The Bi-Channel Networking Paradigm for Database Systems in the Cloud

Georg Kreuzmayr*
georg@tigerbeetle.com
TigerBeetle

Muhammad El-Hindi
muhammad.el-hindi@tum.de
Technische Universität München

Benjamin Wagner
benjamin.wagner@firebolt.io
Firebolt Analytics

Tobias Ziegler*
tobias@tigerbeetle.com
TigerBeetle

Viktor Leis
leis@in.tum.de
Technische Universität München

Abstract

When network links were slow, cloud and distributed database systems could rely on generic kernel abstractions and treat network communication as a black box. With today's fast cloud networks, this approach breaks down: database performance becomes limited by the CPU overhead of the kernel TCP stack. Replacing TCP with user-space UDP can reduce this overhead, but it requires reimplementing essential guarantees, such as reliability and ordering. To solve this conundrum, database systems should no longer treat networking as a black box but co-design it with database operations. We propose the bi-channel paradigm for database systems, which separates communication into two channels: A high-performance data path for latency- and bandwidth-sensitive operations, and a reliable control path for coordination and recovery. We implement the paradigm by combining user-space UDP and kernel-based TCP, though other stack combinations are possible. This design exploits modern NIC capabilities while preserving TCP's reliability. We demonstrate the paradigm's efficiency and simplicity in two representative settings: a distributed shuffle saturating 200 Gbit/s with three CPU cores, and a replicated key-value store processing millions of messages per second.

Keywords

Distributed Database Systems

1 Introduction

Cloud database systems. Networking is a major performance concern in cloud database systems. Online transaction processing (OLTP) systems, such as AWS Aurora [64], require low, predictable network latency to achieve low transaction latency. Online analytical processing (OLAP) systems such as BigQuery [41], Snowflake [11], and Firebolt [48] rely on parallel execution across many nodes to process large datasets efficiently [11, 41].

Network hardware is fast. Historically, network hardware constrained the performance of distributed data systems [5]. For example, network bandwidth limited the throughput of data-intensive operators such as distributed joins [50]. However, this has changed with recent advances in network technology: cloud Ethernet networks now offer high bandwidth at low cost. As Figure 1 (top) shows, in 2013 the fastest network interface provided 10 Gbps; today, instances such as c7gn provide 200 Gbps, and EC2 instances with 600 Gbps have recently been introduced [72].

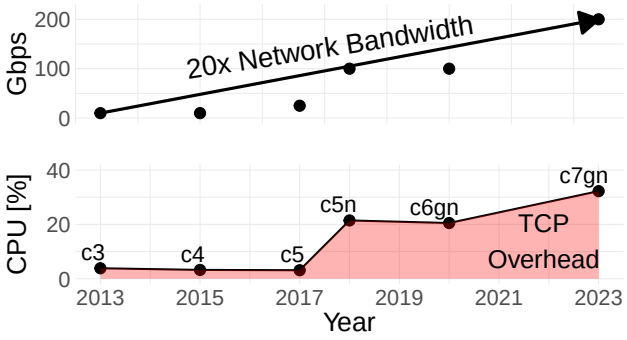


Figure 1: AWS Ethernet bandwidth over the last decade (top). CPU utilization of the kernel TCP stack while saturating full-duplex bandwidth (bottom). Measurements use io_uring in poll mode with 32 TCP connections and a 256 KiB buffer per connection.

Networking clogs the CPU. While network bandwidth has increased rapidly, CPU performance has improved much more slowly. Figure 1 (bottom) shows the CPU utilization required to saturate full-duplex bandwidth on AWS EC2 instances over time. In 2013, saturating 10 Gbps on a c3 instance required about 3% CPU. In contrast, saturating 200 Gbps on a c7gn instance requires over 30% CPU (20 of 64 cores).

TCP is inefficient. A major contributor to this CPU cost is the kernel TCP stack, which most database systems use for inter-node communication. TCP provides congestion control, reliability, in-order delivery, segmentation, and stream semantics, but these features impose substantial overhead at high bandwidths (Figure 1). For high-performance OLTP systems, a recent experimental study [76] showed that the CPU overhead of the network stack had become the "high pole in the tent", limiting overall transactional throughput. The reality is even more severe: modern network interface cards (NICs) can handle message rates far beyond what the kernel TCP stack allows [22]. For example, the NIC in a c7gn instance can reach roughly 30 M messages/s, which corresponds to about 5,000 CPU cycles/message on a 64-core 2.6 GHz CPU ($2.6 \times 10^9 \cdot 64/(30 \times 10^6)$). However, Figure 2 shows that kernel TCP consumes roughly 2× this budget, leaving the NIC underutilized.

Custom UDP-based protocols are complex. Compared to TCP, UDP can be more CPU efficient, and, in principle, database systems could build their networking entirely on top of UDP. However, doing so requires re-implementing key transport features. For example, data shuffling for a join needs reliability, which is non-trivial to achieve efficiently on UDP. Similarly, client connection handling requires congestion control mechanisms to

*Work done while at Technische Universität München

EDBT '27, Lille (France)

10.48786/edbt.2027.01

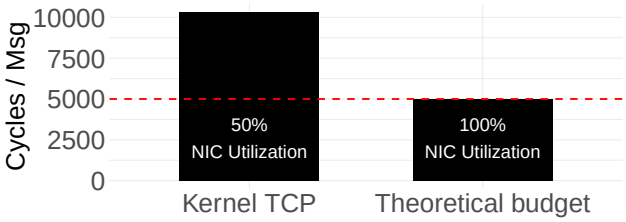


Figure 2: Kernel TCP requires ≈ 2× more cycles/message to fully utilize the NIC (measured with perf while running sockperf [46] + jumbo frames). A budget of ≈ 5k cycles/message is implied by a NIC with 30 M messages/s. The kernel stack is CPU-bound, achieving only ≈ 50% NIC utilization.

fairly share bandwidth among all clients. Thus, TCP is costly, but rebuilding its semantics inside the DBMS is also expensive and error-prone.

The bi-channel communication paradigm. We observe that database operators do not always require the full TCP feature set. We therefore introduce the *bi-channel paradigm*, an architectural pattern for distributed database systems that decouples the control and data paths at the networking layer. Instead of committing to a single transport protocol, the system coordinates two channels: (i) a high-throughput, unordered data channel for bulk transfers and (ii) a reliable, ordered control channel for coordination, retransmission, and failure handling (Figure 3). This design shifts part of the transport boundary into the DBMS, allowing each operator (e.g., shuffle or replication) to choose the channel that matches its required guarantees. As a result, systems can better exploit modern NICs without re-implementing TCP semantics in user space. Analogous to how multi-core and SIMD reshaped CPU-bound processing, the bi-channel paradigm reshapes how distributed database systems use high-speed networks.

Contributions. We make three contributions:

(1) We propose the *bi-channel paradigm*, a design principle for distributed database systems that separates control and data paths at the transport layer. The paradigm generalizes how a DBMS can combine heterogeneous network stacks to match operator requirements.
(2) We derive *vendor-agnostic design guidelines* for high-performance user-space networking in cloud environments. Using AWS as a case study, we analyze constraints such as per-flow bandwidth limits and NIC multi-queue parallelism and distill techniques that generalize across clouds.
(3) We demonstrate the paradigm in two database use cases: (i) a distributed shuffle operator that saturates a 200 Gbps link using three CPU cores and (ii) a replicated key-value store that achieves low latency at high message rates.

