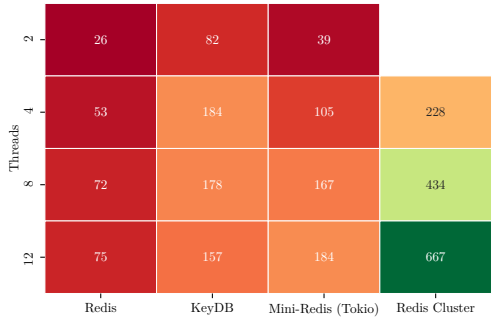
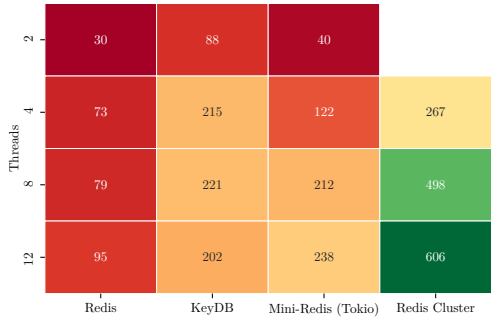
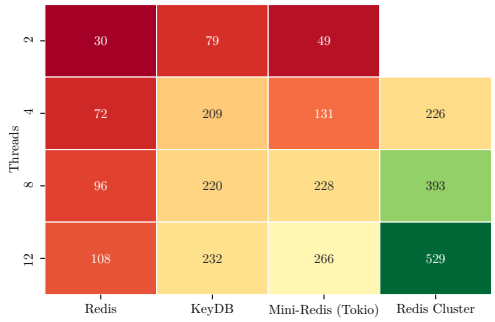
Clients	**Throughput (in %)** compared to reference point (single-threaded Redis)
24	
96	
192	
768	

Table 5.4: Throughput compared to reference point (single-threaded Redis) in % (Higher percentage is better).

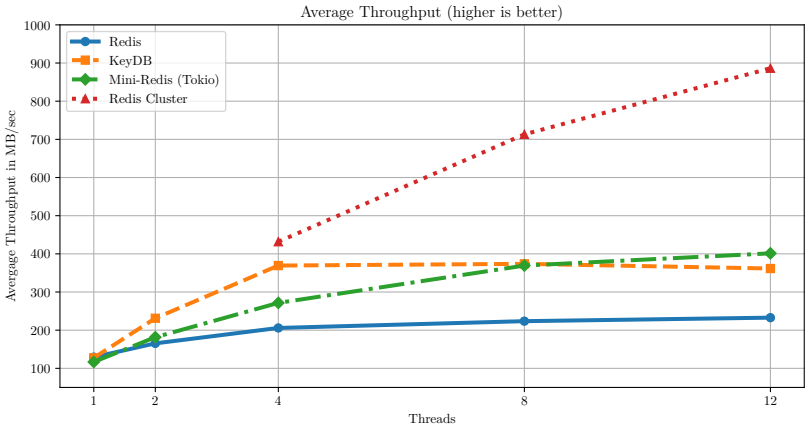


Figure 5-1: Average (mean) throughput of Redis, KeyDB, Mini-Redis and Redis Cluster with varying thread counts.

5.1.2 Key Observations

The shared-something approaches (Redis, KeyDB, Mini-Redis) scale poorly across multiple threads with 24 clients (low concurrency). Redis' I/O threading throughput even degraded slightly with 8 threads compared to the point of reference in this configuration. This can be observed best in the heatmaps in table 5.4. The Redis Cluster on the other hand scaled much better with low concurrency. This can probably be attributed to the fact that each client in cluster-mode establishes a TCP connection with each node in the cluster. So depending on the thread/node configuration, the amount of global concurrent connections on the server is:

$$connections_{total}(nodes) = clients_{configured} * nodes$$

In this study, Redis Cluster establishes four to twelve times more concurrent connections in total than the shared-something approaches.

When increasing the amount of concurrent clients, the shared-something approaches start reaching their potential. At around 192-384 clients the speedup from adding clients plummets. The throughput of the shared-nothing Redis Cluster degrades in all thread configurations at 768 concurrent clients. The graphs in the tables 5.1 and 5.2 shows this very well. This is most likely also related to the higher amount of total connections in Redis Cluster. With 12 nodes the cluster has to serve up to 9216 concurrent connections.

Up to 4 configured threads and under the condition that there are enough concurrent clients, KeyDB lives up to its "marketing hype" in this study: Compared to a single-threaded Redis instance (point of reference) KeyDB delivers a 79-88% increase in throughput with equal to or more than 96 clients and 2 worker threads and it still delivers a respectable 209-224%

increase in throughput with 192 or more clients and 4 worker threads. However, adding more than 4 threads to KeyDB seems to be a waste of resources for the purpose of improving the throughput, because there are no substantial further improvements (See fig. 5-1) and performance even degrades in some scenarios. At this point, it has to be mentioned that KeyDB's developer does not recommend assigning more than 4 worker threads to KeyDB [72]. My educated guess is that this might be related to the complex locking strategies within KeyDB and the custom implementation of KeyDB's "fastlock". Further investigation is required for verification.[1]

Adding threads to Mini-Redis increases the throughput monotonically. Interestingly, when there are more than 24 concurrent clients, the plot of Mini-Redis' throughput over threads seems to be much closer to the predicted "Amdahl speedup" for KeyDB than KeyDB itself. This might be attributed to Mini-Redis using a general purpose Mutex implementation for the core hash table or a possibly more efficient thread utilization in Tokio or a combination of both. Again further investigation is required to draw final conclusions. So although Mini-Redis has a "slower start" than KeyDB[2] Mini-Redis catches up at 8 threads and surpasses KeyDB in throughput at about 12 threads.

When looking at throughput, Redis' approach to multithreading scales particularly poorly in this test. Redis' I/O threading with 12 threads improves the performance compared to the point of reference, which is the single-threaded Redis instance, by about 80% on average (mean). In comparison, a KeyDB server instance with just 2 threads achieves almost the same throughput on average. The results of Redis' I/O threading in this study seem to be about in line with the results of an engineer at "RedisLabs" [56].

Redis Cluster scales almost linearly with 4 nodes (threads) in certain scenarios. For example, a 4-node cluster with 192 configured clients delivers 267% more throughput than one single-threaded Redis instance. The corresponding heatmap in table 5.4 displays this well. When more nodes are added to the cluster, the throughput still improves, but the speedup is far off from the linear mark. Still Redis Cluster delivers the highest throughput and leads with substantial margins among the tested candidates for all configurations with more than 4 threads. This can be observed best by looking at the average (mean) throughput results in the plot in figure 5-1 and the heatmap in figure 5-2.

[1] One possible method that comes to my mind is exchanging KeyDB's "fastlock" with a general purpose "std::mutex" mutex lock implementation.

[2] This is probably because the single-threaded Mini-Redis also delivers less throughput than Redis and KeyDB.

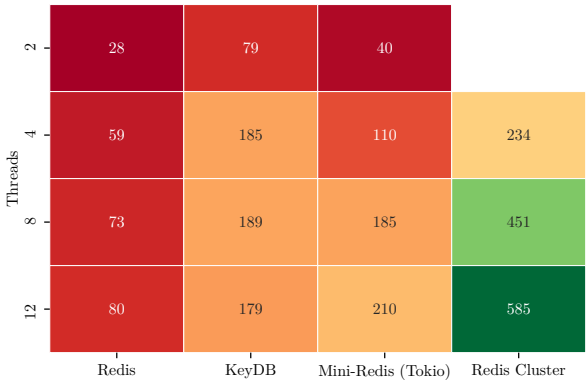


Figure 5-2: Average (mean) throughput compared to reference point (single-threaded Redis) in % (higher is better).

5.1.3 Assumptions in Retrospect

All approaches deliver substantial speedups compared to a single-threaded Redis instance. In most tested scenarios a single-threaded Redis instance (point of reference) does not even get close to fully saturating the capabilities of the system. A Redis Cluster with 12 nodes (threads) is able to increase throughput by up to 667% compared to a single-threaded Redis instance. This proves that single-threaded high concurrency network-bound applications are not inherently hard limited by the network bandwidth.

KeyDB was expected to provide the highest average throughput among the shared-something approaches and up to a thread count of 4 it did. Although KeyDB's developer does not recommend using KeyDB with more than 4 threads, scaling that poorly beyond 4 threads is a surprise.

Mini-Redis' performance in throughput is in-between KeyDB and Redis' I/O threading in most tested scenarios. This corresponds to the assumptions made in chapter 3, too. However, unexpectedly Mini-Redis' average throughput surpassed KeyDB with 8 or more threads. This can mainly be attributed to KeyDB's poor scaling when running on more than 4 threads.

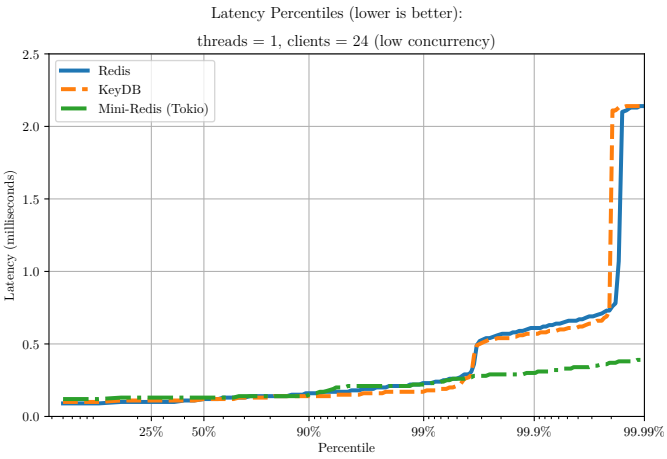
The shared-nothing Redis Cluster, as expected, delivered the best performance in regards to total throughput. In every single test case Redis Cluster provided the highest numbers for throughput. However, for higher node (thread) counts even Redis' Cluster is far off from scaling linearly in the tested benchmarks.

