國　立　清　華　大　學

資訊工程學系

碩士論文

kpgpool: An In-Kernel eBPF Based PostgreSQL Connection Pool

kpgpool：基於 eBPF 實作的 Linux 內核 PostgreSQL 連線池

研究生：	陳劭愷 (Shao Kai Chen)
學號：	111062605
指導教授：	周志遠 教授 (Prof. Jerry Chou)

中華民國一一三年六月

kpgpool: An In-Kernel eBPF Based PostgreSQL Connection Pool

Student: Shao Kai Chen

Advisor: Prof. Jerry Chou

Department of Computer Science

National Tsing Hua University

Hsinchu, Taiwan, 30013, R.O.C.

June 2024

Abstract

Connection pooling is a common technique that allows multiple clients to share and reuse connections. Database connection pooling can reduce the cost of opening and closing connections, especially for traditional database systems, such as PostgreSQL. Traditional connection pools are implemented as user-space applications, which take advantage of high compatibility, security, and isolation networking stack provided by the Linux kernel. However, they suffer from low performance due to excessive user-kernel crossings and kernel networking stack traversing.

We present kpgpool, an eBPF-based connection pool for PostgreSQL that proxies packets between clients and the PostgreSQL server before the packets enter user-space. Our experiments show that kpgpool improve throughput by 19% and lower latency by 10% with a significantly lower CPU usage comparing to other user-space pools.

The source code has been made available at https://github.com/justin0u0/kpgpool.

Keywords: eBPF, Kernel Bypass, Socket, Database, PostgreSQL, Connection Pool

摘要

連線池是一種常見的技術，允許多個客户端共享和重複使用連線。資料庫連線池可以減少開啓和關閉連線的成本，特別是對於傳統的資料庫系統，如 PostgreSQL。傳統的連線池作爲用户空間應用程式實現，利用 Linux 核心提供的高相容性、安全性和隔離的網路堆疊。然而，由於過多的用户核心交互和核心網路堆疊遍歷，它們的性能較低。

我們提出了 kpgpool，一個基於 eBPF 的 PostgreSQL 連線池，在資料包進入用户空間之前代理客户端和 PostgreSQL 伺服器之間的資料包。我們的實驗表明，kpgpool 在與其他用户空間連線池相比，能提高 19% 的吞吐量並降低 10% 的延遲，同時顯著降低 CPU 使用率。

關鍵詞: eBPF, 内核旁路，套接字，資料庫，PostgreSQL，連線池

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my professor, Jerry Chou, for providing invaluable resources and guidance throughout my research. From the paper reading phase to the final stages of paper writing and presentation preparation, his advice and support have been instrumental.

I would also like to extend my thanks to my fellow lab members, especially my partner who researched eBPF alongside me. His assistance in setting up the VM environment for our workstations greatly facilitated the early stages of our research. Our frequent discussions and mutual problem-solving were crucial to our progress.

I am deeply thankful to my family and friends for their unwavering support and encouragement, which were essential in helping me complete my master's degree. Their emotional support played a significant role in the completion of this thesis.

Balancing my academic responsibilities with an internship was a challenging task. However, with the help and support of all the aforementioned individuals, I was able to successfully and timely complete this thesis. Thank you all.

Contents

List of Figures

Chapter 1

Introduction

PostgreSQL [29], a robust open-source relational database management system, serves as a foundational element in diverse applications and is recognized for its reliability, feature robustness, and performance. It is used extensively by cloud providers and large-scale web services.

Connections are developed to communicate with the PostgreSQL, and are luxurious to create. On connection startup, it forks a process, opens a network socket, starts SSL handshakes (optional) and sends startup packets to authenticate the user. Further, for each connection, resources such as *work_mem* [34] and *temp_buffers* [33] are allocated with about 12MB memory footprint per process by default.

For applications at small scale, the startup costs are not expensive enough to worry about. However, as the application scales up, the constant opening and closing of connections becomes more expensive and can become a bottleneck of the performance.

Database connection pooling [10] [30] [17] is a method to reduce the startup cost of connections by preserving and reusing multiple open connections with the database. The connection pool can be an external service positioned between the applications and the database, redirecting packets between them. In the context of PostgreSQL, PgBouncer [25] is a popular and lightweight connection pool solution.

In terms of pooling options, PgBouncer supports three distinct modes [26]: session pooling, in which a connection in the pool remains assigned to the client until the client closes it; transaction pooling, where a connection is returned to the pool after every transaction is completed; and statement pooling, where a connection is returned to the pool after each query is executed. While statement pooling is less practical since transactions are not permitted in this mode, the transaction mode can facilitate an increase in the maximum number of concurrent clients compared to the session mode, provided that the maximum number of concurrent transactions does not exceed the number of connections available in the pool.

Conventional database connection pools constitute user-space applications that establish communication channels between clients and databases, continuously reading and redirecting packets from client sockets to the database sockets, and vice versa. To facilitate features such as transaction pooling, the proxy must comprehend the database L7 network protocols to get the transaction state and subsequently determine whether the connection between the client and the server should be maintained. User-space applications can readily implement these features without concerns regarding the TCP network stack implementation, as the operating system manages it and provides a high-level socket API for utilization. However, user-space applications incur a significant overhead due to the operating system calls for each read and write operation to the socket.

To eliminate the user-kernel crossing overhead, some approaches utilize kernel-bypass methods such as DPDK to achieve this goal. However, DPDK-based approaches require a complete rebuild of the networking stack, which increases engineering complexity and is not conducive to cloud environments. In contrast, our approach leverages eBPF to address this issue. With eBPF, we can intercept packets in the kernel and redirect them to a different socket without the need for the packet to enter user-space. Furthermore, we can embed database network protocol logics directly into the eBPF program, enabling the implementation of advanced features such as connection pooling, transaction pooling and efficient handling of prepared statements. This approach offers significant performance benefits without sacrificing the kernel networking stack safety.

We present kpgpool, a PostgreSQL-compatible connection pool using eBPF. kpgpool supports both session and transaction modes, as well as efficient handling of prepared statements. Our evaluation shows that kpgpool achieves higher throughput, lower latency, and reduced CPU usage compared to traditional user-space connection pools.

Main contributions of kpgpool:

1. **eBPF-Based Connection Pool**: We implemented an innovative connection pool using eBPF, which intercepts and redirects packets within the kernel, reducing user-kernel crossing overhead.

2. **Support for Session and Transaction Modes**: kpgpool supports both session and transaction pooling modes, maintaining flexibility and enhancing scalability for various application requirements.

3. **Prepared Statement Handling**: kpgpool introduces a user-space proxy to handle prepared statements efficiently, processing most of the messages directly in the kernel while offloading the more complex parts to user-space to minimize performance impact.

4. **Enhanced Performance**: Through our experiments, we demonstrate that kpgpool significantly improves throughput by 19% and reduces latency by 10% compared to other user-space pools, showcasing the potential for high-performance database connectivity.

Chapter 2

Background

2.1 eBPF

2.1.1 eBPF Architecture

The Linux kernel is partitioned into two distinct areas: kernel space and user space. Kernel space enjoys full privileges over the entire system, whereas user space operates with limited access, with privileged operations like disk or network I/O executed through kernel system calls.

Given the privileged nature of kernel space, the operating system has long served as an ideal environment for implementing observability, security, and networking functionalities. However, the stringent requirements for stability and security make extending the operating system a challenge.

Linux Kernel Modules (LKMs) have traditionally provided a means for programmers to extend the base kernel without altering the kernel source code. These modules can be dynamically loaded into the kernel at runtime, removing the need for kernel recompilation and system reboot. Nonetheless, kernel modules introduces security vulnerabilities and maintenance challenges, as kernel upgrades may render them incompatible, leading to potential system crashes.

eBPF is an in-kernel execution engine designed to execute sandboxed programs within the operating system. eBPF facilitates the extension of kernel functionality without necessitating modifications to the kernel source code or reliance on kernel modules. Notably, eBPF offers inherent safety, efficiency, and cross-version compatibility guarantees, ensuring robust and reliable program execution within the kernel environment.

The general workflow of running the eBPF program is as follows:

1. The eBPF program is authored in a high-level language, predominantly restricted to C.

2. The LLVM compiler translates the C program into eBPF bytecode, generating an object file.

3. The generated bytecode is loaded into the Linux kernel using the bpf system call.

4. The verifier conducts a safety check on the eBPF program.

5. The Just-In-Time (JIT) compiler translates the bytecode into native machine code for execution.

While writing eBPF programs, developers encounter certain constraints. Only a subset of C language libraries is available. For example the printf() function are restricted. However, eBPF introduces helper functions for program execution within the kernel environment. For instance, the *bpf_printk* function enables printing of debugging messages.

The eBPF verifier [11] conducts several checks to ascertain the program's compliance with predefined criteria. Firstly, the verifier examines the program's instruction count. which is 4096 until kernel version 5.2, which increased the number to 1 million. Additionally, the verifier verifies whether the program terminates and whether all memory accesses within the program adhere to the allocated memory ranges. This also enforces the program to always perform a boundary check when accessing bytes of a packet.

2.1.2 eBPF Program Type and Hooks

Each eBPF program is designated a type, dictating its input type, available set of helper functions, and the hooks it can be attached to. For instance, a *BPF_PROG_TYPE_KPROBE* program type is attachable to a kernel probe or a user probe, while a *BPF_PROG_TYPE_SYSCALL* program type can be affixed to a system call. Similarly, a *BPF_PROG_TYPE_XDP* program type can be linked to the express data path, serving as an early hook in the RX path of the kernel networking stack.

Once an eBPF program is loaded for execution, it must be attached to hooks, enabling the program to execute whenever specific events occur. Linux offers a diverse set of hooks, including system calls, function entry/exit points, kernel tracepoints, network events, and various others. Moreover, it is feasible to create kernel probes (kprobes) or user probes (uprobes) to attach eBPF programs to nearly any location within the kernel or user applications.

2.1.3 eBPF Map

eBPF introduces a fundamental mechanism known as Map, which serves as a key-value storage enabling data exchange between user-space applications and eBPF programs, as well as among multiple eBPF programs. Access to eBPF maps is facilitated through the bpf system call from user-space or via helper functions from within eBPF programs.