Outline. The paper is organized as follows. Section 2 motivates the problem in more depth and reviews limitations of current networking stacks and the role of kernel bypass. Section 3 presents the bi-channel paradigm and its design principles. Section 4 uses AWS as a representative environment to show how to implement and tune the approach, highlighting pitfalls and distilling practical guidelines. Section 5 evaluates using the bi-channel paradigm in a high-bandwidth shuffle operator, and Section 6 evaluates it in a low-latency replicated key-value store. We discuss related work in Section 7 and conclude in Section 8.

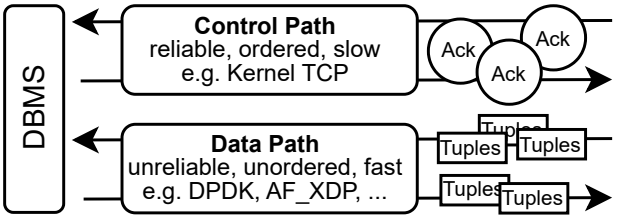


Figure 3: Bi-channel networking uses two coordinated transport paths: a high-throughput, unordered data channel and a reliable, ordered control channel.

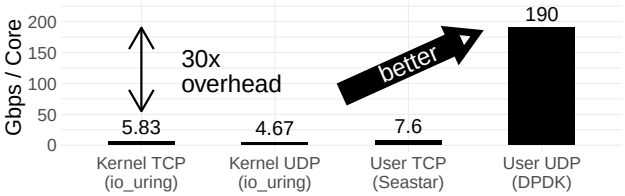


Figure 4: Full-duplex throughput per core between two c7gn.16xlarge instances for kernel- and user-space TCP and UDP, using 256 KiB messages.

2 Motivation & Background

Distributed database systems depend on efficient network communication. Although prior work has explored new hardware technologies such as RDMA, public-cloud networking is still predominantly Ethernet-based. Consequently, TCP and UDP remain the only transport protocols that are universally available across cloud platforms.

Kernel-space TCP. TCP provides reliable, in-order delivery over an unreliable network. The kernel handles loss recovery, reordering, segmentation, and flow/congestion control. Database systems typically use kernel TCP for three reasons. First, when network bandwidth was limited, TCP overhead was rarely a primary performance concern. Second, TCP offers strong end-to-end guarantees behind a stable interface, which simplifies DBMS implementations. Third, kernel TCP is widely deployed and operationally mature.

Today, on high-bandwidth links, enforcing reliability, ordering, and stream semantics can consume a substantial fraction of CPU capacity. Our benchmarks between two c7gn.16xlarge instances (each with a 200 Gbps NIC), using `io_uring` in poll mode [23], quantify this cost. As Figure 4 shows, kernel TCP reaches 150 Gbps per direction while consuming 26 CPU cores (over 40% of the instance's total CPU resources), i.e., an average of 5.8 Gbps per core. As network bandwidth continues to increase (e.g., 600 Gbps today and higher rates expected), this imbalance is likely to worsen.

Kernel-space UDP. UDP is message-oriented and does not provide reliability or in-order delivery. Given its simpler semantics, one might expect kernel UDP to be more efficient than kernel TCP. However, Figure 4 shows that kernel UDP achieves slightly lower throughput in our setup. The reason is that TCP benefits from throughput-oriented kernel optimizations (e.g., packet batching via Nagle's algorithm), whereas the kernel UDP path is often tuned for low latency and minimal buffering.

User-space TCP. To reduce kernel overheads, researchers have developed user-space TCP stacks, where the transport protocol runs entirely in user space. Examples include mTCP [24], IX [7],

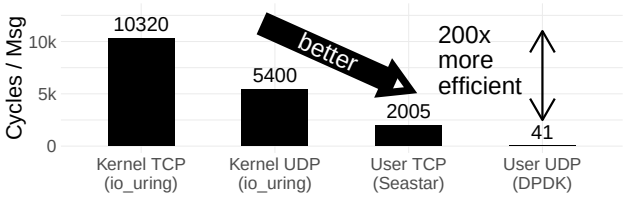


Figure 5: CPU cycles per message between two c7gn.16xlarge instances for kernel- and user-space TCP and UDP, using 64-byte messages.

TAS [28], F-Stack [61], LUNA [78], and Seastar [53]. We use Seastar, the stack used by ScyllaDB [54], in our benchmarks. Seastar employs a shard-per-core architecture to reduce context switching and lock contention. Nevertheless, Figure 4 shows only a ∼30% throughput improvement over the `io_uring`-based kernel baseline in our configuration. Prior work (e.g., IO-TCP [30] and LUNA [78]) reports similar overhead trends.

User-space UDP. User-space network stacks are often built on the Data Plane Development Kit (DPDK), a collection of libraries and drivers designed for high-performance packet processing in user space. Unlike TCP, UDP can be implemented directly on DPDK without complex connection state. In our experiments, a DPDK-based UDP stack reaches 190 Gbps using only **one** CPU core. This means DPDK's raw packet processing capability is more than 25 times faster than the kernel- and user-space TCP network stacks. Message rate is also critical for key-value stores and OLTP workloads. Figure 5 reports cycles per message for 64-byte messages: user-space UDP reduces CPU cost by orders of magnitude relative to kernel TCP and also substantially outperforms user-space TCP.

However, user-space UDP also comes with practical limitations. For example, sharing a NIC across multiple processes is challenging without additional multiplexing support. Moreover, applications that need reliability, ordering, or segmentation must implement these mechanisms above UDP.

Summary. Existing options leave an unsatisfying trade-off between strong semantics and efficiency. Table 1 summarizes the qualitative characteristics of the four stacks.

Table 1: Qualitative comparison of TCP and UDP stacks

	Kernel-space		User-space	
	TCP	**UDP**	**TCP**	**UDP**
ordering	✓	X	✓	X
reliability	✓	X	✓	X
abstraction	stream	message	stream	message
CPU cost	high	high	high	low
impl. effort	–	–	high	low

Kernel TCP provides reliability, in-order delivery, and segmentation, but it can be CPU-intensive and may enforce stronger guarantees than many database operators require. In particular, reliable in-order delivery can trigger head-of-line blocking [52]: loss or delay of an early packet stalls delivery of later packets even when the application could process messages out of order. This increases tail latency and becomes more pronounced at high data rates [55]. While research proposals for alternative transports exist [47], their cloud availability and adoption remain unclear. Conversely, user-space UDP achieves much higher efficiency but lacks reliability and requires applications to implement features such as segmentation for messages larger than an MTU.

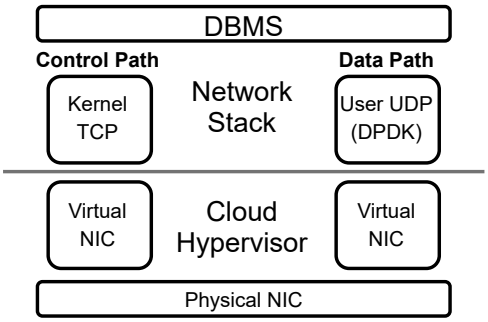


Figure 6: Example bi-channel network stack for a cloud DBMS.