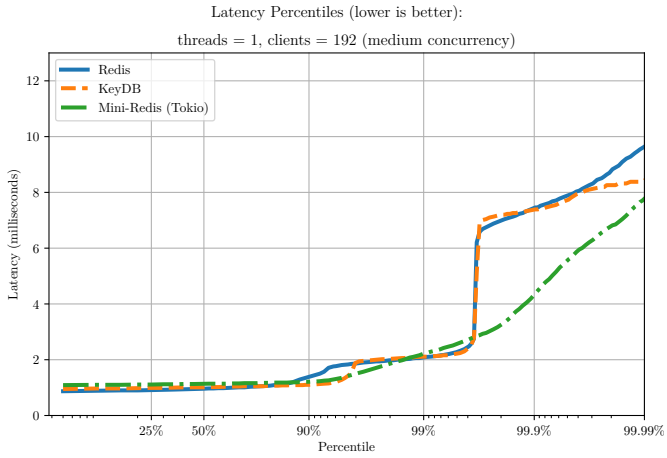
Redis' I/O threading on the other hand delivered the worst throughput results, which was also expected.

5.2 Response Time Percentiles

5.2.1 Plots

Response times: Threads = 1





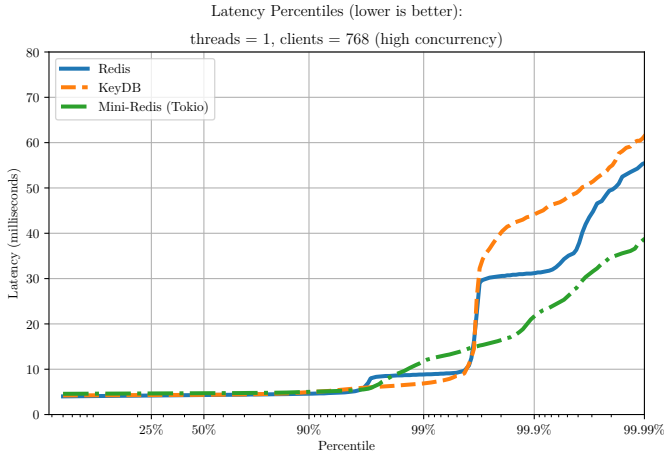


Table 5.5: Response time percentiles with varying client configurations and 1 thread.

Response times: Threads = 2

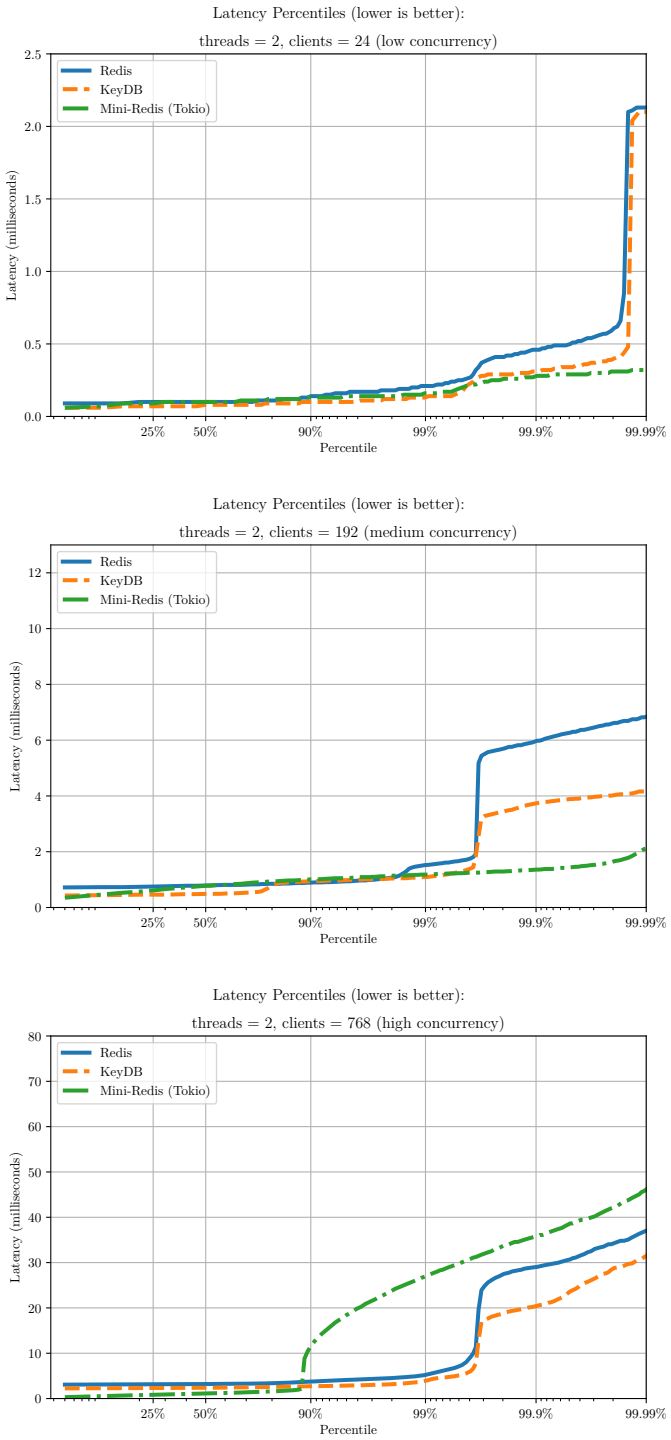
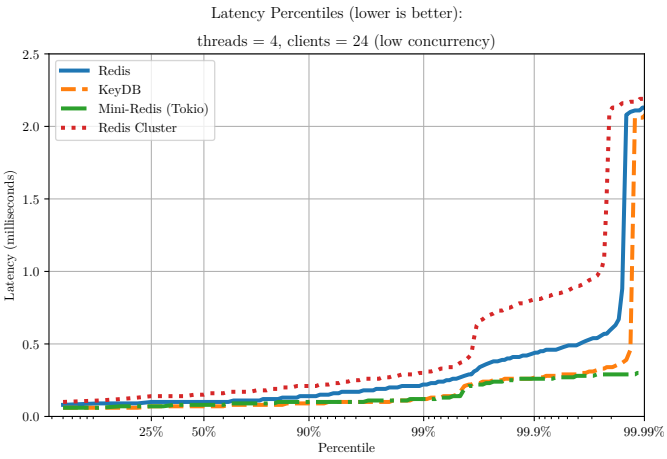
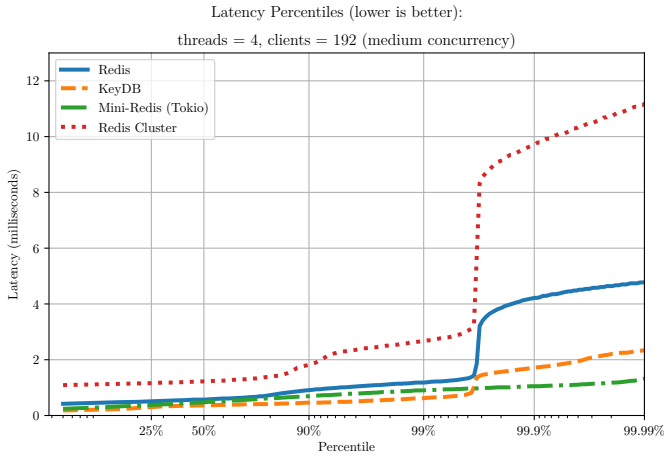


Table 5.6: Response time percentiles with varying client configurations and 2 threads.

Response times: Threads = 4





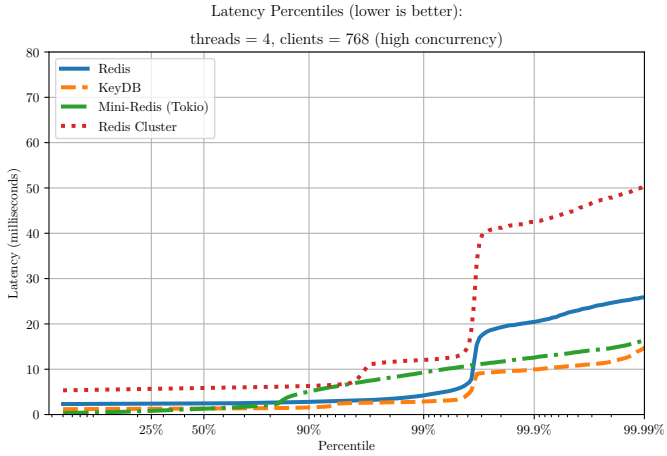
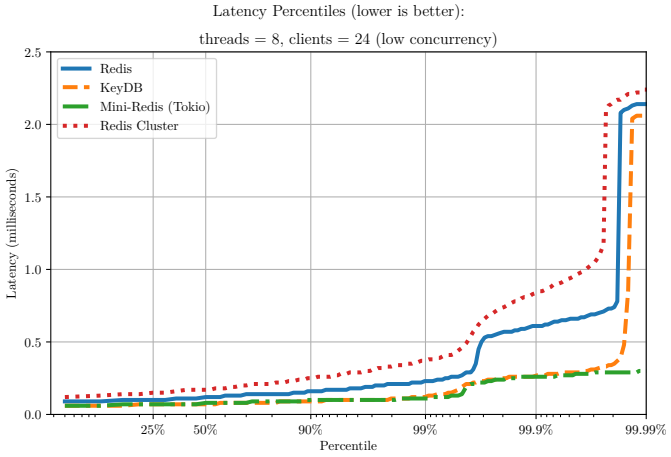
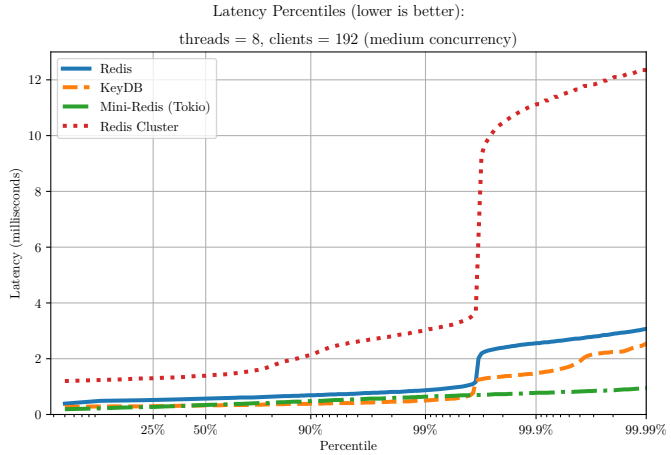


Table 5.7: Response time percentiles with varying client configurations and 4 threads.