Each eBPF map must be defined with a specific type and a fixed size at compilation time. Linux provides a variety of map types, including array, hash map, queue, stack, ring buffer and bloom filter, for various data storage and retrieval requirements.

In terms of concurrency control, eBPF programs execute within the kernel space, which is non-preemptible from the user-space applications view. Hence, it is inherently safe for a single user-space application to access the map at any given time. For concurrent access from multiple eBPF programs, developers can adopt different strategies. They may opt for a per-CPU map variant, where each CPU core possesses its dedicated map accessible to multiple map types such as array and hash. Alternatively, developers can employ *bpf_spin_lock* for manual locking from within the eBPF program [37]. Certain map types, such as queue and stack, are already thread-safe by design, employing spinning locks to ensure data integrity amidst concurrent access [5].

2.1.4 Kernel Packet Flow

To comprehend the eBPF network hooks provided by the Linux kernel, we delve into the packet flow [24] [19], focusing primarily on the receive side. Although the sending side likely exhibits a reverse flow, it's essential to note potential variations in eBPF hook support between the two sides:

1. The flow initiates when a packet arrives at the Network Interface Card (NIC) device. The NIC forwards the packet to the designated RX queue, typically a ring-buffer in RAM that the driver reserves at boot, often through direct memory access (DMA). In earlier kernel versions, each incoming packet triggered a hardware interrupt (IRQ) by the NIC device, incurring significant overhead. However, the introduction of NAPI addressed this issue by implementing a software interrupt (SoftIRQ) mechanism, employing a polling approach to achieve a balance between interrupts and polling.

2. Upon receipt, the kernel network stack takes charge of processing the packet with the CPU. The *clean_rx* path transforms the packet into a socket buffer object, known as skb.

3. The skb then traverses through the Generic Receive Offload (GRO) implementation within the kernel, before reaching the Traffic Control (TC) subsystem. The TC subsystem is responsible for shaping, scheduling, policing, and potentially dropping the packet. It navigates the packet through various queuing disciplines (qdisc), with the simplest being the first-in-first-out queue (pfifo). More complex qdiscs may be classful, featuring nested classes (qdiscs). A filter associated with a classful qdisc determines the class to which

a packet will be enqueued.

4. Following the TC subsystem, the packet encounters the netfilter component, which commonly interfaces with user-space programs like iptables and its successor, nftables. Netfilter operates over the packet from the end of layer 2, just after the TC subsystem, up to the beginning of the layer 4 networking stack (e.g., *udp_rcv*, *tcp_rcv*). It registers multiple hooks, such as before and after *ip_rcv*, executing diverse tasks including packet filtering, network address translation, or port translation.

5. Subsequently, the layer 4 networking stack undertakes protocol implementation tasks specific to the packet's protocol, such as checksum verification and congestion control for TCP connections. Upon completion of protocol-specific tasks, the stack conducts a lookup of the socket associated with the port. It then put the skb onto a linked list known as the *sk_receive_queue*. At last, the stack calls *sk_data_ready* to mark the socket as having data, ensuring that the receiving application is promptly notified of the packet's arrival.

6. Finally, the client, which is listening to a socket, can be notified by the epoll system call. Upon receiving the notification, it employs the recv system call to copy the packet buffer from kernel-space to the user-space application for further processing.

2.1.5 XDP, TC and the sk_skb Hooks

In the ingress packet flow, the XDP, TC, and sk_skb hooks are triggered at different points. They possess the capability to either pass, drop, modify, or redirect the packet

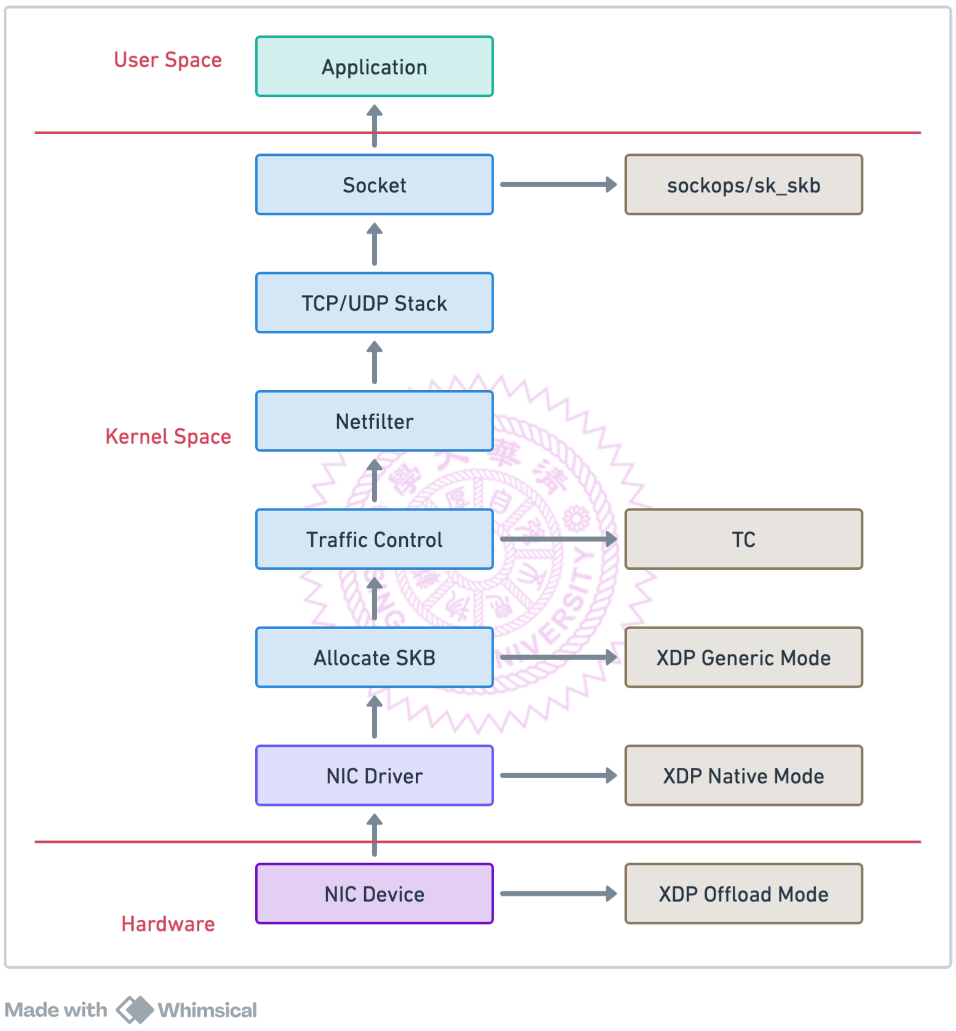


Figure 2.1: The XDP, TC and sk_skb programs along with the kernel packet receiving flow.

within the hook. However, they serve distinct purposes and cater to vastly different use cases.

XDP

The eXpress Data Path (XDP) type program serves as the initial hook trigger in the ingress packet flow, and is absence from the egress packet flow. It can operate in one of three modes: offload, native, or generic. Offloaded XDP enables the program to execute directly on the NIC device if supported, such as with Netronome Agilio CX SmartNICs [43]. In native mode, the program runs if the driver supports it, while drivers lacking XDP support force the program to execute significantly upstack, after skb allocation in the receive_skb path.

The XDP program offers three primary actions: pass, drop, or transmit. When a packet is passed, it continues its normal traversal through the network stack. Dropping packets is valuable for packet filtering tasks, such as DDoS protection [2], as it halts unnecessary network stack traversal at the earliest stage. Transmitting packets can facilitate the creation of load balancers like Katran [18], Meta's L4 load balancer, which leverages XDP as its foundation.

It's important to note that load balancers of this nature typically operate on a per-transport connection basis rather than per-packet basis. Per-transport connection load-balancing ensures that all packets that belong to a transport connection are sent to the same backend. Per-packet load balancing often requires modifications to packet MAC, IP, or port information, necessitating checksum recalculation. A TCP packet is even more challenging to transmit on a per-packet basis, as TCP packets encapsulate critical states such as the sequence and acknowledgment numbers. Transmitting these packets across different connections is not feasible, making per-packet load balancing significantly complex.

TC

The TC type program loads as a specialized qdisc called clsact [40] [22]. Similar to the XDP type program, it possesses the capability to pass, drop, and transmit packets. However, there are two key distinctions between TC and XDP programs:

First, the input context differs. TC programs receive sk_buff, whereas XDP programs receive xdp_buff. Since TC is triggered after skb allocation, sk_buff contains additional information such as priority, queue_mapping, hash, VLAN meta, and more. This enables functionalities like switching network namespaces of packets within the same host using the *bpf_redirect_peer* [6] helper function. For instance, the Cilium project utilizes this feature for faster physical NIC to Pod NIC transmission or local Pod to Pod communication within the Kubernetes network [3]. However, TC is not suitable for per-packet redirection just like the XDP as the L4 protocol may need to be reimplemented.

Secondly, TC can also be triggered from the egress path. This capability allows for the cloning of packets without multiple *send* system calls from user-space, thereby reducing unnecessary user-kernel context switching.

The TC program, however, exhibits lower performance compared to the XDP program. In a comparison conducted by the Cilium project, both TC and XDP were employed for North-South traffic management in Kubernetes. The results demonstrated that XDP can process 10Mpps of inbound packets, whereas TC can handle only approximately 2.8Mpps [4].

sk_skb

The sk_skb type program enables access to the skb upon receiving packets from the socket, thereby facilitating actions such as passing, dropping, modifying, or redirecting the packet. This functionality is used in conjunction with a sockmap (or a sockhash) - a specialized eBPF map designed to store references to socket structures and associated values. The sk_skb type program operates by attaching the eBPF program to the sockmap, where all packets sent through the sockets stored in the map trigger the program for processing.

In the sk_skb type program, there are two types of attachment. The *BPF_SK_SKB_STREAM_PARSER* program is responsible for determining how much data has been parsed, and consequently, how much data needs to be queued to reach a verdict. On the other hand, the *BPF_SK_SKB_STREAM_VERDICT* programs determine the direction of the packet. The parser program can be omitted. Both types of programs give *sk_buff* as input, which provides crucial information, including the source IP, source port, destination IP, and destination port of the packet. Additionally, it contains two important pointers, ‘data‘ and *data_end*, that define the valid memory range for data access.

Unlike the XDP and TC programs, the sk_skb type program operates behind the L4 networking stack, making per-packet redirection feasible.