However, we observe that the transport guarantees required by database operators often fall between these extremes. For example, a join/shuffle operation needs eventual delivery of all tuples, but not in-order delivery. This motivates our main research question: can we combine TCP-like guarantees with the efficiency of user-space UDP, and tailor transport semantics to the needs of individual database operators?

3 The Bi-Channel Paradigm for Database Systems

To leverage the strengths of different network stacks, we introduce the *bi-channel paradigm*, which enables distributed systems to benefit from both high-performance user-space UDP and the robustness of kernel-space TCP.

Separating control and data communication. Database networking naturally decomposes into control and data communication. Control messages (e.g., transaction coordination and query planning) typically require reliability and in-order delivery, whereas data transfers (e.g., tuple shuffling) primarily benefit from low CPU overhead and high throughput. As shown in Figure 6, a cloud DBMS can therefore use kernel TCP for the control channel and a DPDK-based user-space UDP stack [35] for the data channel.

Kernel TCP control channel. DBMSs exchange control messages across workloads and operators. In OLTP, nodes communicate to execute distributed commit protocols such as two-phase commit [43]. In OLAP, control traffic coordinates data exchange and scheduling decisions (e.g., partition assignment and load balancing). Because control traffic requires reliability and ordering guarantees, kernel-based TCP remains the most suitable choice for the control path.

User-space UDP data channel. The data channel targets performance-critical transfers. Its requirements vary by workload: analytical shuffles emphasize high throughput (Section 5), while replication and log shipping in transactional systems emphasize high message rates and low latency (Section 6). User-space UDP with DPDK provides the packet I/O efficiency needed for these hot paths.

DBMS integration. Realizing the bi-channel benefits requires co-design between the network path and DBMS components (e.g., physical operators). This is aligned with a long-standing database practice: bypassing generic OS abstractions on the critical path

Table 2: Packet loss for full-bandwidth UDP ping-pong communication between two `c7gn.16xlarge` instances. In total, 150 million packets were transmitted.

Run	0	1	2	3	4
Lost packets	0	0	0	19	0

when they impose unnecessary overhead. For example, many DBMSs implement custom buffer managers instead of relying on the OS page cache [33]. Analogously, exposing transport choices to operators can significantly improve networking efficiency. At the same time, the paradigm avoids an all-or-nothing shift to user-space networking: the kernel control channel provides mature reliability and ordering where needed, reducing the need to re-implement full TCP semantics in the DBMS.

Leveraging virtualized cloud NICs. A practical obstacle to combining kernel and user-space stacks on the same interface is that frameworks such as DPDK typically require exclusive access to a NIC. All major cloud providers mitigate this constraint by allowing multiple virtualized network interfaces (vNICs) to be attached to a single instance with minimal effort [1]. This enables the architecture shown in Figure 6, where two vNICs are used: one is managed by the kernel (control channel) and the other is bound to DPDK in user space (data channel).

Reliability of cloud Ethernet networks. The bi-channel approach relies on the assumption that packet loss is low enough that recovery traffic does not dominate the data channel. To assess this in practice, we measured packet loss in an AWS data center using a full-duplex UDP ping-pong benchmark between two c7gn.16xlarge instances in the same data center. We ran the benchmark for one minute at full bandwidth and recorded sender-side loss with a transmission depth of 1024. As shown in Table 2, four out of five runs delivered all packets successfully at an average bandwidth of 190 Gbps. In the remaining run, 19 packets were lost out of 150 million, corresponding to an overall loss rate of 1.3×10^{-7} (0.000013%). In an additional one-hour run, we observed a single lost packet. Consistent with these measurements, Google Cloud's performance reporting indicates low packet loss rates (below 0.05%) across regions [19]. Together, these results suggest that cloud Ethernet can support a hot user-space UDP data channel in practice, with infrequent loss recovery.

4 Case Study: Bi-Channel Paradigm in the Cloud (AWS Example)

The bi-channel paradigm is a general architectural framework, independent of any specific hardware or cloud vendor. However, the concrete implementation effort and tuning parameters may differ because each platform exposes different network abstractions and performance limits. Among public clouds, these limits vary substantially: for instance, Google Cloud primarily restricts external egress bandwidth [18], while AWS enforces per-flow limits even for intra-VPC traffic [2]. Therefore, while the bi-channel principles remain unchanged, realizing a high-performance data path requires adapting to the specific flow-control and queueing semantics of the underlying infrastructure.

AWS as a stress-test environment. We use AWS as a representative case study because it imposes the most restrictive intra-cloud networking constraints. These constraints make it an ideal stress test for the bi-channel design, which must remain efficient even under stringent per-flow and virtualization limits. The lessons learned from AWS extend to other clouds: the same structured approach and guidelines discussed below can be applied based on the exposed constraints in other clouds. For example, our experiments use the Elastic Network Adapter (ENA) on `c7gn.16xlarge` instances, but the derived principles are portable across NIC implementations supporting parallel transmit/receive queues, RSS-based load distribution, and user-space packet I/O. We next present a mental model of cloud NICs and derive guidelines for building robust, high-throughput data paths using user-space UDP. All experiments were conducted on Ubuntu 24.04 with DPDK 23.11.2.

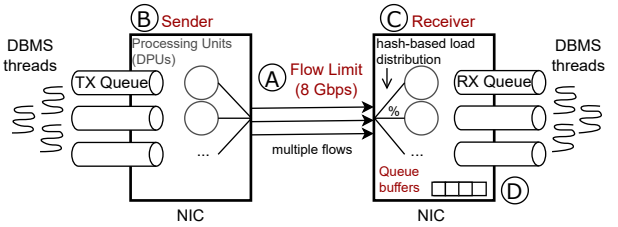


Figure 7: Mental model for cloud NICs: DBMS threads can directly interact with the NIC's hardware queues and use zero-copy communication. However, several pitfalls (A-D) exist that developers must be aware of.

4.1 Mental Model for Cloud NICs

The way of the packet. Leveraging user-space frameworks, such as DPDK, with the bi-channel paradigm enables applications to operate very close to the metal – that is, directly at the NIC. As shown in Figure 7, DBMS threads can directly interact with the NIC's hardware queues. In particular, when sending packets, an application places them in the transmit (TX) queue (see Figure 7 left). Since NICs are highly parallel devices, they have multiple data processing units (DPUs) (circles) that consume packets from the TX queue and send them over the cloud network. On the receiving NIC (Figure 7, right), incoming packets are routed to different DPUs for parallel processing (using a hash-based load distribution). The DPUs copy the packets via Direct Memory Access (DMA) into receive or queue buffers, after which the DPU places a completion event into the receive (RX) queue. Application threads at the receiver poll the RX queues and directly read packets from these buffers. This mechanism is known as zero-copy communication, where packets are copied only once, directly from the NIC into user-accessible receive buffers in RAM, and are never copied again within RAM.

Potential networking bottlenecks. Despite the conceptually simple design shown in Figure 7, many pitfalls exist when trying to achieve robust, high-performance networking. For example, cloud networks have complex topologies [59] and multiple tenants share their underlying infrastructure, forcing cloud providers to impose so-called flow limits on a communication path (Ⓐ), i.e., a path between two endpoints. Moreover, slight misconfigurations can lead to bad performance on the sender or receiver side (Ⓑ & Ⓒ), particularly regarding queue buffers Ⓓ that are the backing memory for entries in the hardware queues.