Response times: Threads = 8





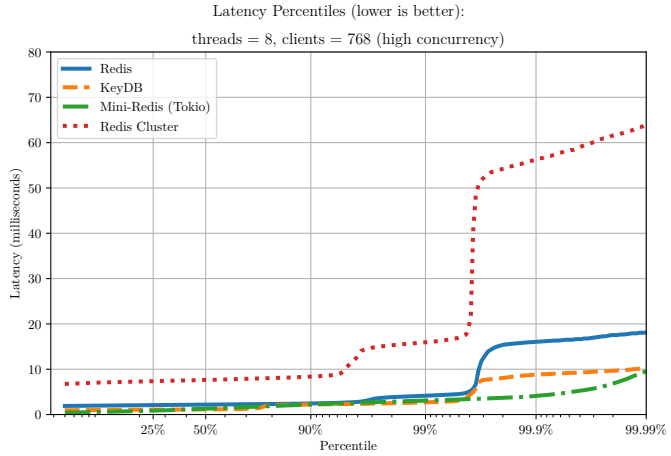
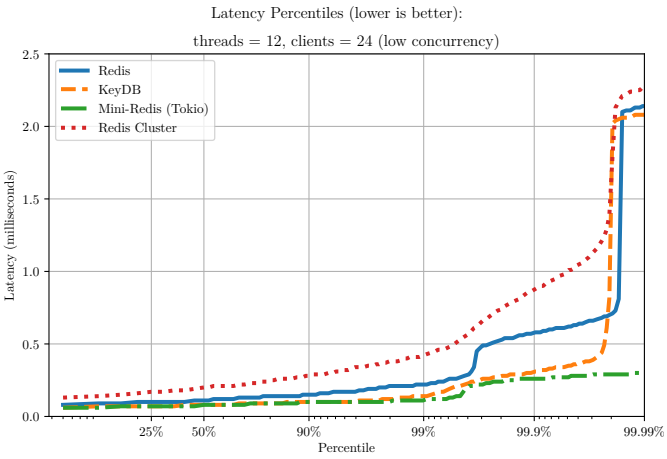
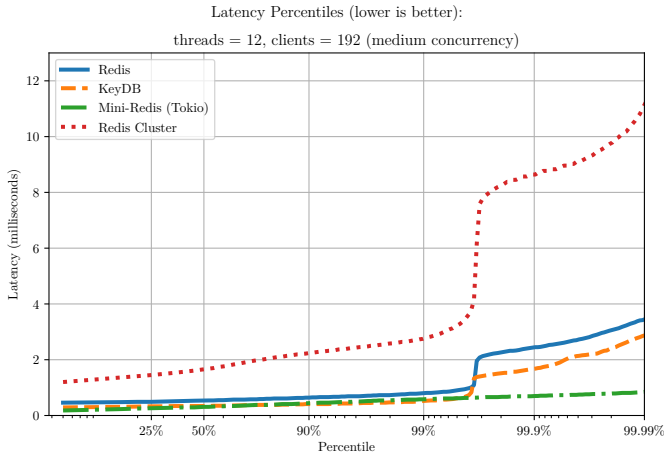


Table 5.8: Response time percentiles with varying client configurations and 8 threads.

Response times: Threads = 12





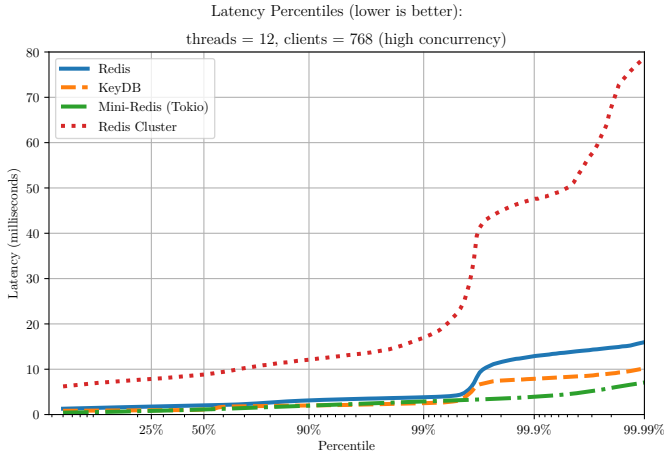


Table 5.9: Response time percentiles with varying client configurations and 12 threads.

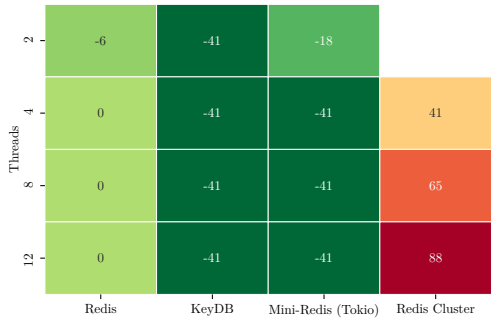
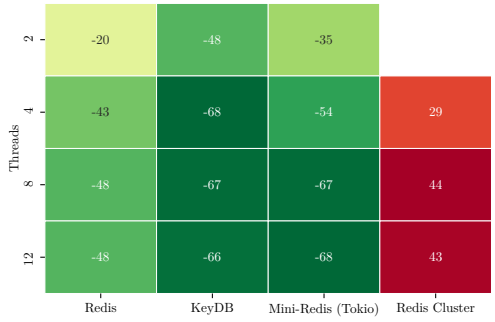
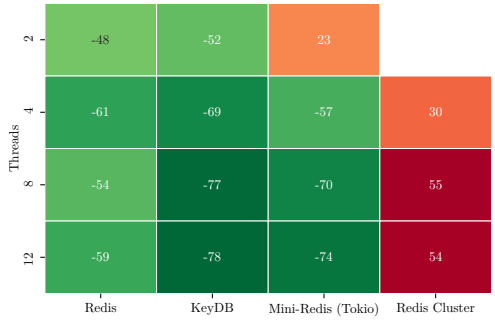
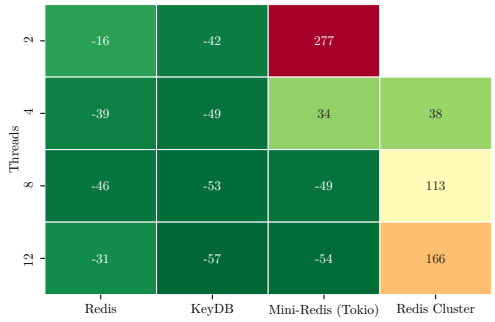
Clients	**95th percentile response time (P95)** compared to reference point in % (single-threaded Redis)
24	
96	
384	
768	

Table 5.10: 95th percentile response time compared to reference point (single-threaded Redis) in % (Lower percentage is better).

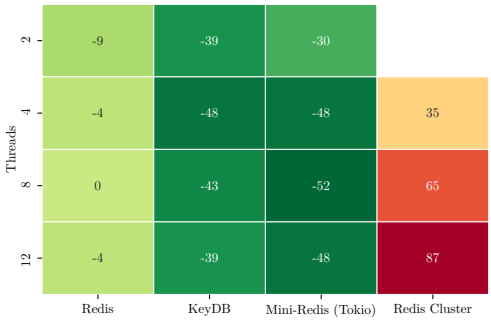
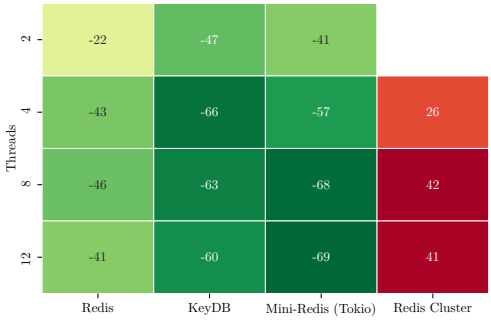
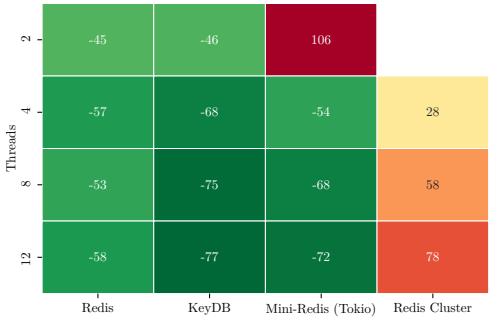
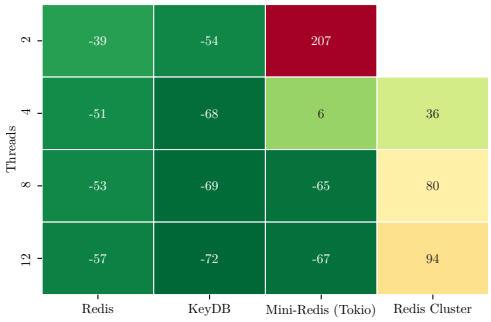
Clients	**99th percentile response time (P99)** compared to reference point in % (single-threaded Redis)
24	
96	
384	
768	

Table 5.11: 99th percentile response time compared to reference point (single-threaded Redis) in % (Lower percentage is better).