Note that while the sk_skb program functions solely on the ingress path, there exists the sk_msg program that operates on the egress path when sending packets to the socket. The sk_msg program operates similarly to the sk_skb program.

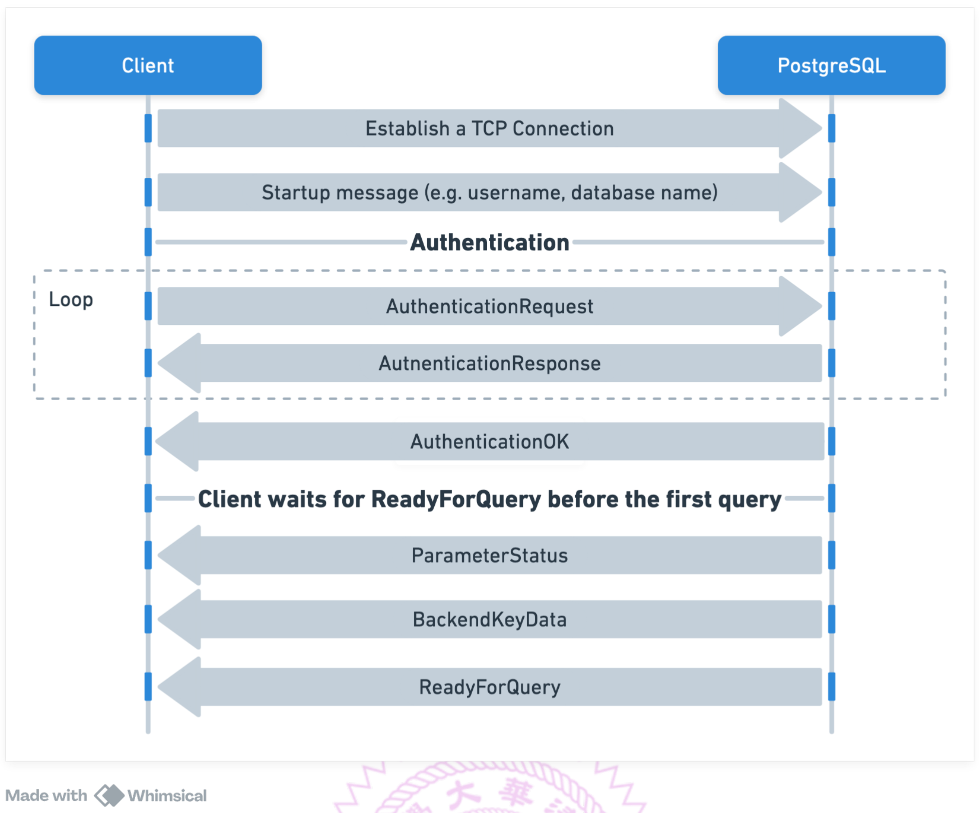


Figure 2.2: PostgreSQL Startup Messages

2.2 PostgreSQL Message Flow

2.2.1 Startup Message

Figure 2.2 shows the PostgreSQL message flow [32] starts after the client establish a TCP connection to the server.

1. Initially, the client sends a startup message packet to the server, which includes the protocol versions, user name, the target database name, and optional parameters.

2. Following the startup message, the server engages in the authentication process by sending an authentication request message packet to the client. The

client responds with the appropriate authentication response message. This exchange may occur multiple times, depending on the authentication method used. It's important to note that while omitting the authentication step is feasible with trusted clients, it's generally discouraged due to security considerations.

3. The authentication cycle ends with the server either rejecting the connection by sending an ErrorResponse message packet or proceeding with an AuthenticationOK message.

4. Upon receipt of AuthenticationOK, the client awaits further communication from the server. Typically, the server sends BackendKeyData for future cancellation, several ParameterStatus messages for informational purposes, and ultimately, a ReadyForQuery message informing the startup is completed.

The client can issue commands after the ReadyForQuery message is received. PostgreSQL supports two different protocols for database query: simple query and extended query.

2.2.2 Simple Query Protocol

1. Optionally, a RowDescription message packet signals the imminent return of rows in response to a (*SELECT*, *FETCH*, etc) query.

2. Zero or more DataRow message packets for each row returned.

3. A CommandComplete message packet indicates that the SQL command is completed successfully.

4. A ReadyForQuery message packet indicates that the query is completed. Note that a SQL query can contains multiple commands, resulting in multiple message packets from point 1 to 3.

The ReadyForQuery message includes a transaction status of the current connection. It can be either *'I'* if idle (not in a transaction block), *'T'* if in a transaction block or *'E'* if in a failed transaction block.

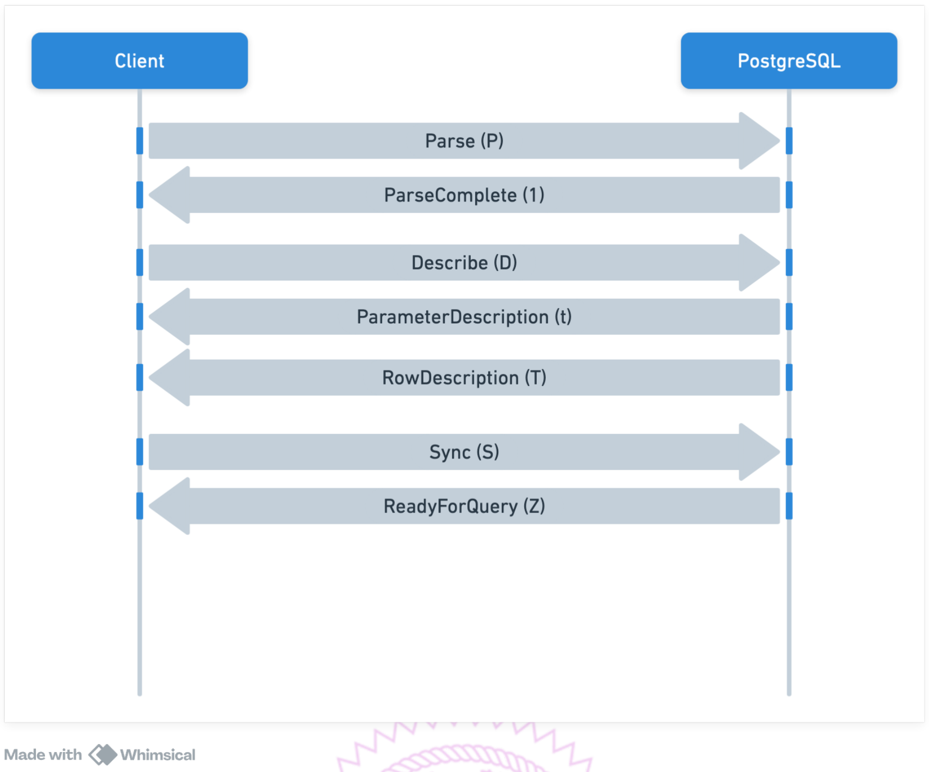
2.2.3 Extended Query Protocol

The extended query protocol [28] provides better performance by reducing the parsing and planning time with prepared statements, and smaller network bandwidth by using binary format data. However, the prepared statements live at the session level, which does not fit into the transaction mode of the connection pool if no additional work is done [23] [27].

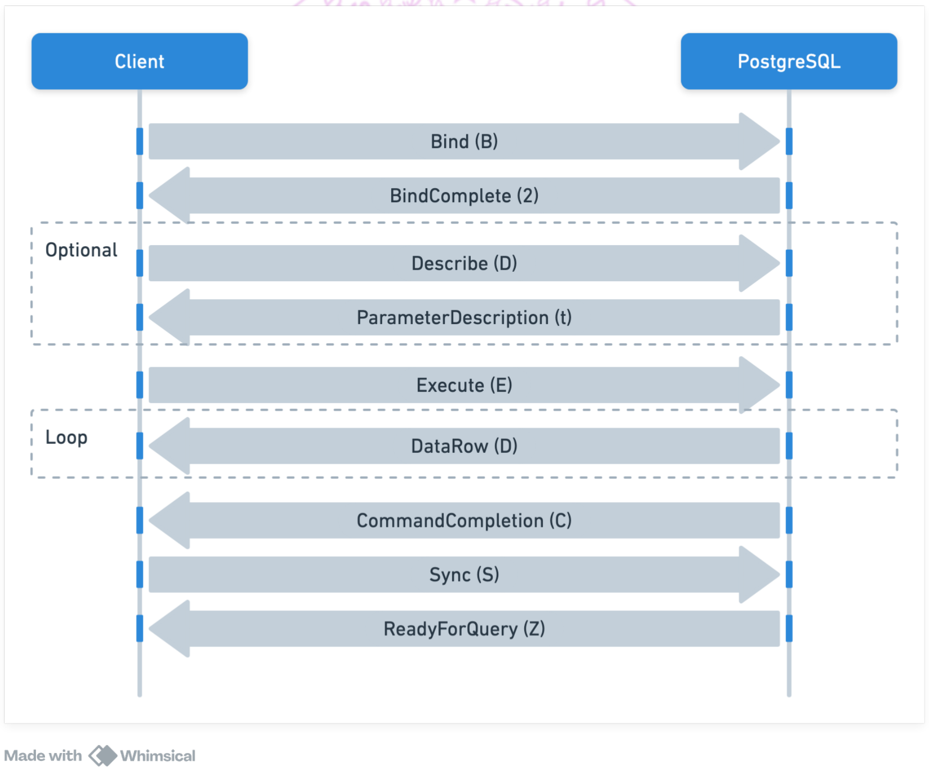
In the extended query protocol, clients initiate communication by sending a Parse message, which includes the textual query string and the prepared statement name. This may be followed by a Describe message, instructing the server to return information about the parameters and the expected return row (such as data types). A Sync message is sent after each series of extended-query messages, prompting the server to respond. Typically, clients send the Parse, Describe, and Sync messages in a single packet, and the server responds to each message in a single packet as well.

If the query is successfully parsed, the server returns a ParseComplete message, indicating that prepared-statement is created and lasted till the end of the current session. After that, the prepared statement is ready to be executed.

Upon successful parsing of the query, the server returns a ParseComplete mes-



(a) Parse, Describe and Sync



(b) Bind, Describe, Execute and Sync

Figure 2.3: Extended query protocol message flow in PostgreSQL.

sage, indicating that the prepared statement has been created and will remain until the end of the current session.