In the following, we detail pitfalls and derive guidelines for achieving high-performance and robust networking on the data path, when optimizing either for bandwidth (Section 4.2) or packet rate and latency (Section 4.3).

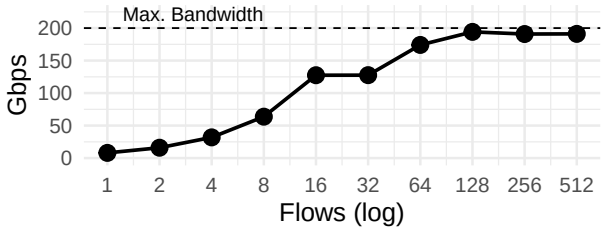
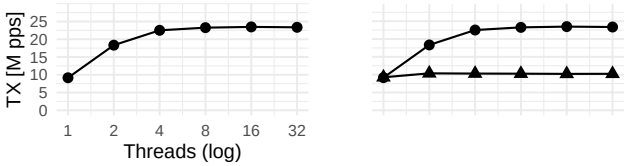


Figure 8: Effect of an increasing number of UDP port combinations (flows) on bandwidth with 8192-byte packet size.



(a) Sender side **(b) Receiver side**

Figure 9: Thread scalability of sender and receiver side with 64-byte packets between two c7gn.16xlarge instances.

4.2 Optimizing for Bandwidth

The bi-channel paradigm's data path represents the hot path for database operations. For example, it is important to optimize this hot path for high-bandwidth data processing in the context of OLAP databases. In the following sections, we highlight pitfalls and demonstrate how to achieve high bandwidth using user-space networking in the cloud.

Single-flow bandwidth limits. Although c7gn.16xlarge instances in AWS offer a 200 Gbps network interface, and using DPDK out-of-the-box in the data path to send data between two instances will surprisingly hit a bottleneck at around 8 Gbps. This is because AWS enforces a single-flow bandwidth limit restricting the throughput of TCP and UDP connections [2]. Within one cluster placement group, which we used for our experiments, the limit is 10 Gbps. Our empirical measurements showed that the throughput is limited to 8 Gbps when using a single UDP port combination. To achieve higher throughput, it is required to use multiple paths between the communicating endpoints by varying the source and destination ports in the UDP headers[1].

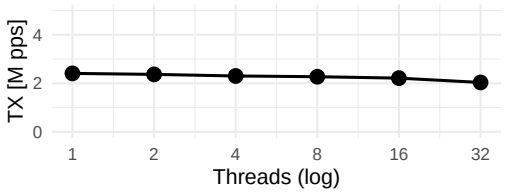
Port scaling. Figure 8 shows the throughput of user-space UDP with an 8192-byte packet size and increasing port combinations, i.e., flows. We increased the number of port combinations by configuring the sender and receiver threads to use varying source and destination ports. Initially, the throughput increases linearly to 16 Gbps using two ports and to 32 Gbps using four ports. After reaching 130 Gbps with 16 ports, the throughput starts to stagnate. Only when further increasing the number of ports to 128, we get close to the maximum throughput of EC2's 'c7gn.16xlarge instance type.

Guideline 1: Beware of bandwidth flow limits. Use multiple flows by varying source and destination ports to increase throughput.

4.3 Maximizing Packet Rate

Single-threaded packet rate. In the previous experiment, we used large packet sizes of 8192 bytes. Hence, data transmission was bottlenecked by the single-flow bandwidth limit, resulting in a packet rate of around 100'000 packets per second. Repeating the same benchmark with small 64-byte packets stalls at around 2.4 M packets per second. This time, throughput is well under 8 Gbps, meaning that the limiting factor is not the single-flow bandwidth limit. To find the root cause of the bottleneck, we analyze whether CPU performance is the limiting factor.

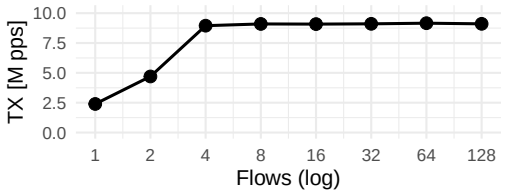
Multi-threaded packet rate. Instead of a single thread, we now use multiple threads to send UDP packets and measure the packet rate for half-duplex communication between two servers. We measure the scalability of the sender's packet rate with the number of threads for a single flow (i.e., UDP port combination):



The results show that simply increasing the thread count while using a single flow achieves the same packet rate as with a single thread (2.4 M packets per second). Hence, neither the flow limit nor the CPU performance is the limiting factor.

Parallel processing in cloud NICs. To understand the root cause of the packet-rate bottleneck, we need to return to our mental model of cloud NICs from Section 4.1. NICs are highly parallel devices that internally utilize multiple DPUs (circles in Figure 7), similar to threads, to send packets in parallel. Similar to sharding in databases, the parallelism is achieved by partitioning work across DPUs using a hash function based on the UDP headers (function $h()$ in Figure 7). Despite scaling the number of CPU threads in the earlier experiment, all packets were sent as part of a single flow, i.e., using the same source and destination ports, and thus processed by a single DPU. This deterministic partitioning is important when processing dependent packets, such as in TCP streams, but it can result in load imbalances, which is the root cause of the stagnating packet rate at 2.4 M packets.

Sender scaling. To overcome this limitation, we apply the port-scaling optimization to use multiple flows and distribute the work across the DPUs. This figure shows the scalability of the sender's packet rate with the number of UDP ports (flows):



It demonstrates that using multiple UDP port combinations improves the packet rate to around 9 M packets per second with a single thread. This confirms that the previously observed rate of 2.4 M was not limited by CPU performance. In practice, it is advisable to perform micro-benchmarks to determine how many port combinations are required to maximize the packet rate.

Guideline 2: Beware of packet rate flow limits. Use multiple flows to achieve high packet rates.

[1] In AWS, a single-flow is considered a unique 5-tuple TCP or UDP flow defined by source IP and port, destination IP and port, and the next protocol [2].

(a) In-flight limit configuration. (b) RX descriptor configurations.

Figure 10: Packet loss for different in-flight and RX queue configurations.

Thread parallelism and port scaling. Having determined the single-threaded packet rate bottleneck, we now attempt to increase the overall packet rate by employing thread parallelism again. We use 64 different UDP port combinations while increasing the number of threads used on the sender side. As shown in Figure 9a, this configuration can scale the transmitted packets per second up to around 25 M. However, Figure 9b (triangle line) shows that with the same configuration (i.e., using 64 different flows), the receiver side does not scale in the same way.

Receiver scaling. Examining the receiver side closely, reveals a similar problem as on the sender side: Packet processing on the receiver is skewed, and not all DPUs are processing packets. The triangle configuration in Figure 9b used UDP port combinations that caused all packets to be distributed to the same DPU, limiting the receiver side's packet rate. This was configured deliberately and shows the worst-case effect of skewed packet distribution.