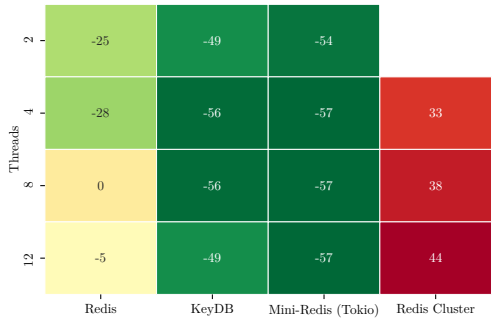
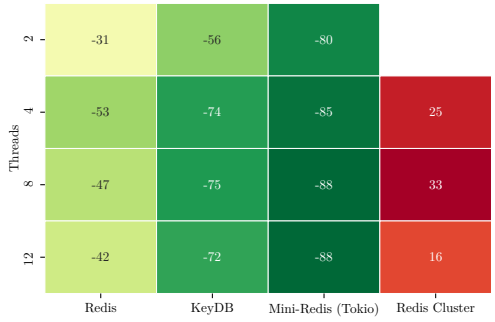
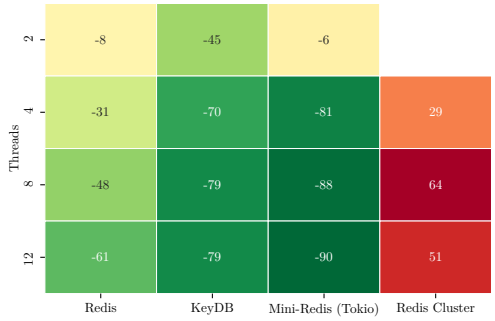
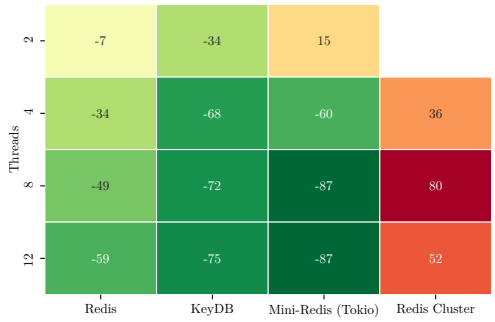
Clients	**99.9th percentile response time (P99.9)** compared to reference point in % (single-threaded Redis)
24	
96	
384	
768	

Table 5.12: 99.9th percentile response time compared to reference point (single-threaded Redis) in % (Lower percentage is better).

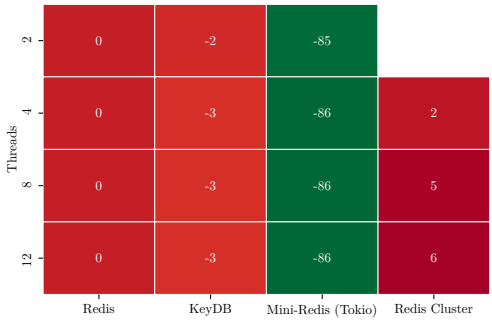
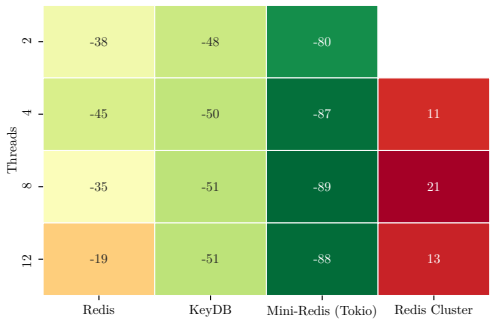
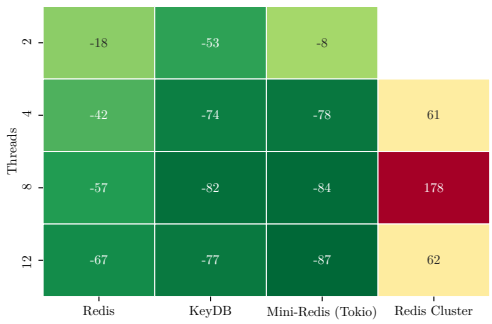
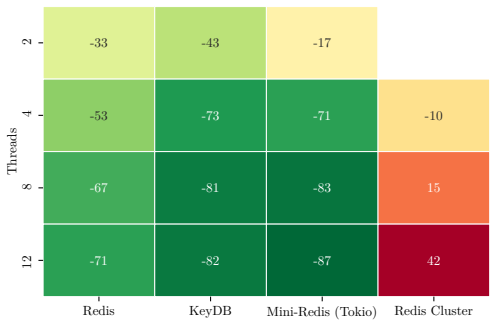
Clients	**99.99th percentile response time (P99.99)** compared to reference point in % (single-threaded Redis)
24	
96	
384	
768	

Table 5.13: 99.99th percentile response time compared to reference point (single-threaded Redis) in % (Lower percentage is better).

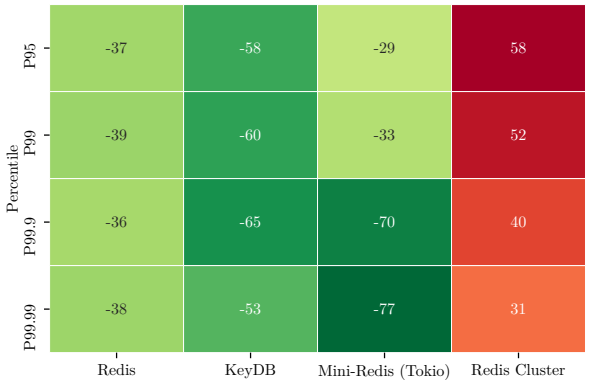


Figure 5-3: Average (mean) response time percentiles compared to reference point (single-threaded Redis) in % (lower is better).

5.2.2 Key Observations

All the shared-something candidates have substantially improved P95, P99, P99.9 and P99.99 response time percentiles on average (mean) compared to the single-threaded Redis server instance (point of reference). The heatmap in figure 5-3, which displays the average (mean) response time percentiles compared to the point of reference, gives a good overview of this. This demonstrates that the improvements that multi-core scaling provides for highly concurrent network-bound applications is not only limited to throughput. The "sweet spot" for response time percentile improvements compared to the point of reference seems to be around 192-384 concurrent clients in this study. All tested applications show higher response time percentiles when increasing the amount of concurrent connections in the benchmark. That is why the HDR histograms use different scales with *low concurrency* (24 clients), *medium concurrency* (192 clients) and *high concurrency* (768) clients in the corresponding figures (see table 5.5, table 5.6, table 5.7, table 5.8 and table 5.9).

On the other hand, increasing the thread count generally improves tail latencies slightly in all applications except for the shared-nothing Redis Cluster, where increasing the amount of nodes leads to deteriorating tail latency results in most cases.

Interestingly, Mini-Redis seems to struggle when there are equal to or more than 192 concurrent connections per thread.[3] This can be observed particularly well in the configuration with 768 clients in table 5.6 or alternatively in the heatmaps of the P95 and P99 percentile in the configurations with 384 and 768 clients in table 5.10 and 5.11. My personal hypothesis

[3] In Tokio's work stealing scheduler implementation, worker threads have not a statically assigned fixed set of connections like in KeyDB.

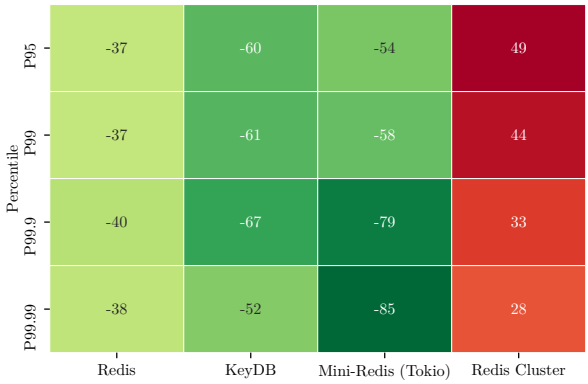


Figure 5-4: Average (mean) response time percentiles without Mini-Redis' outliers compared to reference point in % (lower is better).

is that this might be related to the capacity of the run queue, which stores up to 256 Tasks.[4] However, when there are less than 192 connections per thread, Mini-Redis delivers substantially lower tail latencies than the other applications. In fact, when Mini-Redis' "outliers" are removed it beats the other applications in regards to P99.9 and P99.99 response time percentiles in every single case, sometimes even by a substantial margin. On average (mean), the differences of Mini-Redis' P95 and P99 response time percentiles to the point of reference are almost equivalent to corresponding results of KeyDB, once these "outliers" are removed. The heatmap in figure 5-4 displays the average (mean) response time percentiles without these "outliers" compared to the point of reference.

Redis' I/O threading offers the least improvements in tail latencies among the shared-something applications in most cases, which is analogous to the results in throughput. However, in a lot of cases tail latencies in Redis' I/O threading are not that different from KeyDB's and Mini-Redis' results. Redis' I/O threading even beats Mini-Redis in regards to the average (mean) P95 and P99 percentiles, but this is mainly due to Mini-Redis delivering untypical results when there are more than or equal to 192 connections per thread.

KeyDB's improvements in tail latencies compared to the point of reference are consistently better than the improvements of Redis' I/O threading. KeyDB has lower P95 and P99 percentiles than Mini-Redis (see table 5.10 and 5.11), but Mini-Redis performs better in the P99.9 and P99.99 percentiles (see table 5.12 and 5.13).

Overall Redis Cluster brought up the rear in tail latency performance. This can probably be attributed to an observation made earlier: When benchmarking Redis Cluster, each

[4] If this local run queue is full, tasks are pushed onto a global task queue.

client establishes a TCP connection with each node in the cluster. Increasing the amount of concurrent connections worsens the tail latency results for all applications.

5.2.3 Assumptions in Retrospect

The major assumption was that Tokio's work-stealing scheduling should give Mini-Redis an advantage in regards to tail latency performance. Mini-Redis generally performed well, but in scenarios where the amount of clients per thread was greater or equal to 192, Mini-Redis performed particularly poorly and delivered much worse tail latency results than all the other tested applications.

Furthermore, the tested workloads in the "memtier" benchmark are not particularly skewed, because as explained earlier, the 1% bigger, more expensive queries are unfortunately grouped together. This most likely leads to all worker threads processing demanding queries at about the "same time" and less "stealing" in Mini-Redis' work stealing scheduler. A "truly skewed" workload would probably have been better for showcasing the full benefits that work stealing scheduling can provide. So the fact that Mini-Redis (Tokio) delivered the best results on the P99.9 and P99.99 response time percentiles might rather be attributed to Tokio limiting the task budgets.

What is also interesting is that the shared-nothing database Redis Cluster delivered the worst tail latency results in the conducted tests. This opposes the results from the paper "The Impact of Thread-Per-Core", which was discussed in chapter 3 in the section about "thread-per-core". However, the database described in this paper is also not as "strict shared-nothing" as Redis Cluster.

5.3 Assessment and Further Discussion

The shared-nothing database Redis Cluster offers by far the best improvements in regards to throughput in the benchmark tests. However, a Redis Cluster needs to consist of at least three nodes and its tail latency results, on the other hand, are the worst with a substantial margin. It is the only candidate that almost always delivered even worse tail latency results than a single-threaded Redis instance under full load.