Once the prepared statement is ready for execution, clients proceed by sending a Bind message, specifying the prepared statement name along with its parameters. This may be followed by a Describe message for return row information. Immediately after, an Execute message is sent to initiate execution.

The server's response to these commands includes DataRow messages, a CommandComplete message, and a ReadyForQuery message after receiving a Sync message. Those messages are similar as described in the simple query message flow.

Chapter 3

Related works

We discuss previous research on eBPF programming, eBPF networking, and other implementations that bypass the kernel networking stack in DBMS.

Distributed Protocols in Kernel-Space: "Electrode: Accelerating Distributed Protocols with eBPF" [45] and "DINT: Fast In-Kernel Distributed Transactions with eBPF" [46] demonstrate performance gains by implementing distributed protocols, such as consensus algorithms or transactions, in kernel space.

Caching in Kernel-Space: "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing" [14] uses an XDP program in kernel space to cache records, achieving lower latency. Conversely, "FarReach: Write-back Caching in Programmable Switches" [38] caches records in programmable switches for higher throughput but requires hardware support.

kpgpool applies a similar concept by moving a network-intensive application from user-space to kernel space to achieve higher throughput.

eBPF Programming: "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications" [42] provides an in-depth exploration of eBPF programming workflows, limitations, and potential workarounds. It compares XDP and TC, giving valuable insights into selecting the appropriate program types for different use cases. "Fast In-Kernel Traffic Sketching in eBPF" [21] implements "Nitrosketch: robust and general sketch-based monitoring in software switches"[20] in eBPF. The study highlights that stringent eBPF programming constraints make migration challenging. For instance, hash tables are limited due to the lack of dynamic allocation and unbounded loops in eBPF. The paper also notes that the fastest user-space implementation may not perform similarly in eBPF.

These works show that eBPF programs have constraints. We need to carefully consider the capabilities of each program type and make workarounds to bypass the limitations. kpgpool adopts a similar idea by carefully splitting user-space and kernel-space responsibilities to ensure feasible implementation.

DPDK and eBPF Comparison: "Revisiting the Open vSwitch DataPlane Ten Years Later" [41] notes that OVS was originally implemented with a kernel module. Due to maintenance and performance considerations, they explored DPDK and eBPF. However, DPDK is difficult to maintain and requires a dedicated CPU core, leading them to ultimately adopt eBPF with the AF_XDP program as the final solution.

eBPF in Storage Systems: "XRP: In-Kernel Storage Functions with eBPF"[44] observes that databases commonly access storage devices using B-Tree or LSM Tree patterns, which lead to multiple context switches and incur overhead. They found that using SPDK, similar to DPDK, presents issues such as lack of access control and file system support due to the absence of OS integration. Therefore, they extended eBPF to offload custom code onto NVMe devices, effectively bypassing the kernel storage stack.

User-Bypass Connection Pool: "Tigger: A Database Proxy That Bounces With User-Bypass" [8] is an independently and concurrently developed eBPF-based PostgreSQL connection pool. It shows performance improvements over user-space connection pools but lacks support for prepared statements, a crucial feature for enhancing database performance.

DPDK in Database Networking: ScyllaDB [35], a NoSQL database, offers a DPDK networking mode. However, they do not deploy DPDK in production due to the trade-off between performance and the deployment or maintenance burden [36]. To the best of our knowledge, there are no other papers implementing a kernel-bypass connection pooling solution.

These works conclude that while complete kernel-bypass solutions often provide better performance, the trade-off between performance and maintainability needs to be considered. eBPF offers a better balance in this trade-off. kpgpool integrates eBPF for this reason.

Chapter 4

Motivation

4.1 Naive TCP Proxy Performance Gain through eBPF

To evaluate the performance enhancement achieved using the eBPF sk_skb program compared to the user-space application, we implemented a TCP proxy setup. This proxy facilitates packet exchange between the client and server across both versions, allowing us to evaluate the respective efficiencies in handling TCP traffic.

The sequence of events unfolds as follows: Initially, the client establishes a TCP connection with the proxy, which subsequently establishes a connection with the server. Once connections are established, the client sends a random bytes message to the proxy. The proxy then forwards this message to the server. Upon receiving the message, the server responds with an echo message, which the proxy intercepts and relays back to the client. This exchange continues in a loop, simulating ongoing communication between the client and server.

4.1.1 The eBPF TCP Proxy

The installation of the eBPF program on the proxy side involves attaching the program to the socket and inserting sockets into the sockmap type eBPF map. The attachment process can be done by the eBPF client library using the *bpf* system call. The insertion of sockets into the sk_skb map is achieved by a specialized type of eBPF program called sockops [7], which triggers in response to specific socket events. These events may include the establishment of active or passive connections, or the retransmission of packets.

The sockops program offers a convenient helper function that facilitates the registration of sockets into the socket map. This mechanism serves as one of the simplest ways to configure the sk_skb type program.

The sockops program is structured to respond to specific events triggered during TCP connection establishment. When a passive TCP connection is established, denoted by the SYN-ACK step in the TCP 3-way handshake, the *BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB* event is invoked. This event signifies the initiation of the client-proxy connection if the local port given matches the proxy port. Conversely, the *BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB* event is triggered during an active TCP connection establishment, corresponding to the ACK step in the TCP 3-way handshake. This event denotes the establishment of the proxy-server connection if the remote port given matches the server port.

The *bpf_sock_map_update* helper function then register sockets into the socket map. Once registered, packets sent through these sockets immediately trigger the eBPF program attached to the sockmap. We specified the client-proxy socket to be associated with key 0 and the proxy-server socket to be associated with key 1 for simplicity.

Regarding the sk_skb program, it checks whether a packet originates from the client or the server by extracting the local and remote port of the packet. It's important to note that the remote port information arrives in network byte order, necessitating the utilization of the *bpf_ntohl* helper function to convert it to the host byte order. Subsequently, the program employs the *bpf_sk_redirect_map* helper function to direct the packet either to the client or the server, assigning key 0 or 1, respectively, for appropriate routing.

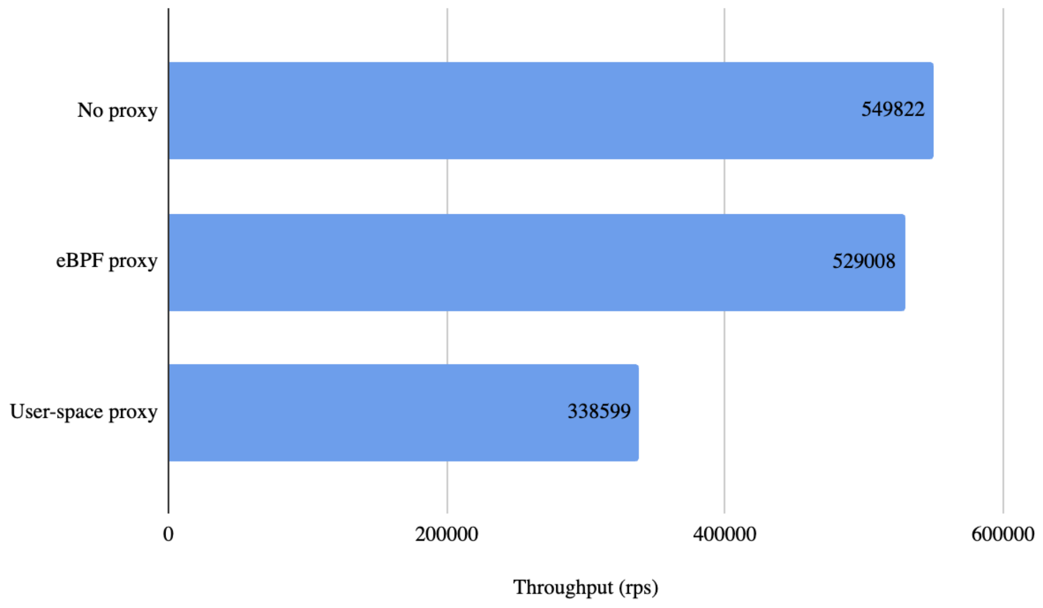
To initiate the process from the user-space program, we attach the sockops program to the cgroup using the cilium/ebpf [9] library. Subsequently, we start a TCP listener on the proxy port. Upon establishing a TCP client connection, the client-proxy socket is automatically registered to the sockmap by the sockops program. On the user-space side, we establish a connection from the proxy to the server for each client connection established. The proxy-server socket is also automatically registered to the sockmap by the sockops program.

4.1.2 The User-Space Proxy

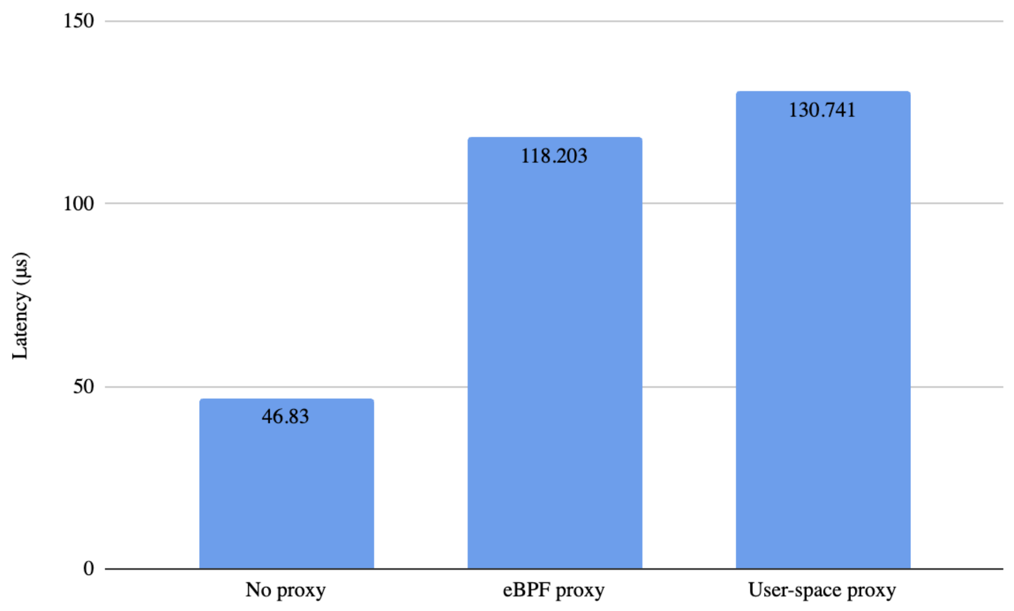
The user-space proxy is implemented through 2 goroutines [16] (lightweight user-thread), with 1 piping requests from the client the the server, and another piping the server to the client.