Queue targeting. A NIC typically exposes multiple RX queues. Polling all RX queues from every thread, however, would be inefficient. Instead, it is preferable to determine the RX queue to which each packet is steered, so that the corresponding thread can process it directly. This capability enables efficient implementations, such as sharded key-value stores, in which requests are routed to the worker thread responsible for the target shard. The mechanism that enables such packet steering is commonly known as receive side scaling (RSS) and is widely supported by modern physical and virtual NICs. Because RSS is deterministic, a given packet is always assigned to the same RX queue. AWS, Azure, and GCP document the hash functions used by their RSS implementations [3, 12, 20]. This allows the sender to precompute the receiving thread. We refer to this technique as *Queue Targeting*. It enables applications to control packet routing in user space and balance load across cloud instances.

The effect of queue targeting. The circle configuration in Figure 9b uses 64 different UDP port combinations per thread while leveraging our queue targeting technique. This way, we achieve scalability not only on the sender side but also on the receiver side.

Guideline 3: Leverage queue targeting to control packet routing from the application level and avoid RSS bottlenecks.

4.4 Minimizing Packet Loss

As with performance, network reliability for user-space UDP in the cloud does not come out of the box. Yet, reliability is critical for applications to benefit from a high-performance data path.

Packet loss in low load scenarios. Under low load, applications rarely observes packet loss when using user-space UDP. For example, in the experiment for Table 2, we limited the number of in-flight ping-pong packets to only 1024.

Packet loss in high load scenarios. However, the situation is different under high load when applications rely on user-space UDP. To illustrate the problem, Figure 10a shows the packet loss for single-threaded communication using 64-byte packets for two settings. In the "No-Limit" (triangle line) setting, we do not limit the number of outstanding requests, which increases receiver load over time. Due to the lack of a control flow mechanism in user-space UDP, the packet loss rate begins to skyrocket.

Application-side load-balancing. The bi-channel paradigm addresses this problem by enabling application-specific control flow. This is illustrated with the "1024 In-Flight" setting in Figure 10a. In this setting, the receiving server can use the kernel-based control path to report to the sender which packets arrived successfully. This way, the sender can avoid overloading the receiver by sending more than 1024 packets. As shown in Figure 10a (circle line), this application-side load balancing mechanism avoids increasing packet loss even when the receiver is busy.

Queue buffers. A key question when implementing application-side load-balancing is how many in-flight requests to allow. In other words, how to set the in-flight packet rate for an application or workload? To answer this question, we need to understand the inner workings of an NIC in depth. In particular, developers must be aware of the number of queue buffers (Ⓓ in Figure 7). Queue buffers serve as the backing store for incoming network requests. That is, when a DPU places a packet into the RX queue for the application to consume, it must allocate a queue buffer. However, when the queue buffers are exhausted, the NIC will simply drop the incoming UDP packet. The number of buffers is limited and configurable during device startup. For example, the c7gn.16xlarge instance type supports between 128 and 8192 queue buffers.

Choosing an in-flight packet rate limit. A common assumption is that if the number of in-flight packets is kept below the number of RX queue buffers, the NIC should have enough capacity to receive all packets without drops. However, our experiments show that this assumption does not always hold and that the in-flight limit must be chosen with care.

Figure 10b shows the packet loss for a varying number of RX queue buffers and two different in-flight limits. To simulate a busy receiver, we perform 100 ns of work in a hot loop per received packet, as described in Figure 10a. When the in-flight limit is higher than the number of RX queue buffers, the RX queue (Figure 7 right) runs out of buffers, and the packet loss rate goes up.

Even when the in-flight limit equals the number of RX queue buffers, we observe a packet loss rate of around 10%. This happens because not all RX descriptors are available for receiving packets at any given time. With a limit of 256 packets, the loss is almost eliminated once the RX queue has 512 or more buffers. With 1024 in-flight packets, packet loss disappears with 2048 buffers.

Guideline 4: Use application-side load-balancing via the control path to limit the number of in-flight packets. As a rule of thumb, choose the in-flight packet rate relative to the RX queue buffers.

While our case study focuses on AWS, the mental model and best practices derived here, such as Guideline 4, capture vendor-independent principles that can be applied to other cloud platforms. We next show how the bi-channel paradigm integrates with database systems and can be applied in concrete database use cases.

5 Use Case 1: Efficient Data Shuffling in Distributed Joins

This section demonstrates how the bi-channel paradigm and low-level NIC optimizations presented previously are applied in database-specific use cases. We focus specifically on distributed joins due to their intensive network communication requirements. Modern analytical query engines process queries involving petabytes of data [63, 65], distributing and partitioning data across multiple nodes. During a join, data is frequently repartitioned – commonly referred to as "shuffling" – across the network based on join keys. This shuffle operator efficiently repartitions data across nodes via the network, making it a good fit for illustrating the benefits of the bi-channel paradigm.

5.1 Anatomy

Shuffle operator characteristics. Implementations of the shuffle operator differ widely in both design and execution. Systems such as Spark[73], or BigQuery [41] rely on distributed storage layer for cross-node data transfers . Recently, systems without dedicated shuffle services have become more prevalent, often employing direct point-to-point communication. In these modern architectures, each node can directly transfer data to every other node [48], i.e., all-to-all. Joins and aggregations typically employ hash-based partitioning functions to identify the destination node for each data item. In this case study, we focus specifically on a point-to-point hash-based shuffle implementation.

Communication characteristics. Before applying the bi-channel paradigm, it is important to understand the shuffle operator's communication characteristics. We identify three main traits: First, the primary goal of any shuffle implementation is to achieve high bandwidth, as data transfer time heavily influences overall query execution time. Second, since we focus on hash-based shuffle operations, the order of tuple arrival is not important. Third, reliability is crucial – all data must eventually be processed without loss. Mapping these characteristics into the bi-channel paradigm, we see that the first two align well with the user-space UDP data path. As demonstrated in Section 2, a single core reaches nearly 200 Gbps on the data path, making it an optimal choice, especially given the shuffle's insensitivity to tuple ordering. The third attribute – reliability – necessitates a reliable communication mechanism to guarantee complete data delivery. This divergence in requirements is precisely what the bi-channel paradigm addresses, making it a natural and effective fit.

5.2 Bi-Channel Shuffle

This section demonstrates how to use the bi-channel paradigm in a shuffle operator. Since high throughput is crucial for the shuffle, we first leverage the fast user-space channel to meet this performance objective. We then discuss the qualitative advantages of the secondary, slower channel using kernel-space TCP.

Implementation overview. Figure 11 provides a high-level overview of our solution. It depicts two example nodes that both send and receive data. The input relation is processed using morsel-driven parallelism [34]. Worker threads partition the data into local output buffers for local processing and into outgoing packets for remote nodes ①. Remote communication is handled via the fast path, using UDP packets that are filled incrementally with tuples and sent once full ②. Because each worker also receives data asynchronously from remote nodes, it periodically polls the receive queues for incoming packets stored in receive buffers. When data arrives, we place pointers to the receive buffers directly into the local output buffer, thereby avoiding unnecessary data copying ③.

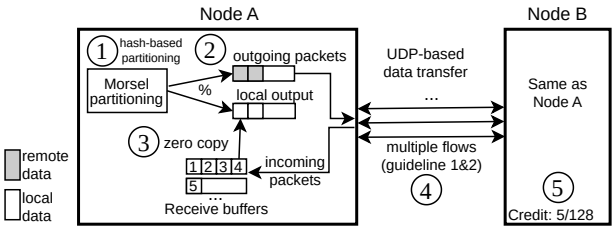


Figure 11: The bi-channel paradigm applied for data shuffling: We use morsel-driven parallelism and a UDP-based data transfer with zero-copy and multiple flows.