KeyDB lives up to its marketing claims when configured with up to 4 threads and offers substantial improvements in throughput and tail latencies when compared to a single threaded Redis instance.

Redis' I/O threading is the shared-something approach which performs the worst in both throughput and tail latencies.

The poor results of Redis' I/O threading in both performance metrics in this study might be attributed to the overhead associated with Redis' I/O threading. Redis' I/O threading has to be activated and can be stopped, although that is unlikely in a high load scenario. When it is activated, read and write operations are distributed to the consumer threads on each event loop tick and the main thread synchronously waits for the consumers to finish. There is neither a similar overhead in KeyDB nor in Mini-Redis (Tokio):

KeyDB assigns connections statically and each thread handles its own part of connections. The Tokio Tasks in Mini-Redis are only stolen from another thread, when a worker has no other task to process. The overhead in Redis' I/O threading is described in more detail in chapter 3 about architectures & paradigms.

Overall, in low to medium concurrency scenarios, Mini-Redis (Tokio) performs the best in regards to tail latencies. This would most likely be the scenario for something like an in-memory database. Mini-Redis performs well enough in high concurrency situations if it has enough worker threads for handling the connections. The implications of extreme concurrency situations, such as HTTP based server applications with thousands of concurrent connections, on Tokio's performance will have to be investigated more thoroughly.

Furthermore, Redis and KeyDB both support command pipelining. However, all these tests were performed with command pipelining disabled. Command pipelining is supposed to trade latency for throughput. Whether this command pipelining could improve Redis' or KeyDB's operations per second to a level similar to Redis Cluster, while aiming for similar latency results on a single (physical) node, would be an interesting starting point for further research.

Chapter 6

Conclusions

6.1 Overview

A defining quote for this thesis could be: *"The free lunch is over."* Making this rather humorous statement, in 2005, 16 years ago, Herb Sutter wisley predicted that applications would increasingly need to incorporate parallelism in order to substantially benefit from modern hardware [73].

This thesis demonstrates that in certain scenarios the question is not *whether* to incorporate parallelism into highly concurrent network-bound applications for scaling on multi-core CPUs, but rather *how* to efficiently benefit from parallelism in these applications. This study examines *how* different highly concurrent network-bound applications and library abstractions implement concurrency & parallelism. It captures and classifies conceptual differences between the architectures and paradigms, which these applications make use of, without losing sight of the big picture, while also preserving relevant nuances. All things considered, the conducted research and the assumptions, which were formulated based on inductive reasoning and logic, are supported by the quantitative performance evaluations. These evaluations demonstrate particularly well that "there is no free lunch" when implementing such systems. There is a trade-off with almost every detail, so decisions about implementation details should be made according to the requirements of the application.

As a conclusion I would like to summarize some of the key findings of the research conducted for this thesis and shed light on possible further developments in this field.

6.2 Principles when developing Highly Concurrent Network-Bound Applications

Developing usually involves dealing with requirements which correspond to a problem that an application is supposed to solve. It is important to define clear requirements and prioritize them. Since "there is no free lunch", these requirements should be the major influence on architectural decisions. In highly concurrent network-bound applications, such requirements could include concrete numbers in throughput, response time percentiles and further scalability in a distributed environment. In a serverless architecture requirements could even include something like application startup time.

Knowledge about the data that is served and how it is accessed is crucial. It is important to know how computational effort and payload sizes are distributed among requests. If most of the payloads are small, bypassing the kernel altogether with a user space network stack should be considered. Large payloads often are files and optimizations such as the sendfile system call can be applied to reduce copies. The network bandwidth or the speed of secondary storage often bottleneck these larger payloads. Demanding computations or other "blocking" operations within requests should not interfere with making progress in an event driven architecture and can be offloaded to a threadpool, for example. "Head-of-line blocking" caused by demanding requests in skewed workloads can be reduced by techniques such as Tokios's task budget. This is supported by Tokio delivering the best P99.9 and P99.99 response time percentiles in the benchmarks for this study.

Sometimes useful high level information about the data access is available. This information should even be considered for implementing "lower level" details. For example, in large distributed fan-out architectures some requests might have particularly strict response time requirements. This is a scenario where optimizations utilizing techniques similar to the "latency ring" in Glommio could become interesting.

Essentially, the major differences in respect to parallelism in network-driven architectures are the queueing models and related work distribution and scheduling algorithms. One practical example for a trade-off regarding the queueing model is whether to use separate socket instances with "SO_REUSEPORT" or not. It should also be assessed, whether or not to share the stateful I/O selector instance across multiple threads of execution. Different approaches have different implications on performance, which usually depend on the concrete workload. Similarly, different strategies of distributing work among multiple threads can result in better

or worse performance depending on the workload. Rather "fair" scheduling algorithms, such as the discussed "work-stealing", tend to share more resources and have more algorithmical overhead. However, they can improve response times and throughput among clients under certain non-uniform load scenarios.

Standard benchmarks, such as the "memtier" benchmark that is used in this study, often test with uniform load distribution among clients, where each simulated client issues as many requests as the server (or the client) can handle, which is the best case scenario for most applications. When data is partitioned, benchmarks usually also generate uniform load on nodes, which means that they do not test highly skewed workloads.

But probably the most important consideration is the question whether to leverage the shared-something or the shared-nothing approach. Sharing resources usually limits the theoretical speedup, especially if process synchronization is involved that decreases p in Amdahl's law. The Redis' I/O threading that was put to the test in the benchmarks is a good example for an application that scales rather poorly across multiple CPU cores, which is most likely due to a "low" p and further overhead associated with its implementation. On the other hand, a "high" p can lead to almost linear scaling across multiple cores. This is why shared-nothing should be considered on a single CPU with multiple cores, too. In this study the shared-nothing database Redis Cluster was the only application that achieved almost linear scaling in regards to throughput in some cases. However, shared-nothing approaches and the associated data partitioning come at the cost of either making "non-delightful" transactions substantially more complex or not supporting them at all. Shared-nothing systems are also prone to experiencing hot spot problems. The shared-nothing database Redis Cluster also delivered substantially worse tail latency results than all the other shared-something approaches in this study.

Other important aspects are implementing the finest possible lock granularity or lock-free data structures that remove the need for locking altogether. When locking is needed, different locking algorithms should be considered. The best way is to test and verify multiple approaches by applying the relevant metrics. Furthermore, the client might be able to reduce process synchronization on the server by keeping state local. One example of this is the cluster state on Redis Cluster, which is (also) stored on the client.

Thread-level parallelism is not the only paradigm that enables parallelism on modern CPUs. SIMD instructions can massively improve certain operations on all array-like data structures. A use case where server applications can substantially benefit from SIMD is the parsing of

data exchange formats.

Highly concurrent network-bound applications should minimize the amount of system calls, which are performed for serving requests. On GNU/Linux systems, “io_uring” seems to redefine the I/O related interfaces of the operating system and offers lots of potential performance improvements including reducing the amount of total system calls. Everything in-between the application code and the hardware can bottleneck. As already mentioned, the kernel probably bottlenecks the overall throughput if the network payloads are “small”. However, some performance aspects of the kernel can be improved by simply reconfiguring it. For example, certain threads can be dedicated to receiving interrupts by setting up the IRQ affinity. Furthermore, software layers that are added in-between the application code and the hardware could also be potential bottlenecks. The Node.js runtime, for example, is prone to bottlenecking the throughput.

Performance benchmarks should be built into the CI pipeline of these applications and should specifically put the defined requirements to the test. Test cases should be as close to real use cases as possible. Numbers can fool, so it is important to know the error of the results and to test a variety of different software/hardware configurations.

6.3 Possible further developments in this field

One obvious trend for highly concurrent network-bound applications is making use of the shared-nothing paradigm on a single node, which is what the “thread-per-core” model embraces. In the past, many developers considered shared-nothing to be an architectural paradigm inherently linked to distributed computing. However, the growing core counts of state of the art CPUs increase the need for applications that scale (almost) linearly and thus benefit from (all) the cores that the CPU provides.

The creation of light weight “async runtimes” to support the development of high-performance server applications such as Tokio, Glommio or Seastar is another emerging trend. Glommio and Seastar even embrace the shared-nothing paradigm.

“io_uring” will most likely revolutionize I/O on Linux. However, this adoption might take some time because “io_uring” ’s “truly asynchronous” interface differs vastly from what Linux and other UNIX-like operating systems have provided in the past. Highly concurrent network-bound applications usually implement the code that handles the I/O as “zero-cost abstractions” on top of Linux APIs and other supported platforms are often “second class

citizens", which is not a substantial problem if the APIs are similar. In order not to miss out on too much performance, other UNIX-like operating systems might be forced to adopt "io_uring", too. Glommio, for example, is Linux exclusive and already built on "io_uring". Tokio plans to adopt "io_uring" this year[1].

The boundaries between what is typically done in user space and what is done in kernel space have started to blur: User space network stacks shift functionality that was previously handled by the kernel to user space to remove possible overhead within the kernel. eBPF programs, on the other hand, enable running custom application logic within the kernel. In the coming years, these boundaries will probably be blurred even further.

In regards to hardware, NIC's will certainly become even faster. Currently, new ARM CPUs, RISC-V CPUs and further hardware acceleration are turning the previously Intel dominated server industry upside down. For example, today AWS offers competitive EC2 instances with custom ARM CPUs (AWS Graviton) and the AWS Nitro System, which is a family of ASICs that accelerate cloud computing environments. I assume that the future might bring server hardware that has an even tighter integration with the NIC and other custom ASICs for hardware acceleration. FPGAs are already popular in software defined networking. Considering the large scale at which cloud providers act, highly concurrent system applications, such as in-memory databases or high performance web servers, might be impacted by the trend of hardware acceleration, too.

[1] 2021

References

[1] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *SOSP 2007*. Stevenson, WA, USA: ACM, 2007. URL: https://www.allthingsdistributed/files/amazon-dynamo-sosp2007.pdf.

[2] M. Kagan et al. “MMX TM Microarchitecture of Pentium ® Processors With MMX Technology and Pentium ® II Microprocessors”. In: Intel Technology Journal 3 (1997). URL: https://www.intel.com/content/dam/www/public/us/en/documents/research/1997-vol01-iss-3-intel-technology-journal.pdf.