4.1.3 Evaluation

To evaluate the performance of an eBPF-based TCP proxy against a user-space proxy, we conducted an experiment to measure throughput and latency. The testbed



(a) Throughput



(b) Latency

Figure 4.1: TCP Proxy Performance Comparison

uses three GCP VM instances [13]: one n1-standard-4 machine for the client and two n2-standard-4 machines for the proxy and server. Throughput and latency were measured using the sockperf [39] tool.

The experiment included three scenarios: a baseline with no proxy, an eBPF-based proxy, and a user-space proxy. Figure 4.1 shows that the eBPF-based proxy achieves throughput approximately 1.56 times higher than the user-space proxy. However, the baseline with no proxy still outperforms both, with 1.04 times higher than the user-space proxy. In terms of latency, the eBPF-based proxy has an average latency 1.11 times lower than the user-space proxy. However, with an additional hop, the eBPF-based proxy latency is 2.52 times higher than the baseline's latency with no proxy.

These results demonstrate that the eBPF-based proxy provides a significant throughput and latency improvement over a traditional user-space proxy, suggesting that an eBPF-based connection pool could deliver performance advantages comparing to a traditional user-space connection pool.

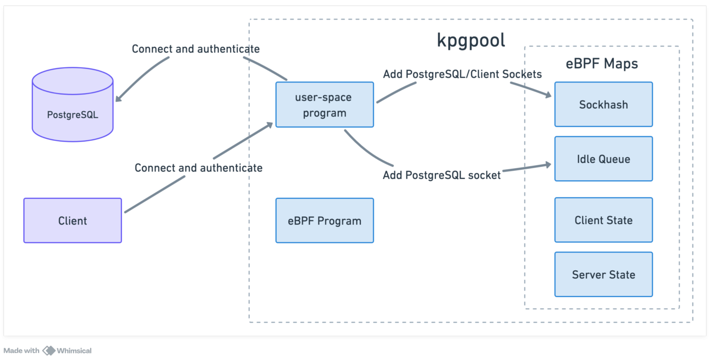
Chapter 5

Implementation

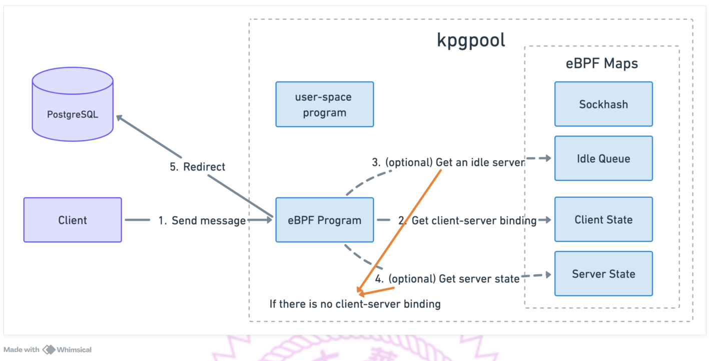
In the following sections, we present how kpgpool is implemented using Go [15] for the user-space program and C for the eBPF program.

5.1 Session Mode

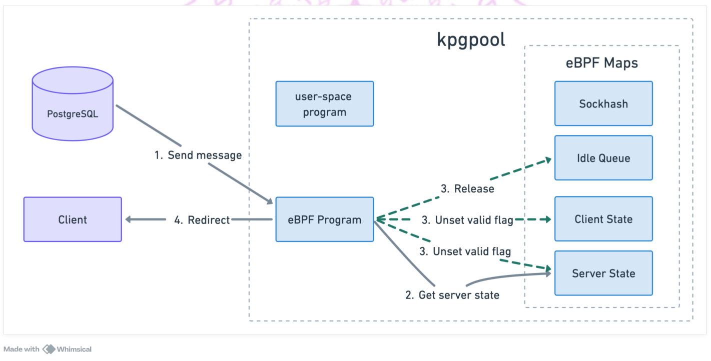
In session mode, a client-proxy connection sticks to the same proxy-server connection throughout the entire session, functioning much like the TCP proxy described earlier. However, the key difference lies in the initial connection setup. As detailed in the ”PostgreSQL Message Flow” section, once the TCP connection is established, authentication must occur before any queries can be executed. If we register the socket to the sockmap (or sockhash) using the sockops program immediately after the TCP connection is established, it would require the eBPF program to handle the authentication process. Due to the stringent programming constraints in eBPF, implementing such an authentication process would be nearly impossible. Instead, we implement the authentication process in the user-space program, registering the connection to the sockmap only after it is ready for query.



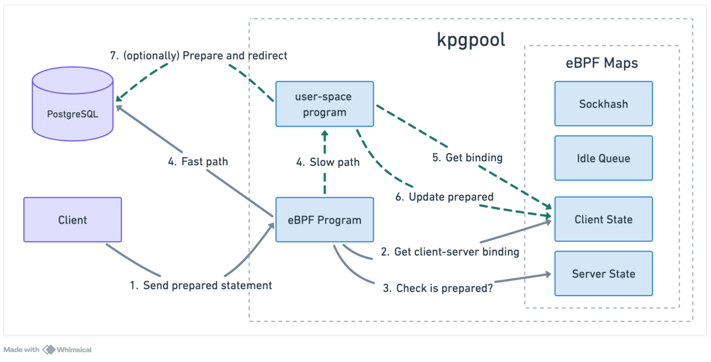
(a) Registering the sockets



(b) Maintaining the sessions



(c) Transaction Mode



(d) Supporting Prepared Statements

Figure 5.1: kpgpool Architecture

5.1.1 Registering the sockets

Implementing the startup and authentication process in user-space is straightforward with the appropriate libraries. When the connection pool program starts, it sets up multiple TCP connections with the server and completes the authentication process for each one. The startup process completes once the connection has received a ReadyForQuery message from the server.

The connection pool program listens on a designated port for incoming client connections. Once a client establishes a TCP connection with the pool, the pool handles the authentication process in user-space. When the authentication is complete, the pool sends a ReadyForQuery message to the client, signaling that it is ready to handle database queries.

After completing the startup process for either a server or client connection, the corresponding socket must be registered with a sockmap or sockhash type eBPF map. This registration ensures that every packet received through the socket will trigger the eBPF program associated with the map.

Unlike the earlier described eBPF TCP proxy, which used a sockmap, we opted for a sockhash. This choice enables us to use the 4-tuple, comprising source IP, destination IP, source port, and destination port, as a key to access the sockhash table.

Figure 5.1(a) shows the how kpgpool' s user-space interact with the client, PostgreSQL and other eBPF maps to setup connections.

5.1.2 Maintaining the sessions

Whenever a packet from either the client or the server arrives at the proxy, it triggers the eBPF program if the socket is registered in the loaded sockhash map. To determine whether the packet originates from the client or the server, we check its 4-tuple, which includes the source IP, destination IP, source port, and destination port. Once identified, we can use the *bpf_sk_redirect_hash* helper function to redirect the packet to the appropriate destination.

The pool maintains multiple clients and servers at the same time, which means we need to know which server a client is connected to and vice versa.

We maintain a server state or a client state for each connection, using a hash type eBPF map and keyed by the 4-tuple. The client state holds a valid flag and the server's 4-tuple (source IP, source port, destination IP, destination port), indicating which server the client is connected to. Upon receiving a client packet, the eBPF program gets the client state using the *bpf_map_lookup_elem* helper function. If the valid flag is set, the packet is redirected to the corresponding server using the *bpf_sk_redirect_hash* helper function and the stored 4-tuple. If the valid flag is not set, the system must assign an idle server to the client.

Figure 5.1(b) presents how kpgpool maintain sessions with a client packet.

Handling server packets follows a similar pattern. The server state contains a valid flag and the client's 4-tuple, enabling the system to determine where the server packet should be redirected.

Idle servers are kept in a separate eBPF map of the queue type. After the connection pool completes the startup process for each server, the server's 4-tuple is added to this queue. When a client packet arrives and finds no corresponding server, it retrieves an idle server from the queue, then updates both the client and server states with their respective 4-tuple and sets the valid flag to true.

5.2 Transaction Mode

To increase the maximum number of concurrent clients a connection pool can manage with a fixed number of servers, transaction mode can be used. This mode returns a server to the idle state whenever it is not engaged in an active transaction block. Specifically, when a server sends a ReadyForQuery message indicating an idle transaction state, the connection between the client and server is released, and the server is returned to the idle queue, ready to be assigned to another client.

As noted in the simple query message flow section, a single TCP packet may contain multiple PostgreSQL messages. Each message begins with a one-byte code that identifies the message type, followed by a four-bytes field specifying the total length of the message, including the length of the length field itself.

A for loop can be used to iterate over each message, starting with a offset of 0 at *sk_buff->data* and incrementing it by the specified length for each message. However, eBPF does not support unbounded loops. This limitation means that even if we ensure the pointer stays within safe memory access boundaries, we still need to impose a hard limit on the number of messages per packet and the maximum length of each message to prevent the pointer from incrementing infinitely.

Upon receiving server packets, we iterate over the messages to determine if any include a ReadyForQuery message. If such a message is found and it indicates that the transaction state is idle, we retrieve the client state from the map and unset the valid flags for both client and server. Finally, the server is placed back into the idle queue.

Figure 5.1(c) shows how kpgpool handles a server packet and releases the connection if the transaction ended.

5.3 Supporting Prepared Statement

Prepared statements are a commonly used feature in PostgreSQL, offering performance benefits by allowing the reuse of parsed and planned queries. However, they present a challenge for transaction mode connection pools, as prepared statements are session-specific. Executing a prepared statement created in another session is not feasible without additional handling.

To support prepared statements in a transaction mode pool, the connection pool must track which prepared statements have been created in each session. When a prepared statement needs to be executed in a different session—typically due to a client-server binding change after a transaction is completed—the pool must re-prepare the statement by injecting a Parse message beforehand.