Thread scalability. To ensure that our operator scales with the number of threads, avoiding the NIC from becoming a scalability bottleneck is important. To achieve this, we apply the principles outlined in the previous chapter: (1) avoiding the bandwidth and packet rate limitations by utilizing multiple parallel flows (guideline 1+2) and (2) carefully choosing sender and receiver ports to enable queue targeting (guideline 3) ④. We encapsulate this port-selection logic in a dedicated component, PortPicker, which is used during cluster provisioning to establish the necessary connections (flows). At system startup, we configure communication and allocate multiple flows per thread to ensure that single-flow limitations do not constrain system bandwidth. This approach means that if we use a single core, it uses multiple parallel flows, effectively multiplexing communication, eliminating the flow bottleneck, and fully parallelizing the NIC. Once these implications are clearly understood, satisfying them becomes straightforward, enabling our implementation to scale nearly perfectly, as demonstrated by the later experimental evaluation.

Receive-buffer management. One crucial aspect we have not yet addressed is the management of queue buffers (i.e., guideline 4). Exhausting queue buffers results in significant packet loss, making proper management essential for reliability. One straightforward solution is to have the remote node synchronously acknowledge each received UDP packet. However, limiting communication to only a single in-flight packet severely restricts bandwidth utilization. To achieve higher efficiency, we instead allocate multiple buffers per worker, enabling multiple packets to be in flight concurrently.

Credit-based coordination. We implement a credit-based mechanism [27, 32] to prevent buffer exhaustion at the receiver: the sender maintains a credit count, representing the available receive buffers at the remote node ⑤. This count is decremented each time a packet is sent and replenished upon receiving acknowledgments (credit returns) from the remote node. For example, if a remote worker has 128 receive buffers allocated, the initial credit for communication is set to 128. Each packet transmission reduces this credit by one, and each acknowledgment restores it accordingly. Since we have direct control over packet destinations via queue targeting (i.e., which specific thread each packet is delivered to), we can precisely track credit consumption and replenishment on a per-thread and node basis. For instance, thread 0 on one node only talks to its buddy thread 0 on the other node. Our PortPicker and the underlying awareness of the NIC architecture have enabled this per-thread communication pattern. The allocation and initial assignment of receive buffers (credits) are determined during system startup.

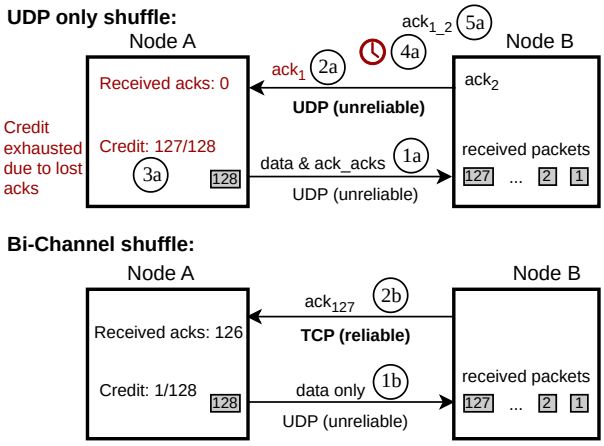


Figure 12: A pure UDP-based shuffle implementation (top) comes with a significant implementation complexity to ensure reliable data transmission (steps 1a-5a). The bi-channel approach (steps 1b-2b) combines the efficiency of UDP and the reliability of TCP (bottom).

What can go wrong? Consider Figure 12 (top), which illustrates nodes A and B communicating over UDP only. Node A transmits data ①a, while node B returns a credit acknowledgment ②a. If this acknowledgment is lost, the corresponding credit is not replenished. At best, this merely reduces bandwidth utilization by reducing the number of packets in flight. In the worst-case, repeated acknowledgment losses could lead to complete credit exhaustion ③a, causing node A to block indefinitely and halt progress. One potential solution is for node A to send an additional acknowledgment, confirming receipt of node B's credit acknowledgment. However, this "acknowledgment of acknowledgment" could also be lost, forcing node B to use a timeout ④a and periodically retransmit the original credit acknowledgment ⑤a. Unfortunately, such a timeout-based retransmission can cause issues: suppose the original acknowledgment was successfully received, but only the acknowledgment-of-acknowledgment was lost. In that case, retransmission might erroneously increment the credit twice, causing receiver buffers to underflow as credits exceed actual available buffers. To address this duplication issue, we might introduce sequence numbers to identify and eliminate redundant retransmissions. However, another complexity emerges: acknowledgment packets themselves consume receive buffers, requiring additional buffer management. Clearly, the complexity escalates, prompting an attentive reader to notice that we are effectively reimplementing fundamental TCP mechanisms, including flow control, buffer management, and sequence numbering. This observation raises the question of whether such a reimplementation from scratch is necessary.

Control path to the rescue. The solution provided by the bi-channel approach is straightforward: use the reliable TCP-based control channel to transmit credit acknowledgments. Similarly, we use the TCP-based control channel also to re-transmit any outgoing packets that were not acknowledged by the receiver. Because TCP inherently ensures reliability through established mechanisms such as flow control and retransmission, our implementation is significantly simplified. In summary, the fast channel manages the high-throughput data path ①b, while the control channel, using TCP, reliably handles credit acknowledgments and packet retransmissions ②b (cf. Figure 12 bottom).

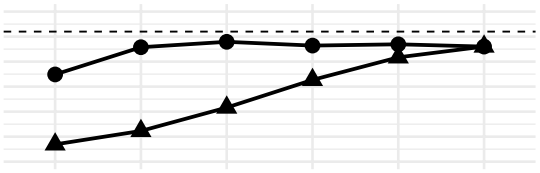


Figure 13: Shuffle operator benchmark results showing the impact of an increasing number of threads in the networking layer on tuple throughput. Threads are always fully-utilized. Each node shuffles 256 GiB.

5.3 Evaluation

In the following, we evaluate the performance of the shuffle operator using the bi-channel communication paradigm. Our experiments show that the bi-channel approach offers a significant efficiency advantage over the traditional TCP stack. Furthermore, we show that the bi-channel approach scales well with cluster size and achieves higher throughput than the kernel TCP stack for small table sizes.

Experimental setup. The shuffle is performed by scanning data from an in-memory table in columnar format with 16 columns, each 8 bytes wide. The data is partitioned by hashing one input column with a murmur hash function [4]. The result data is gathered in row-based format, simulating the gather phase of a hash join operation.

The kernel TCP implementation uses the io_uring interface with submission queue poll mode. For sending data, we use io_uring's zero-copy send interface [29] to avoid main memory bandwidth becoming a bottleneck. We use 256 TCP connections and a buffer size of 256 KiB, as this setup proved to be the most efficient. On submission, we set the *IOSQE_ASYNC* flag to indicate that the operation should be executed by a kernel worker thread. Our benchmark scales the kernel worker threads, while the networking layer uses a single thread for submitting tasks and processing completions.

We conduct the experiments on the c7gn.16xlarge instance type with 200 Gbps network bandwidth in AWS. The instances are running Ubuntu 24.04 with Linux kernel version 6.8.