[3] S. Vijay Prasad et al. “A Comparative Study Between Multi Queue Multi Server And Single Queue Multi Server Queuing System”. In: *International Journal of Scientific & Technology Research* 8.12 (2019). ISSN: 2277-8616. URL: http://www.ijstr.org/final-print/dec2019/A-Comparative-Study-Between-Multi-Queue-Multi-Server-And-Single-Queue-Multi-Server-Queuing-System.pdf.

[4] Xiang Ren et al. “An Analysis of Performance Evolution of Linux’s Core Operations”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 554–569. ISBN: 978-1-4503-6873-5/19/10. DOI: 10.1145/3341301.3359640. URL: https://doi.org/10.1145/3341301.3359640.

[5] Martin André. *Rust vs Go - Load testing webserv (>400k req/s)*. [Online; accessed 16-Febuary-2021]. Nov. 2020. URL: https://dev.to/martichou/rust-vs-go-load-testing-400k-req-s-53l.

[6] ARM®, ed. *Programmer’s Guide for ARMv8-A*. [Online; accessed 27-January-2021]. ARM®, 2015. Chap. 6.3 Memory access instructions, 14.1 Multi-processing systems. URL: https://documentation-service.arm.com/static/5fbd26f271eff94ef49c7020.

[7] Eli Bendersky. *Measuring context switching and memory overheads for Linux threads*. [Online; accessed 02-February-2021]. 2018. URL: https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/.

[8] Henry C.Baker Jr. and Carl Hewitt. “The Incremental Garbage Collection of Processes”. In: *AI PL Conference*. AI Working Paper 149. Rochester, N.Y., USA: Massachusetts Institute of Technology Artificial Intelligence Laboratory, Aug. 1977. URL: https://core.ac.uk/download/pdf/4406556.pdf.

[9] Alibaba Cloud, ed. *Improving Redis Performance through Multi-Thread Processing*. [Online; accessed 03-December-2020]. 2018. URL: https://www.alibabacloud.com/blog/improving-redis-performance-through-multi-thread-processing_594150.

[10] Abseil Contributors, ed. *Swiss Tables Design Notes*. [Online; accessed 03-February-2021]. URL: https://abseil.io/about/design/swisstables#swiss-tables-design-notes.

[11] ACTIX Contributors, ed. *actix*. Version 0.10.0. URL: https://github.com/actix/actix.

[12] Future Crate Contributors, ed. *Crate futures*. [Online; accessed 19-January-2021]. 2020. URL: https://docs.rs/futures/0.3.12/futures/.

[13] GCC Contributors, ed. *Auto-vectorization in GCC*. [Online; accessed 09-February-2021]. 2011. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[14] Linux Contributors, ed. *Efficient IO with io_uring*. [Online; accessed 27-December-2020]. 2019. URL: https://kernel.dk/io_uring.pdf.

[15] Linux Contributors, ed. *epoll(7) — Linux manual page*. [Online; accessed 27-December-2020]. 2019. URL: https://man7.org/linux/man-pages/man7/epoll.7.html.

[16] Linux Contributors, ed. *futex(2) — Linux manual page*. [Online; accessed 16-Febuary-2021]. 2020. URL: https://man7.org/linux/man-pages/man2/futex.2.html.

[17] Linux Contributors, ed. *io_submit(2) — Linux manual page*. [Online; accessed 27-December-2020]. 2020. URL: https://man7.org/linux/man-pages/man2/io_submit.2.html.

[18] Linux Contributors, ed. *MSG_ZEROCOPY*. [Online; accessed 27-December-2020]. 2018. URL: https://github.com/torvalds/linux/blob/master/Documentation/networking/msg_zerocopy.rst.

[19] Linux Contributors. *Networking*. [Online; accessed 27-December-2020]. 2020. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html.

[20] Linux Contributors, ed. *poll(2) — Linux manual page*. [Online; accessed 27-December-2020]. 2020. URL: https://man7.org/linux/man-pages/man2/poll.2.html.

[21] Linux Contributors, ed. *select(2) — Linux manual page*. [Online; accessed 27-December-2020]. 2020. URL: https://man7.org/linux/man-pages/man2/select.2.html.

[22] Linux Contributors, ed. *sendfile(2) — Linux manual page*. [Online; accessed 27-December-2020]. 2017. URL: https://man7.org/linux/man-pages/man2/sendfile.2.html.

[23] Linux Contributors, ed. *signalfd(7) — Linux manual page*. [Online; accessed 7-January-2021]. 2020. URL: https://man7.org/linux/man-pages/man2/signalfd.2.html.

[24] Linux Contributors, ed. *socket(7) — Linux manual page*. [Online; accessed 27-December-2020]. 2020. URL: https://man7.org/linux/man-pages/man7/socket.7.html.

[25] LLVM Contributors, ed. *Auto-Vectorization in LLVM*. [Online; accessed 09-February-2021]. 2021. URL: https://llvm.org/docs/Vectorizers.html.

[26] Rust-Lang Contributors, ed. *Asynchronous Programming in Rust*. [Online; accessed 27-January-2021]. 2019. URL: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html.

[27] Rust-Lang Contributors, ed. *Trait std::future::Future*. [Online; accessed 19-January-2021]. 2020. URL: https://docs.rs/rustc-std-workspace-std/1.0.1/std/future/trait.Future.html.

[28] Wikipedia Contributors, ed. *Futures and promises*. [Online; accessed 02-January-2021]. URL: https://en.wikipedia.org/wiki/Futures_and_promises.

[29] Wikipedia Contributors, ed. *Shared-nothing architecture*. [Online; accessed 27-December-2020]. URL: https://en.wikipedia.org/wiki/Shared-nothing_architecture.

[30] Jonathan Corbet and Alessandro Rubini. *Linux Device Drivers, Second Edition*. O'Reilly, 2001, pp. 425–469. ISBN: 9780596000080.

[31] Glauber Costa. [Online; accessed 01-February-2021]. 2020. URL: https://github.com/DataDog/glommio/issues/237.

[32] Loris Cro. *You Don't Need Transaction Rollbacks in Redis*. [Online; accessed 30-November-2020]. 2020. URL: https://redislabs.com/blog/you-dont-need-transaction-rollbacks-in-redis/.

[33] Ryan Dahl. *deno*. Version 1.7.2. URL: https://github.com/denoland/deno.

[34] DataDog, ed. *glommio*. Version 0.2.0-alpha. URL: https://github.com/DataDog/glommio.

[35] Ulrich Drepper. *Futexes Are Tricky*. [Online; accessed 18-February-2021]. July 2004. URL: https://dept-info.labri.fr/~denis/Enseignement/2008-IR/Articles/01-futex.pdf.

[36] Cristian F. Dumitrescu. *Design Patterns for Packet Processing Applications on Multi-core Intel® Architecture Processors*. [Online; accessed 27-December-2020]. 2008. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-multicore-packet-processing-paper.pdf.

[37] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. "The Impact of Thread-Per-Core Architecture on Application Tail Latency". English. In: *Architectures for Networking and Communications Systems (ANCS),2019 ACM/IEEE Symposium on*. 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS ; Conference date: 24-09-2019 Through 25-09-2019. United States: IEEE, 2019, pp. 1–8. ISBN: 978-1-7281-4388-0. DOI: 10.1109/ANCS.2019.8901874.

[38] Matt Fleming. "A thorough introduction to eBPF". In: (2017). [Online; accessed 18-January-2021]. URL: https://lwn.net/Articles/740157/.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2012, pp. 262–287. ISBN: 9780123838728.

[40] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008, pp. 13–14, 45–48, 63, 223–238, 369–391, 473–476. ISBN: 9780124159501.

[41] Intel®, ed. *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture*. [Online; accessed 09-February-2021]. 2012. URL: https://software.intel.com/content/www/us/en/develop/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html.

[42] Intel®, ed. *Intel® 64 Architecture Memory Ordering White Paper*. [Online; accessed 10-January-2021]. 2007. URL: http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf.

[43] Intel®, ed. *Intel® Advanced Vector Extensions Programming Reference*. [Online; accessed 09-February-2021]. Intel® Corporation, 2011. URL: https://software.intel.com/sites/default/files/4f/5b/36945.

[44] Intel®, ed. *PCIe* GbE Controllers Open Source Software Developer's Manual*. [Online; accessed 27-December-2020]. Intel® Corporation, 2012. Chap. 2.3 Microarchitecture. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/pcie-gbe-controllers-open-source-manual.pdf.

[45] Dan Kegel. *The C10K problem*. [Online; accessed 19-December-2020]. 1999. URL: http://www.kegel.com/c10k.html.

[46] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010, pp. 1245, 1260–1262, 1370–1372. ISBN: 1593272200.

[47] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017, pp. 13–16, 205. ISBN: 9781449373320.

[48] Matt Kulukundis. "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step". [Online; accessed 30-November-2020]. 2017. URL: https://www.youtube.com/watch?v=ncHmEUmJZf4.

[49] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (Oct. 2019), pp. 941–960. ISSN: 0949-877X. DOI: 10.1007/s00778-019-00578-5. URL: http://dx.doi.org/10.1007/s00778-019-00578-5.

[50] Carl Lerche. *Making the Tokio scheduler 10x faster*. [Online; accessed 03-January-2021]. 2019. URL: https://tokio.rs/blog/2019-10-scheduler.

[51] Carl Lerche. *Reducing tail latencies with automatic cooperative task yielding*. [Online; accessed 03-January-2021]. 2020. URL: https://tokio.rs/blog/2020-04-preemption.

[52] Marek Majkowski. *io_submit: The epoll alternative you've never heard about*. [Online; accessed 04-January-2021]. 2019. URL: https://blog.cloudflare.com/io_submit-the-epoll-alternative-youve-never-heard-about/.

[53] Marek Majkowski. *Why does one NGINX worker take all the load?* [Online; accessed 30-January-2021]. 2017. URL: https://blog.cloudflare.com/the-sad-state-of-linux-socket-balancing/.

[54] Tyler Mandry. *How Rust optimizes async/await I + II*. [Online; accessed 30-January-2021]. 2019. URL: https://tmandry.gitlab.io/blog/posts/.