In a user-space program, this can be managed by maintaining a hash map for each client and server connection. For client connections, the prepared statement name is stored as the key, and the complete query string (or the full Parse message) as the value. Each Parse message received from the client updates this map. For server connections, the prepared statement name is mapped to a boolean value indicating whether the statement has already been prepared on that connection.

However, implementing this entirely bookkeeping in eBPF is nearly impossible for several reasons:

1. **Handling Hash Collisions**: Storing string-type keys in a hash map introduces

the potential for hash collisions. Managing these collisions in eBPF is particularly challenging and inefficient, whether using open hashing (requiring dynamic allocation of new hash table entries) or closed hashing (potentially necessitating unbounded loops).

2. **Variable Query Strings**: Query strings can vary in length, which could result in significant space wastage if a large fixed-size value is used in the eBPF map, as dynamic value sizes are not supported in eBPF maps.

3. **Message Injection Complexity**: Injecting messages within the sk_skb program in eBPF is inefficient and challenging due to the strict memory access constraints enforced by the eBPF verifier. This process involves expanding the packet, locating and creating space at the correct offset, and then copying the bytes into the packet.

Due to these limitations, supporting prepared statements in eBPF-based connection pools completely remains a significant challenge. The complexity and constraints of eBPF necessitate alternative approaches for efficiently managing prepared statements in transaction mode.

5.3.1 User-space supported prepared statement

In kpgpool, we support prepared statements by introducing a user-space proxy specifically for handling them.

In the eBPF program, client packets are parsed similarly to the transaction mode described earlier to identify if they contain a Parse or Bind message. Other types of messages in the extended query protocol, such as Execute or Sync, do not need identification as they are always sent after Parse or Bind messages in the same session. These packets are forwarded to the user-space application by returning *SK_PASS* in the eBPF program.

In the user-space application, we set up a thread to receive packets from each client after the socket is bound to the eBPF program. Upon receiving packets, we know they must contain a Parse or Bind message. However, we need to determine which server the client is currently connected to in order to handle these messages. Thankfully, the eBPF program ensures that the client is bound to a specific server by setting up the client and server states before passing packets to the user-space program. We can easily get the designated server by querying the client state using the *bpf_map_lookup_elem* helper function from the user-space program.

Once we know which server the packets belong to, we parse the packets to extract PostgreSQL messages. For a Parse message, we store the prepared statement name along with the query string in the client's hash map and send the Parse message to the server. For a Bind message, we check the server’s hash map by the prepared statement name to see if it is already prepared on this server connection. If it is not, we form a Parse message from the client’s hash map and send it to the server beforehand, followed by the Bind message just received. To reduce network overhead, the injected Parse message can be sent with the Bind message in the same packet.

The server will return an additional ParseComplete message after we send the injected Parse with a Bind message. Most PostgreSQL drivers simply ignore this message, so there is no need to eliminate it. However, if necessary, the message can be handled by setting a boolean state in the server state indicating that we should remove the next ParseComplete message. Removing a single message in eBPF is trivial.

5.3.2 Bypass prepared statement with eBPF

Despite the challenges in fully managing prepared statements within an eBPF program due to hash collision handling complexities, we can implement a solution to redirect most of the Bind messages directly to the server using eBPF, thereby improving overall performance. It's important to note that Parse messages, which are only sent once per unique query when using prepared statements, are still handled by the user-space program. This approach minimizes the performance impact, as the frequency of Parse messages is significantly lower compared to Bind messages.

In the eBPF program, we introduce a 2D array *prepared* to the server state, with a capacity of 256 x 64 entries. This setup allows for storing up to 256 prepared statement names, each limited to 64 bytes, which aligns with PostgreSQL's constraints on prepared statement names [31]. Upon receiving a Bind message, we hash the prepared statement name to determine the corresponding entry and then check if the name matches the stored entry. This verification step is necessary to identify and handle collisions where multiple prepared statement names hash to the same entry. If the prepared statement name matches, it indicates that the statement is already prepared, allowing us to redirect the message directly to the server.

In the user-space program, before sending a Parse message (either received from the client or injected) to the server, it updates the server's *prepared* map using the *bpf_map_update_elem* helper function. If two names result in a single index, the more recent one will overwrite the previous entry, potentially causing the earlier statement to become unprepared. This design may lead to false-negative reports, where a Bind message is incorrectly identified as not prepared. However, this scenario is acceptable because the user-space program can handle these cases using a slower path. As long as collisions are infrequent, most already prepared Bind messages will bypass the user-space and be redirected to the server directly from the eBPF program.

Additionally, errors resulting from failed Parse messages are not a concern. Clients should not use a prepared statement if they have not received a corresponding ParseComplete message. Therefore, updating a failed prepared statement in the map does not cause any issues.

For hashing, we use the FNV-1a [12] algorithm, masking the result to fit within 256 entries. Research [21] shows that more advanced hashing algorithms, such as xxHash, do not provide significant benefits in eBPF due to the lack of SIMD instructions.

By employing this method, we can effectively bypass most of the Bind messages with eBPF, reducing the reliance on the user-space proxy for handling these messages and enhancing overall system performance.

Figure 5.1(d) presents how kpgpool handles a prepared statement packet with either a fast-path that bypass the user-space, or a slow-path that requires the user-space proxy for advanced processing.

Chapter 6

Evaluation

We evaluate **kpgpool** using three GCP VM instances: one n1-standard-4 machine for the client and two n2-standard-4 machines for the connection pool and PostgreSQL server. All machines run on Linux version 6.1.0. The PostgreSQL server operates on version 15.3 with its default configuration.

We evaluate kpgpool by comparing it against two different connection pools.

1. **Baseline**: A non-optimized user-space connection pool implemented in Go, offering minimal features but supports session mode, transaction mode and prepared statements.

2. **PgBouncer**: A production-ready connection pool. The PgBouncer is set on version 1.20.1.

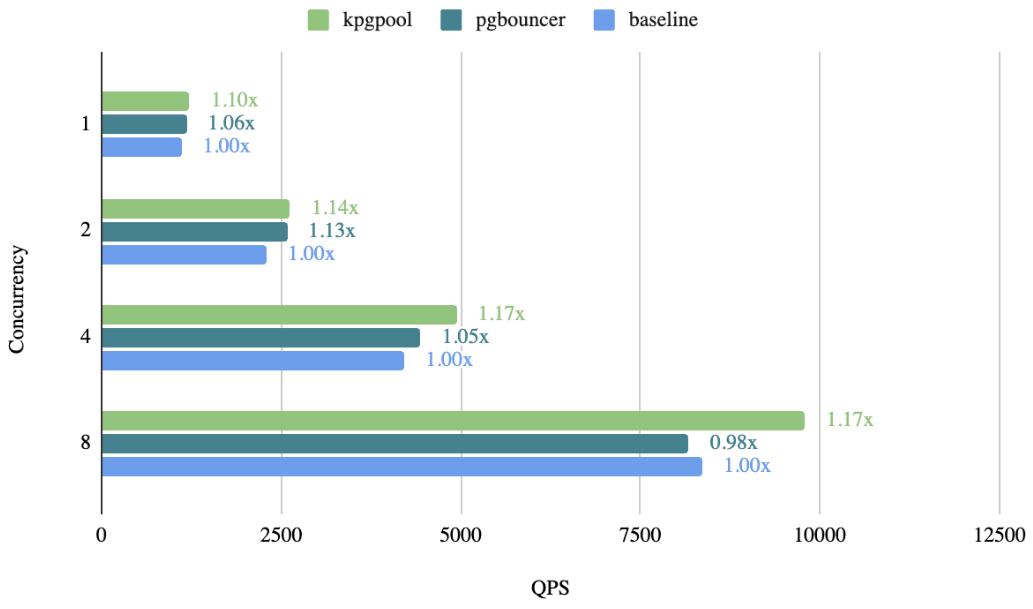


Figure 6.1: Comparing throughput using simple query protocols under transaction mode.

6.1 Simple Query Throughput Under Transaction Mode

To evaluate throughput, we measured performance by varying the number of concurrent clients, each sending queries and awaiting responses. This approach allowed us to assess how the connection pools perform under different loads. Throughput was measured using the simple query protocol: each client sent a *BEGIN* command, followed by a single query, and then a *COMMIT* command. Only the query was counted in the queries per second (QPS) metric.

Figure 6.1 shows that under high load conditions, kpgpool achieved approximately 1.19 times higher throughput than the user-space proxy, highlighting its efficiency in handling high volumes of concurrent queries.

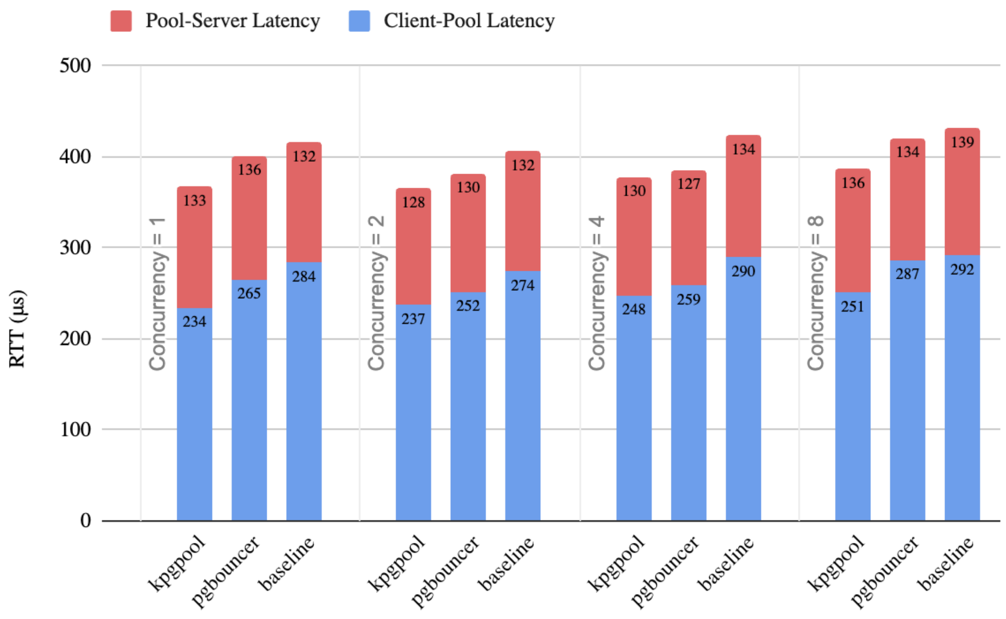


Figure 6.2: Comparing latency using simple query protocols under transaction mode.