Scalability. In the first benchmark, we evaluate the performance of the shuffle operator on a four-node cluster, shuffling 256 GiB of data. Since the main memory size of the c7gn.16xlarge instance is limited to 128 GiB, we iterate multiple times over a table with 64 GiB of data. The query processing layer uses 8 threads, producing enough remote tuples to saturate the network bandwidth.

Figure 13 shows the effect of increasing the number of fully-utilized threads in the networking layer on the throughput of the shuffle operator. The bi-channel approach achieves a tuple throughput of 175 M tuples per second with a single thread in the networking layer. The kernel TCP stack, on the other hand, achieves less than 50 M tuples per second with a single networking-layer thread.

Scaling the bi-channel approach to four threads maximizes the throughput at 240 M tuples per second, reaching over 90% of the theoretical maximum of 260 M tuples per second. The kernel TCP stack, on the other hand, only reaches 230 M tuples per second even when scaling up to 32 kernel worker threads. At this point, the TCP shuffle operator uses more than 50% of all CPU resources exclusively for network communication.

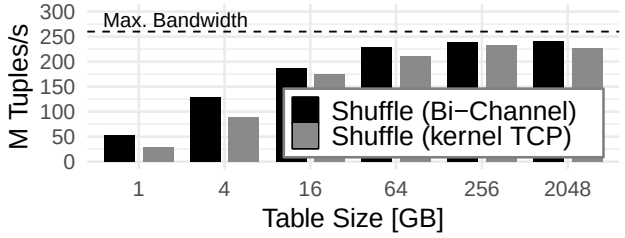


Figure 14: Shuffle operator benchmark results for increasing table size. Each node uses 4 threads (Bi-Channel) and 32 threads (kernel TCP).

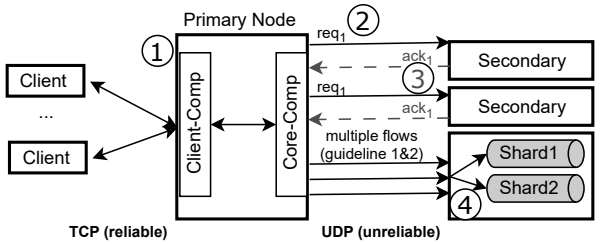


Figure 15: The bi-channel paradigm applied for KV-Store replication: Our approach combines a reliable TCP channel for client-facing communication with a low-latency UDP channel for data replication within a cluster.

Table size. In the second benchmark, we evaluate shuffle performance across a range of table sizes. Figure 14 compares the kernel-based shuffle implementation with the bi-channel variant for table sizes ranging from 1 GB to 2 TB.

For 1 GB tables, the bi-channel approach is twice as fast. This result shows that the bi-channel paradigm also benefits less data-intensive operations such as distributed aggregations. Whereas the kernel-based shuffle uses all available cores, the bi-channel implementation runs on only four. This advantage stems from TCP overheads, including congestion control, which requires shuffling more data to reach peak throughput. The performance gap begins to narrow at around 16 GB, where the larger data volume gives TCP enough time to ramp up and fully utilize the available bandwidth. Although small tables may appear less relevant given modern data volumes, analyses of the Snowset [65], Redset [63], and trace-derived workload synthesis [69] show that nearly 90% of tables are smaller than 1 GB, underscoring the importance of performance for small data sizes.

6 Use Case 2: Low-Latency Transactional Systems

While the shuffle use case focused on bandwidth, this section demonstrates how the bi-channel paradigm benefits latency-critical applications. We present a replicated key-value store that achieves very high message rates with predictable low latency.

6.1 Anatomy

Replication use case. Modern distributed systems increasingly depend on replicated key-value stores for state management, session handling, and real-time feature storage. Various replication approaches exist, including consensus algorithms such as Raft and Paxos. In this use case, we focus on quorum-based replication, where a client sends key-value updates to a designated primary. The primary replicates updates to multiple secondary nodes and waits for majority confirmation before responding to the client.
Communication characteristics. In this use case, the main goal is to minimize replication latency. While client communication generally remains beyond our direct control and commonly relies on TCP, internal replication presents an excellent opportunity for optimization. Given that the primary node communicates with multiple secondary nodes, achieving high message rates with low latency is critical. For instance, if clients generate one million requests per second and each request is replicated across five secondary nodes, the primary node must process a total of five million messages per second. As a result, the effective incoming message rate increases proportionally to the replication factor. Preserving the order of independent key updates is unnecessary; however, maintaining it for dependent updates is important.

6.2 Bi-Channel Replicated KV-Store

We now demonstrate how the bi-channel paradigm can be applied to implement a replicated key-value store. We highlight the flexibility of the bi-channel approach and emphasize that achieving optimal performance requires deliberate co-design of system requirements and the bi-channel implementation.
System overview. Figure 15 provides a high-level overview of the system architecture. The primary node consists of a client-facing and a core component that processes key-value updates and manages replication. Incoming requests ① are first handled by the client-facing component and then forwarded to the core component. The core component replicates these requests to secondary nodes ②, which apply the updates to their local state and acknowledge receipt back to the primary ③. Once the primary receives acknowledgments from a majority of secondaries, it applies the update to its local hash table and responds to the client's request.
Thread scalability. As illustrated in the figure, we adopt a sharded architecture ④, commonly used in key-value stores. In this design, dedicated worker threads handle client-facing requests and forward them to the appropriate shard-specific core workers via internal queues. This separation ensures that each core worker can process updates independently, minimizing contention. We carefully follow the guidelines to avoid the flow bandwidth limit and to choose appropriate ports to utilize the NIC, as discussed in Section 4. As in the shuffle use case, we leverage our knowledge of the specific worker handling each replication request. This approach allows us to route replication requests directly to the corresponding shard on secondary nodes, avoiding contention points, e.g., mutexes.
Receive Buffer Management. As in the shuffle use case, we employ a credit-based mechanism on the slow path to manage receive buffers. However, unlike shuffle, we decouple replication acknowledgments from credit-based flow control messages. Credits are sent asynchronously in batches over the control path, and we register enough receive buffers to sustain high throughput. Replication acknowledgments, by contrast, are handled directly on the hot path, as they are latency-critical: the primary must wait for a quorum of these acknowledgments before confirming completion to the client.
Low-Latency quorum acknowledgments on the data path. To support this design, we rely on user-space UDP to send acknowledgments, enabling us to meet the strict latency requirements of the hot path. Figure 16 shows a microbenchmark comparing kernel-based TCP with user-space UDP, highlighting the difference in 95th percentile latency at a fixed message rate. We

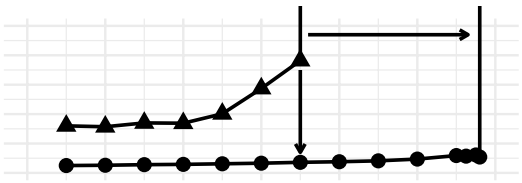


Figure 16: Impact of an increasing send rate on 95th percentile reply latency between two c7gn.16xlarge instances for Kernel TCP and User UDP with 64-byte messages.