[55] Inc. Nginx, ed. *NGINX*. Version 1.19.6. URL: http://hg.nginx.org/nginx/.

[56] Filipe Oliveira. *Benchmarking the experimental Redis Multi-Threaded I/O*. [Online; accessed 13-Febuary-2021]. 2019. URL: https://itnext.io/benchmarking-the-experimental-redis-multi-threaded-i-o-1bb28b69a314.

[57] Orestis Polychroniou and Kenneth A. Ross. "Vectorized Bloom Filters for Advanced SIMD Processors". In: *DaMoN'14 (10th International Workshop on Data Management on New Hardware (DaMoN 2014), June 22-27 2014*. Snowbird, UT, USA: Association for Computing Machinery, 2014. URL: http://www.cs.columbia.edu/~orestis/damon14.pdf.

[58] OpenFastPath Project, ed. *OpenFastPath*. Version 1. URL: https://github.com/OpenFastPath/ofp.

[59] Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt. "Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events". In: *4th annual Pattern Languages of Programming conference*. Allerton Park, Illinois: Department of Computer Science, Washington University, St. Louis, MO, 1997. URL: https://www.dre.vanderbilt.edu/~schmidt/PDF/proactor.pdf.

[60] RedisLabs, ed. *memtier_benchmark*. Version 1.3.0. URL: https://github.com/RedisLabs/memtier_benchmark.

[61] RedisLabs, ed. *Redis*. Version 6.0.9. URL: https://github.com/redis/redis.

[62] RedisLabs, ed. *Redis Cluster Specification*. [Online; accessed 29-November-2020]. 2018. URL: https://redis.io/topics/cluster-spec.

[63] cpp reference, ed. *std::memory_order*. [Online; accessed 13-January-2021]. 2020. URL: https://en.cppreference.com/w/cpp/atomic/memory_order.

[64] Alice Ryhl. *Async: What is blocking?* [Online; accessed 20-January-2021]. 2020. URL: https://ryhl.io/blog/async-what-is-blocking/.

[65] Salvatore Sanfilippo. [Online; accessed 05-January-2021]. 2019. URL: https://twitter.com/antirez/status/1111165726340075520.

[66] Salvatore Sanfilippo. [Online; accessed 05-February-2021]. 2019. URL: https://twitter.com/antirez/status/1110973405539565569.

[67] Douglas C. Schmidt. *Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. [Online; accessed 21-December-2020]. 1995. URL: https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf.

[68] Amazon Web Services, ed. *Amazon ElastiCache for Redis 5.0.3 enhances I/O handling to boost performance*. [Online; accessed 01-December-2020]. 2019. URL: https://aws.amazon.com/about-aws/whats-new/2019/03/amazon-elasticache-for-redis-503-enhances-io-handling-to-boost-performance/.

[69] Dariusz Sosnowski. *How can DPDK access devices from user space?* [Online; accessed 27-December-2020]. 2019. URL: https://codilime.com/how-can-dpdk-access-devices-from-user-space/.

[70] Michael Stonebraker. *The Case for Shared Nothing*. [Online; accessed 27-December-2020]. 1986. URL: https://dsf.berkeley.edu/papers/hpts85-nothing.pdf.

[71] Bjarne Stroustrup. *Are lists evil?* [Online; accessed 31-January-2021]. URL: https://www.stroustrup.com/bs_faq.html#list.

[72] John Sully. *KeyDB*. Version 6.0.16. URL: https://github.com/JohnSully/KeyDB.

[73] Herb Sutter. "The Free Lunch Is Over". In: (2005). [Online; accessed 20-December-2020]. URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[74] libuv team, ed. *libuv*. Version 1.40.0. URL: https://github.com/libuv/libuv.

[75] tokio-rs, ed. *Mini-Redis*. Version Tokio 1.2. URL: https://github.com/tokio-rs/mini-redis.

[76] tokio-rs, ed. *Module tokio::task*. [Online; accessed 02-February-2021]. 2020. URL: https://docs.rs/tokio/1.1.0/tokio/task/index.html.

[77] Linus Torvalds. "No nuances, just buggy code (was: related to Spinlock implementation and the Linux Scheduler)". [Online; accessed 30-January-2021]. 2020. URL: https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723.

[78] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. [Online; accessed 05-January-2021]. 2020. URL: https://support.google.com/faqs/answer/7625886.

[79] Tony Van Eerd. "An Interesting Lock-free Queue - Part 2 of N". [Online; accessed 29-November-2020]. 2017. URL: https://www.youtube.com/watch?v=HP2InVqgBFM.

[80] Hielke de Vries. *io_uring bare minimum echo server*. [Online; accessed 05-January-2021]. 2020. URL: https://github.com/frevib/io_uring-echo-server.

[81] *Web Framework Benchmarks - Round 20*. [Online; accessed 16-Febuary-2021]. 2021. URL: https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=fortune.

Appendix A

Concurrent Queues

A.1 SPSC Queue

Listing A.1: Herlihy's "WaitFreeQueue<T>" in C++

```
#ifndef __SPSC_RING_H__
#define __SPSC_RING_H__

/**
    * Basically Herlihy's WaitFreeQueue<T>
    * in modern CPP with memory barriers
    * Author: Lucas Craemer
    * */

#include <array>
#include <atomic>
#include <cstddef>
#include <cstdint>
#include <optional>

template <typename T, std::size_t N> class SPSCRing {
private:
    std::atomic<std::size_t> m_head;
    std::array<T, N> m_buff;
    std::atomic<std::size_t> m_tail;

public:
```

```
// initialize head and tail
constexpr SPSCRing() noexcept : m_head(0), m_tail(0) {}

// return ring size
constexpr std::size_t size() noexcept { return N; }

bool enqueue(T val) noexcept {
    std::size_t current_tail = m_tail.load(std::memory_order_relaxed);
    if (current_tail - m_head.load(std::memory_order_relaxed) < N) {
        // The queue is not full.
        // Insert the element at the current tail index.
        m_buff[current_tail % N] = std::move(val);
        // (possible optimization to get rid off the modulus, N = 2^x, index =
        // current_tail & (N - 1)). Increment the tail. Do not reorder any
        // any read or writes after the increment. (Before means before!)
        // 'Release' the previous writes.
        // <-------------------------------------------------------------------->
        // compiler and memory barrier
        m_tail.store(++current_tail, std::memory_order_release);
        // Success
        return true;
    }
    // The queue is full. Return false.
    return false;
}

std::optional<T> dequeue() noexcept {
    // Load the tail.
    // Do not reorder any read or writes before the load. (After means after!)
    // Acquire barrier makes writes before the release store of the atomic
    // variable visible.
    const std::size_t current_tail = m_tail.load(std::memory_order_acquire);
    // <-------------------------------------------------------------------->
    // compiler and memory barrier
    std::size_t current_head = m_head.load(std::memory_order_relaxed);
    // Is the buffer not empty?
    if (current_head != current_tail) {
```

```
            auto value = std::move(m_buff[current_head % N]);
            // Update the head.
            m_head.store(++current_head, std::memory_order_relaxed);
            return std::optional(std::move(value));
        }
        // Else return NULL.
        return std::nullopt;
    }
};

#endif
```