6.2 Simple Query Latency Under Transaction Mode

To evaluate the latency of the connection pools, we used the same methodology as for throughput measurement. Latency was measured using bcc’ s [1] tcprtt tool, capturing both the client to pool and pool to server TCP round-trip times (RTT).

Figure 6.2 shows that the pool to server latency remained consistent across all cases, averaging about 130 140 µs. This indicates that kpgpool, did not significantly alter the pool to server latency compared to the user-space implementations. In contrast, the client to pool latency has differences among the pool implementations. Overall, kpgpool achieved a latency that was approximately 10% faster than the PgBouncer.

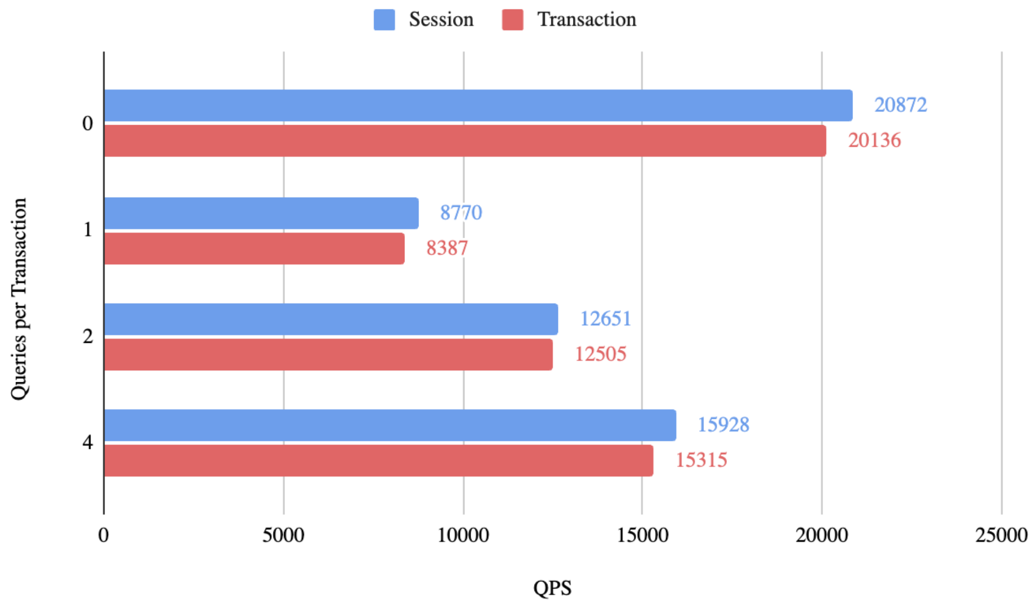


Figure 6.3: Comparing throughput with session and transaction mode under different frequency of connection switches.

6.3 Session Mode Performance

We also evaluated the performance of session mode against transaction mode. Transaction mode introduces a slight overhead due to the need to parse PostgreSQL messages and maintain connections in the eBPF map. However, it increases the maximum number of concurrent clients with a fixed number of server connections.

The performance was measured by varying the number of queries per transaction. We anticipated that as the number of transactions increases, the throughput of transaction mode would decrease due to the additional overhead of releasing and re-acquiring connections. The ”queries per transaction” value of 0 indicates that queries are sent without a transaction, meaning the connection is released after every query.

Figure 6.3 shows that the overhead introduced by transaction mode is negligible. Specifically, the throughput of transaction mode is about 1.03 times lower than

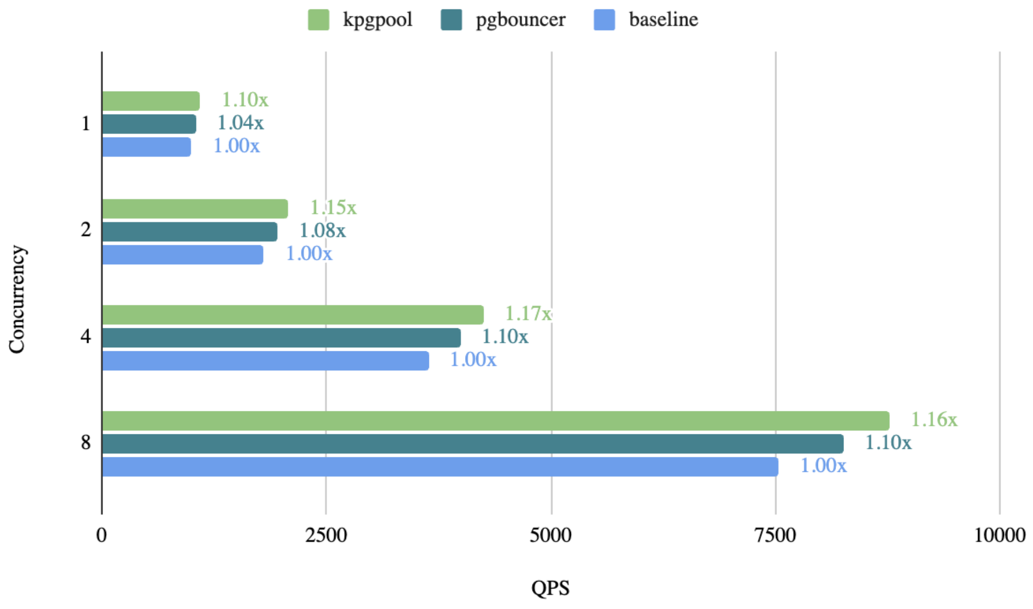


Figure 6.4: Comparing throughput using simple query protocol under session mode.

session mode. Contrary to our expectations, the number of transactions did not significantly affect the overhead.

We also measured the performance of kpgpool in session mode against the user-space pool and pgbouncer to ensure it performs well in comparison to these user-space pools. Figure 6.4 indicates that kpgpool achieves 1.06 times higher throughput than PgBouncer under high load in session mode.

6.4 Extended Protocol Performance

We also measured the performance of kpgpool using the extended query protocol. In the experiment, we includes two queries for each transaction. Identical queries are sent using the same prepared statement name, with only their parameters differing.

It is possible for some prepared statement names to hash into the same entry, leading to unnecessary traversal and user-space handling. To evaluate the performance impact, we also measured a configuration where the fast path was disabled,

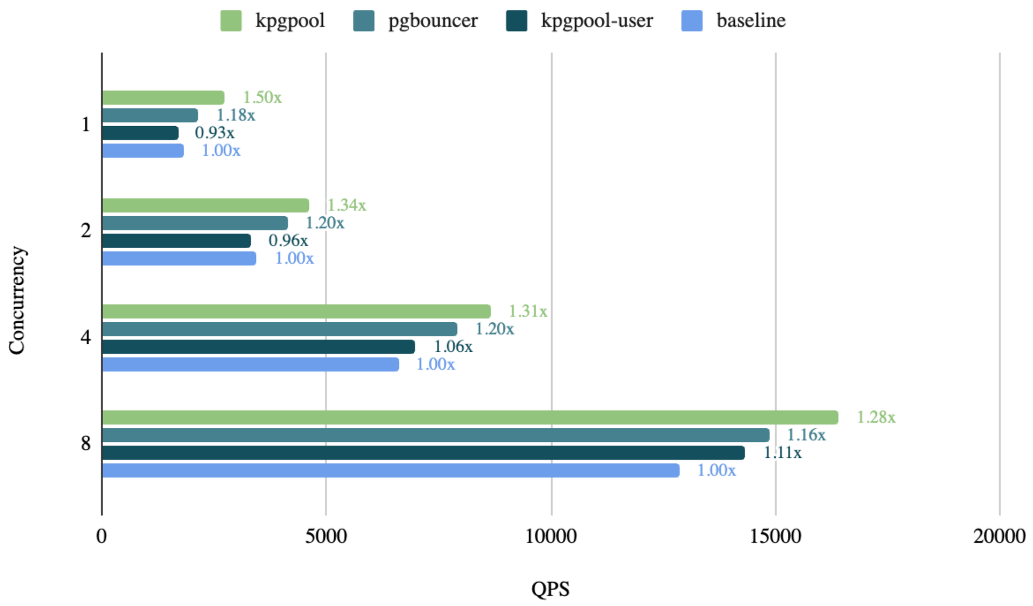


Figure 6.5: Comparing throughput using extended query protocol under transaction mode.

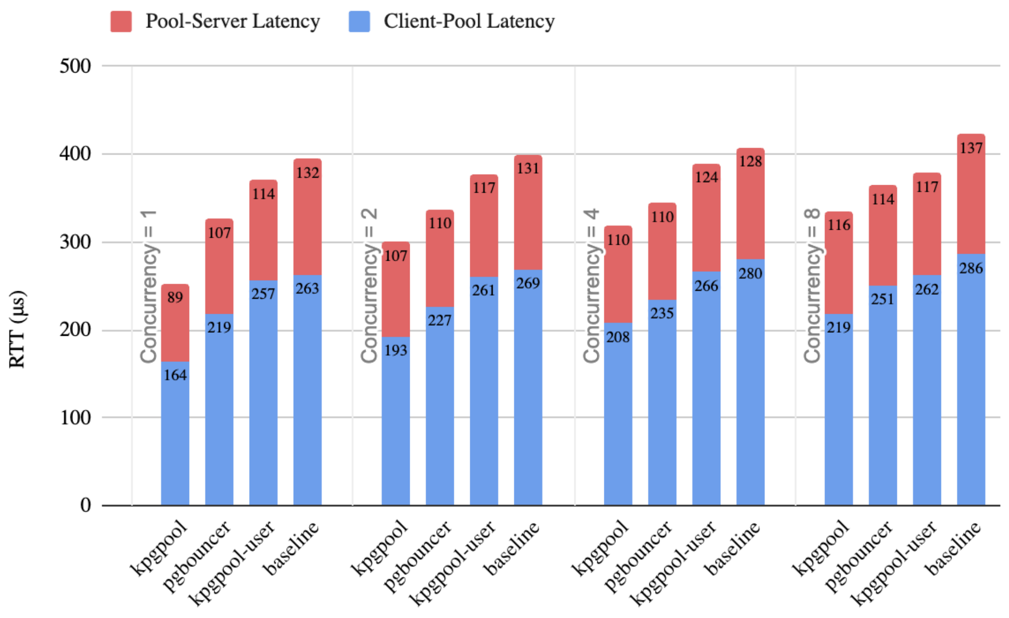


Figure 6.6: Comparing latency using extended query protocol under transaction mode.

named **kpgpool-user**.

Figures 6.5 and 6.6 show that with the extended query protocol, kpgpool outperformed PgBouncer by achieving 1.10 times the throughput and 0.92 times lower latency under high load conditions. In the worst-case scenario, where all prepared statements were handled in user-space via the slow path, kpgpool-user achieved 0.96 times the throughput and 1.04 times higher latency compared to PgBouncer under high load.

In real-world applications, the number of prepared statements should be relatively low, as having too many would negate the performance benefits of using prepared statements. Therefore, the likelihood of hash collisions causing significant performance degradation is minimal. Consequently, kpgpool is expected to provide better performance with the extended query protocol due to its efficient handling of prepared statements.

6.5 CPU Usage

kpgpool is expected to provide higher CPU efficiency by avoiding complete network stack traversal and reducing context switches.

We measured CPU usage by running each pool under a Linux cgroup and collecting cgroup CPU metrics.

Figure 6.7 demonstrate that kpgpool uses only one-third of the CPU compared to PgBouncer while achieving higher throughput. This highlights the efficiency and performance benefits of leveraging eBPF for connection pooling in PostgreSQL.

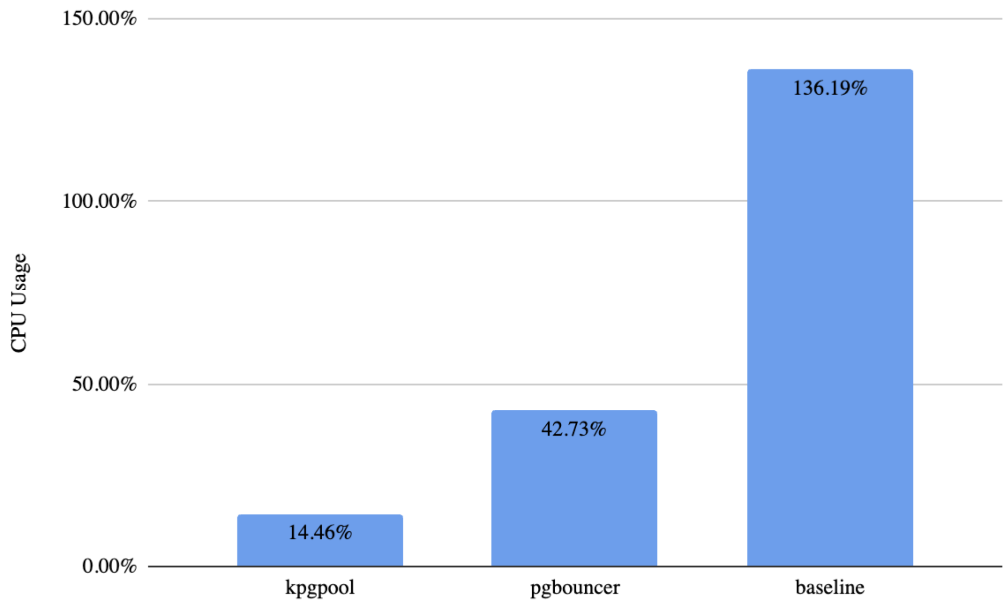


Figure 6.7: Comparing CPU usage using extended query protocol under transaction mode.

Chapter 7

Conclusion

Traditional user-space connection pools for PostgreSQL reduce the cost of opening and closing connections but suffer from performance issues due to excessive user-kernel crossings and kernel stack traversing.

We introduced **kpgpool**, an eBPF-based connection pool for PostgreSQL that proxies packets between clients and the server before they enter user-space. Our experiments show that kpgpool improves throughput by 19% and reduces latency by 15% compared to user-space pools.

These findings demonstrate the potential of eBPF for high-performance database connection pooling, offering a significant boost in efficiency and responsiveness. Out implementations also shows that other types of proxy-based applications can leverage eBPF to achieve similar performance improvements with application-specific logic implemented.

References

[1] BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc.

[2] Bertin, G. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev* (2017), vol. 2, The NetDev Society, pp. 1–5.

[3] Borkmann, D. Virtual Ethernet Device Optimization with eBPF. https://cilium.io/blog/2020/11/10/cilium-19/veth.

[4] Borkmann, D., and Pumputis, M. Kubernetes service load-balancing at scale with bpf xdp. In *Linux Plumber Conference* (2020).

[5] Linux source code of BPF queue/stack maps. https://elixir.bootlin.com/linux/v6.1/source/kernel/bpf/queue_stack_maps.c.

[6] Linux kernel patch: bpf: Add redirect_peer helper. https://github.com/torvalds/linux/commit/9aa1206e8f482.

[7] Linux kerenl patch: bpf: Adding support for sock_ops. https://lwn.net/Articles/727189/.

[8] Butrovich, M., Ramanathan, K., Rollinson, J., Lim, W. S., Zhang, W., Sherry, J., and Pavlo, A. Tigger: A database proxy that bounces with user-bypass. *Proc. VLDB Endow. 16*, 11 (jul 2023), 3335—3348.

[9] cilium/ebpf: ebpf-go is a pure-Go library to read, modify and load eBPF programs and attach them to various hooks in the Linux kernel. https://github.com/cilium/ebpf.

[10] Custer, C. What is connection pooling, and why should you care . https://www.cockroachlabs.com/blog/what-is-connection-pooling/.

[11] Isovalent: eBPF Docs: Verifier. https://ebpf-docs.dylanreimerink.nl/linux/concepts/verifier/.

[12] Fowler—Noll—Vo hash function. https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#FNV-1a_hash.

[13] Google Cloud Platform: feedbackGeneral-purpose machine family for Compute Engine. https://cloud.google.com/compute/docs/general-purpose-machines.

[14] Ghigoff, Y., Sopena, J., Lazri, K., Blin, A., and Muller, G. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 487–501.

[15] Go: An open-source programming language supported by Google. https://go.dev/.

[16] Effective Go: goroutines. https://go.dev/doc/effective_go#goroutines.

[17] Gupta, K., and Mathuria, M. Improving performance of web application approaches using connection pooling. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)* (2017), vol. 2, pp. 355–358.

[18] Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[19] devconf.cz 2018. Linux packet journey, napi, hardware queue, skb. https://www.youtube.com/watch?v=6Fl1rsxk4JQ.

[20] Liu, Z., Ben-Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., and Sekar, V. Nitrosketch: robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2019), SIGCOMM '19, Association for Computing Machinery, p. 334—350.

[21] Miano, S., Chen, X., Basat, R. B., and Antichi, G. Fast in-kernel traffic sketching in ebpf. *SIGCOMM Comput. Commun. Rev. 53*, 1 (apr 2023), 3—13.

[22] Monnet, Q. Understanding tc "direct action" mode for BPF. https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/.

[23] Mullane, G. S. CrunchyData: Prepared Statements in Transaction Mode for PgBouncer. https://www.crunchydata.com/blog/prepared-statements-in-transaction-mode-for-pgbouncer#why-prepared-statements-can-be-a-problem-in-transaction-mode.

[24] Flow of network packets through Netfilter with legacy iptables packet filtering. https://en.wikipedia.org/wiki/Netfilter#/media/File:Netfilter-packet-flow.svg.

[25] PgBouncer: Lightweight connection pooler for PostgreSQL. https://www.pgbouncer.org/.

[26] PgBouncer: Lightweight connection pooler for PostgreSQL: Configuration. https://www.pgbouncer.org/config.html#pool_mode.

[27] PgBouncer: Support of prepared statements. https://github.com/pgbouncer/pgbouncer/pull/845.

[28] PostgreSQL Message Flow: Extended Query. https://www.postgresql.org/docs/13/protocol-flow.html#PROTOCOL-FLOW-EXT-QUERY.

[29] PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[30] DigitalOcean: How to Manage Connection Pools for PostgreSQL Database Clusters. https://docs.digitalocean.com/products/databases/postgresql/how-to/manage-connection-pools/.

[31] PostgreSQL Limits. https://www.postgresql.org/docs/current/limits.html.

[32] PostgreSQL: Message Flow: Start-up. https://www.postgresql.org/docs/13/protocol-flow.htmlid-1.10.5.7.3.

[33] PostgreSQL: Resource Consumption: temp_buffers. https://www.postgresql.org/docs/current/runtime-config-resource.html#GUC-TEMP-BUFFERS.

[34] PostgreSQL: Resource Consumption: work_mem. https://www.postgresql.org/docs/current/runtime-config-resource.html#GUC-WORK-MEM.

[35] ScyllaDB. https://www.scylladb.com/.

[36] CMU Database Group: ScyllaDB: No-Compromise Performance (Avi Kivity). https://www.youtube.com/watch?t=2586&v=0S6i9BmuF8U.

[37] SELTZER, G. BPF Map Concurrency Techniques . https://www.grant.pizza/blog/bpf-concurrency/.

[38] Sheng, S., Puyang, H., Huang, Q., Tang, L., and Lee, P. P. C. FarReach: Write-back caching in programmable switches. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 571–584.

[39] sockperf: Network Benchmarking Utility. https://github.com/Mellanox/sockperf.

[40] Linux kernel patch: net, sched: add clsact qdisc. https://lwn.net/Articles/671458/.

[41] Tu, W., Wei, Y.-H., Antichi, G., and Pfaff, B. revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (New York, NY, USA, 2021), SIGCOMM '21, Association for Computing Machinery, p. 245—257.

[42] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C., and Vieira, L. F. M. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv. 53*, 1 (feb 2020).

[43] Wang, F., Zhao, G., Zhang, Q., Xu, H., Yue, W., and Xie, L. Oxdp: Offloading xdp to smartnic for accelerating packet processing. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)* (2023), pp. 754–761.

[44] Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., and Cidon, A. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 375–393.

[45] Zhou, Y., Wang, Z., Dharanipragada, S., and Yu, M. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 1391–1407.

[46] Zhou, Y., Xiang, X., Kiley, M., Dharanipragada, S., and Yu, M. DINT: Fast In-Kernel distributed transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 401–417.