A.2 MPMC Queue

Listing A.2: Van Eerd's "An Interesting Lock-free Queue"

```
#ifndef __MPMC_RING_H__
#define __MPMC_RING_H__

/**
    * Inspired by 'An Interesting Lock-free Queue'
    * Talk by Tony Van Eerd at the CppCon 2017:
    * https://www.youtube.com/watch?v=HP2InVqgBFM
    * Author: Lucas Craemer
    * */

#include <algorithm>
#include <array>
#include <atomic>
#include <cstddef>
#include <cstdint>
#include <optional>

// This is the entry structure for the buffer.
// It holds a pointer to the actual value and a generation for that pointer.
template <typename T> struct PtrGen {
    std::optional<T *> ptr;
    std::uint64_t gen;

    inline bool is_empty(std::uint64_t cmp_gen) noexcept {
        // The entry is empty if the ptr points to NULL and it is in the current
        // gen.
        return !ptr.has_value() && gen == cmp_gen;
    }

    inline bool is_valid(std::uint64_t cmp_gen) noexcept {
        // The entry is valid if the ptr does not point to NULL and it is in the
        // current gen.
        return ptr.has_value() && gen == cmp_gen;
    }
```

```
};

// This is the data structure for the head and tail of the queue.
// It holds an index for head/tail into the buffer (this index is likely close
// the real index) and the the generation of that index to mitigate the ABA
// problem.
template <std::size_t N> struct IdxGen {
    std::size_t idx;
    std::uint64_t gen;

    inline void incr() noexcept {
        // Increment the index, if the index reaches the buffer size start at 0
        // again and increment the generation.
        if (++idx == N) {
            idx = 0;
            ++gen;
        }
    }

    inline bool operator<(const IdxGen<N> &rhs) noexcept {
        return gen < rhs.gen || (gen == rhs.gen && idx < rhs.idx);
    }
};

// Multiple-producer multiple-consumer circular FIFO queue
// Stores pointer to a type T.
// N is the cacapcity of the queue.
template <typename T, std::size_t N> class MPMCRing {
private:
    std::atomic<IdxGen<N>> m_headish;
    // The buff in between head and tail, so that depending on the buff size they
    // will likely end up on a different cache line. (no contention on the cache
    // line)
    std::array<std::atomic<PtrGen<T>>, N> m_buff;
    std::atomic<IdxGen<N>> m_tailish;

public:
```

```
constexpr MPMCRing() noexcept
    : m_headish(IdxGen<N>{0, 0}), m_tailish(IdxGen<N>{0, 0}) {
    // Start by zeroing the buffer.
    std::fill(std::begin(m_buff), std::end(m_buff), PtrGen<T>{std::nullopt, 0});
}

// return ring size
constexpr std::size_t size() noexcept { return N; }

// Enqueue a pointer to a type T.
// Returns true on success.
// Returns false if the queue is full and enqueueing was not successful.
bool enqueue(T *val) noexcept {
    std::uint64_t prev_gen = 0;
    IdxGen<N> old_tailish;
    // Load the 'current tail'. This tail is just a hint
    // and does not have to be the 'real' tail.
    // No memory barrier required due to acquire-release CAS (I think).
    IdxGen<N> new_tailish = old_tailish =
        m_tailish.load(std::memory_order_relaxed);
    PtrGen<T> current_entry;
    do {
        // Iterate over the buffer while the entry is not empty.
        // No memory fencing required.
        while (!(current_entry =
                    m_buff[new_tailish.idx].load(std::memory_order_relaxed))
                    .is_empty(new_tailish.gen)) {
            // The entry is not empty because the pointer is either not NULL
            // or not in the same generation as the current tail.
            if (current_entry.gen < prev_gen) {
                // The buffer is full, because the entry is not empty and the
                // generation of the current entry has decreased.
                // Update the tail if its smaller and return.
                if (old_tailish < new_tailish) {
                    // Update the tail if its smaller and return.
                    // Strong CAS, because not in a loop.
                    // No writes, no memory barrier required.
```

```
                    m_tailish.compare_exchange_strong(old_tailish, new_tailish,
                                                      std::memory_order_relaxed,
                                                      std::memory_order_relaxed);
                }
                return false;
            }
            // Increment the tail for further iteration.
            new_tailish.incr();
            // Replace the previous generation with the generation of the
            // current entry if the pointer points to a valid address.
            if (current_entry.ptr.has_value())
                prev_gen = current_entry.gen;
        }
        // Finally: Try adding the entry to the buffer. If another thread
        // changed the entry retry. Weak CAS, because loop.
        // Do not reorder any read or writes before/after the successful CAS.
    } while (!m_buff[new_tailish.idx].compare_exchange_weak(
        current_entry, PtrGen<T>{val, new_tailish.gen},
        std::memory_order_release, std::memory_order_acquire));
    // Increment the tail, a new value was added.
    new_tailish.incr();
    // Update the tail.
    // Strong CAS, because not in a loop.
    // No memory barrier required due to acquire-release CAS (I think).
    m_tailish.compare_exchange_strong(old_tailish, new_tailish,
                                      std::memory_order_relaxed,
                                      std::memory_order_relaxed);
    // Success.
    return true;
}

// Returns an optional pointer to a value T. If the queue is empty return
// NULL.
std::optional<T *> dequeue() noexcept {
    IdxGen<N> old_headish;
    // Load the 'current head'. This head is just a hint and
    // does not have to be the 'real' head.
```

```
// No memory barrier required due to acquire-release CAS (I think).
IdxGen<N> new_headish = old_headish =
    m_headish.load(std::memory_order_relaxed);
PtrGen<T> current_entry;

do {
    // Iterate over the buffer while the entry is not valid.
    // No memory fencing required.
    while (!(current_entry =
                m_buff[new_headish.idx].load(std::memory_order_relaxed))
                .is_valid(new_headish.gen)) {
        // The entry is not valid because the pointer is either NULL or not
            in
        // the same generation as the current head.
        if (current_entry.gen == new_headish.gen) {
            // If the entry is in the same generation as the current head
            // the queue is empty.
            if (old_headish < new_headish) {
                // Update the head if its smaller and return.
                // Strong CAS, because not in a loop.
                // No writes, no memory barrier required.
                m_headish.compare_exchange_strong(old_headish, new_headish,
                                                  std::memory_order_relaxed,
                                                  std::memory_order_relaxed);
            }
            return std::nullopt;
        }
        // Increment the head for further iteration.
        new_headish.incr();
    }
    // Finally: Try removing the entry from the buffer. If another thread
    // changed the entry retry. Weak CAS, because loop.
    // Do not reorder any read or writes before/after the successful CAS.
} while (!m_buff[new_headish.idx].compare_exchange_weak(
    current_entry, PtrGen<T>{std::nullopt, (new_headish.gen + 1)},
    std::memory_order_release, std::memory_order_acquire));
// Increment the head, a new value was removed from the buffer.
```

```
        new_headish.incr();
        // Update the head.
        // Strong CAS, because not in a loop.
        // No memory barrier required due to acquire-release CAS (I think).
        m_headish.compare_exchange_strong(old_headish, new_headish,
                                          std::memory_order_relaxed,
                                          std::memory_order_relaxed);
        // Success. Return the pointer.
        return current_entry.ptr;
    }
};

#endif
```

Appendix B

Futex

Listing B.1: Drepper's "mutex2/mutex3" in C++

```
#ifndef __FUTEX_H__
#define __FUTEX_H__

/**
    * A Mutex implementation utilizing Linux's futex system call
    * Basically Ulrich Drepper's "mutex2"/"mutex3" Mutex in modern CPP
    * Described in the paper "Futexes are tricky" (2004)
    * Author: Lucas Craemer
    * */

#include <atomic>
#include <cstddef>
#include <cstdint>
#include <linux/futex.h>
#include <sys/syscall.h>
#include <unistd.h>

// helper functions
static inline int futex(std::uint32_t *uaddr, int futex_op, std::uint32_t val1,
                        const struct timespec *timeout, std::uint32_t *uaddr2,
                        std::uint32_t val3) {
    return syscall(SYS_futex, uaddr, futex_op, val1, timeout, uaddr2, val3);
}
```

```
static inline int futex_wait(std::uint32_t *uaddr, std::uint32_t expected_val) {
    return futex(uaddr, FUTEX_WAIT, expected_val, NULL, NULL, 0);
}

static inline int futex_wake(std::uint32_t *uaddr,
                             std::uint32_t threads_to_wakeup) {
    return futex(uaddr, FUTEX_WAKE, threads_to_wakeup, NULL, NULL, 0);
}

enum FutexState : std::uint32_t {
  UNLOCKED = 0,
  LOCKED_NO_WAITERS = 1,
  LOCKED_ONE_OR_MORE_WAITERS = 2,
};

class RawFutexLock {
private:
    std::atomic<FutexState> m_state;

public:
    constexpr RawFutexLock() noexcept : m_state(FutexState::UNLOCKED) {}

    // fast wait-free lock access try
    bool try_lock() noexcept {
        FutexState current_state = FutexState::UNLOCKED;
        return m_state.compare_exchange_strong(
            current_state, FutexState::LOCKED_NO_WAITERS, std::memory_order_relaxed,
            std::memory_order_acquire);
    }

    // lock algorithm from Drepper's mutex2
    void lock2() noexcept {
        FutexState current_state = FutexState::UNLOCKED;
        // IMPORTANT: acquire barrier when trying to access the lock
        // this implements the acquire-release protocol around the atomic within
        // the lock
        if (!m_state.compare_exchange_strong(
```

```
            current_state, FutexState::LOCKED_NO_WAITERS,
            std::memory_order_relaxed, std::memory_order_acquire)) {
        do {
            // if there are already one or more waiters
            // or if the lock is set with no waiters, set the state
            // to LOCKED_ONE_OR_MORE_WAITERS
            // if this fails the current state is either unlocked
            // or it is locked with one or more waiters
            // both cases require one CAS per loop iteration if there
            // is no spurious failure
            if (current_state == FutexState::LOCKED_ONE_OR_MORE_WAITERS ||
                m_state.compare_exchange_weak(
                    (current_state = FutexState::LOCKED_NO_WAITERS),
                    FutexState::LOCKED_ONE_OR_MORE_WAITERS,
                    std::memory_order_relaxed, std::memory_order_relaxed)) {
                futex_wait((std::uint32_t *)&m_state,
                           FutexState::LOCKED_ONE_OR_MORE_WAITERS);
            }
            // hopefully the futex is unlocked now
            current_state = FutexState::UNLOCKED;
            // at this point either the thread was woken up or the futex was
            // unlocked or spurious failure try to acquire lock and set the new
            // state to LOCKED_ONE_OR_MORE_WAITERS because there is
            // no certainty whether other threads are (were) still waiting
            // IMPORTANT: acquire barrier again when accessing the atomic
        } while (!m_state.compare_exchange_weak(
            current_state, FutexState::LOCKED_ONE_OR_MORE_WAITERS,
            std::memory_order_relaxed, std::memory_order_acquire));
    }
}

// lock algorithm from Drepper's mutex3
void lock3() noexcept {
    FutexState current_state = FutexState::UNLOCKED;
    // IMPORTANT: acquire barrier when trying to access the lock
    // this implements the acquire-release protocol around the atomic within
    // the lock
```

```
    if (!m_state.compare_exchange_strong(
            current_state, FutexState::LOCKED_NO_WAITERS,
            std::memory_order_relaxed, std::memory_order_acquire)) {
        // when the CAS fails this thread becomes a waiter
        // set the state to LOCKED_ONE_OR_MORE_WAITERS if thats not already
        // the case
        if (current_state != FutexState::LOCKED_ONE_OR_MORE_WAITERS)
            current_state =
                m_state.exchange(FutexState::LOCKED_ONE_OR_MORE_WAITERS,
                                         std::memory_order_relaxed);
        // the futex was possibly unlocked
        while (current_state != FutexState::UNLOCKED) {
            // futex is not unlocked call into futex syscall
            futex_wait((std::uint32_t *)&m_state,
                       FutexState::LOCKED_ONE_OR_MORE_WAITERS);
            // the thread was woken up and the futex is probably unlocked
            // try to acquire lock and set the new state to
            // LOCKED_ONE_OR_MORE_WAITERS because there is no certainty
            // whether other threads are (were) still waiting
            // IMPORTANT: acquire barrier againwhen accessing the atomic
            current_state =
                m_state.exchange(FutexState::LOCKED_ONE_OR_MORE_WAITERS,
                                         std::memory_order_acquire);
        }
    }
}

void unlock() noexcept {
    // IMPORTANT: release barrier to "publish" all previous non-atomic writes
    if (((std::atomic<std::uint32_t>)m_state)
            .fetch_sub(1, std::memory_order_release) !=
        FutexState::LOCKED_NO_WAITERS) {
        // when there are one or more waiters the threads should be woken up
        // and the subtraction did not do the trick yet to unlock the atomic
        m_state.store(FutexState::UNLOCKED, std::memory_order_relaxed);
        // wake up one thread in the kernel queue
        futex_wake((std::uint32_t *)&m_state, 1);
```

```
        }
    }
};

#endif
```