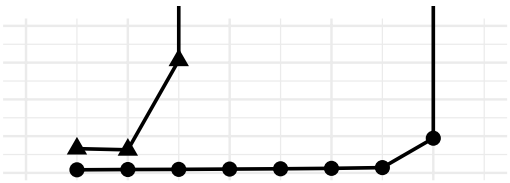


Figure 17: KV-Store replication benchmark with increasing update rate comparing the bi-channel approach to kernel TCP. The sender uses Poisson-distributed send timings.

use small 64-byte packets to minimize the latency overhead of transmission. User-space UDP consistently achieves predictable latencies below 100 microseconds up to the NIC's bi-directional limit of 12 million messages per second. In contrast, kernel TCP consistently exceeds 150 microseconds in latency and delivers 70% fewer messages per second at a peak with 10 × higher latency. These results demonstrate that relying on kernel TCP for acknowledgments (as in the shuffle) would severely degrade latency and throughput – an unacceptable trade-off for a latency-critical system. We implement a minimal, tailored subset of TCP functionality to address this, ensuring reliable delivery without unnecessary overhead. For instance, congestion control is not required, as the primary node already regulates the load on secondaries. We also simplify protocol semantics by leveraging domain-specific knowledge: version numbers used for recovery are reused as lightweight sequence numbers to ensure consistency across replicas. Packet loss is rare due to adequate buffer sizing and is handled via simple timeouts and retransmissions. Moreover, when data is replicated across four out of five nodes, retransmissions to the remaining node can be safely skipped thanks to redundancy.

Client communication on the control path. However, there are other opportunities for utilizing the slow path. Client communication differs fundamentally from the internal replication communication patterns. In a static environment, such as our primary and secondary node cluster, we can easily pre-allocate receiver buffers, precisely calculate credits, and maintain persistent connections. In contrast, clients typically connect intermittently and unpredictably, making it difficult to anticipate the number and frequency of connections. As a result, static rate-limiting approaches, which evenly distribute credits across cluster nodes, are inadequate for client communications. Instead, the primary node must dynamically allocate credits to clients based on the number of clients and their individual sending rates. This requirement closely resembles the flow control mechanisms built into the TCP protocol. Handling an arbitrary number of clients transmitting at varying rates introduces complexity beyond managing communications within a static node cluster. Therefore, for client interactions, we leverage the kernel-based TCP path to effectively address these dynamic communication challenges.

6.3 Evaluation

We next evaluate the performance of the KV-store using the bi-channel communication paradigm. Our experiments show that the bi-channel network stack achieves higher packet rates at lower latencies and CPU usage than the kernel TCP stack. The end-to-end benchmark shows that the bi-channel approach enables the KV-store to fully saturate the network interface.

Methodology & setup. We conduct the experiments in the same environment as in the previous use case. The kernel TCP implementation uses the io_uring interface. This time, compared to the shuffle benchmark, we avoid dispatching work to kernel worker threads by setting the defer task run option for improved latency. We scale CPU resources by increasing the number of threads handling io_uring submissions and completions. Each thread uses a separate io_uring instance.

We evaluate the performance of the KV-store replication process accelerated by the bi-channel paradigm. We compare performance with a KV-store implementation that uses the kernel TCP stack instead of the bi-channel network stack.

End-to-End latency measurement. The benchmark sends updates to the KV-store from a client at an increasing rate. After receiving the update at the KV-store, the KV-store replicates its state and replies to the client once the update is durably stored. We use small 64-byte updates to minimize the impact of transmission time on our latency measures. We measure the end-to-end latency at the client. This means that, for the KV-store with the bi-channel network stack, the total latency includes both the round-trip latency of TCP for client-server communication and the latency of the bi-channel network stack for replication.

Results. Figure 17 shows the p95 latency of the two key-value store implementations under increasing update rates. The bi-channel variant consistently achieves 100 microseconds lower p95 latency at low load than the kernel-based TCP version, and this gap widens as load increases. The kernel TCP stack reaches a maximum throughput of 1.5 million updates per second. In contrast, the bi-channel approach achieves up to 2.6 × higher throughput – approximately 3.9 million updates per second – while maintaining a p95 latency of around 250 microseconds.

At this peak rate, the primary node handles 24 million packets per second, saturating the NIC's packet-processing capacity. We conclude that the bi-channel approach enables the key-value store to fully utilize modern network hardware, significantly outperforming the kernel TCP-based alternative.

Latency distribution. Figure 18 presents the cumulative distribution function (CDF) of latencies at a load of 1 million updates per second for both bi-channel and kernel TCP. The y-axis represents cumulative probability, and the x-axis shows latency in microseconds. We observe that bi-channel exhibits significantly more stable latency compared to kernel TCP. For bi-channel, the difference between the minimum recorded latency and the 99th percentile latency is only 9 microseconds. In contrast, for kernel TCP, this gap is approximately 140 microseconds; even the median latency deviates by around 80 microseconds from the

Figure 18: Cumulative distribution function for KV-Store replication with 1 M updates/s comparing the tail latencies of bi-channel and kernel TCP approach.

minimum. These results indicate that bi-channel offers substantially better predictability and stability, making it more suitable for scenarios requiring strict Service Level Agreements (SLAs).

7 Related Work

Hardware-supported kernel bypass. Several studies have explored RDMA [8, 26] and EFA [55, 80], networking technologies that bypass the kernel data path and expose low-latency communication to user space. These solutions, however, require specialized hardware. EFA, for example, is proprietary and currently available only on AWS, while RDMA is not widely available in public clouds. In the context of database systems, extensive research has examined how to exploit such specialized networks. Systems such as FaRM [13, 14, 56], Herd [25], ScaleStore [79], and others [36, 40, 42, 67, 74] build on these approaches. Other work has targeted specific operations, including distributed joins [37], data-structure optimization [39, 66, 68, 71], replication [6, 15], and operator offloading [31].

User-space network stacks. Several systems move network processing into user space to reduce kernel overhead, including mTCP [24], IX [7], TAS [28], F-Stack [61], LUNA [78], and Seastar [53]. These stacks are attractive when applications require TCP-compatible semantics, but building a general-purpose, production-ready stack remains challenging. ScyllaDB [54] and Yellowbrick [10] are among the few production database systems that rely on a user-space networking stack, namely Seastar. More recent DPDK-based systems, including the fast path used in our work, and cloud-oriented approaches such as Machnet [51], show that user-space networking can substantially improve performance in cloud environments.

Specialized communication abstractions. Specialized Remote Procedure Call (RPC) protocols have been developed for low-latency, small-message communication. While some systems use RDMA [60], others rely on datagram-based designs [27]. Higher-level data-processing frameworks such as MPI [38] and DFI [62] abstract away transport details and provide structured communication models, such as bulk-synchronous parallelism. These abstractions are effective when application logic matches their communication model.

Transport offload and alternative transports. Recent work has accelerated TCP by offloading parts of the kernel network stack to FPGAs or SmartNICs [30, 45, 57]. A complementary line of work proposes alternative transport protocols for modern large-scale datacenter networks [16, 21, 44, 55]. Tux [77] recently explored a database-oriented kernel-bypass networking stack that combines a message-based transport protocol with DBMS-specific pushdown mechanisms. These approaches are orthogonal to the bi-channel paradigm and complement it: when available in the target environment, they can serve as alternative implementations of the fast data channel instead of a DPDK/UDP-based fast path.

Control/data-plane separation. The separation of control and data planes originates in networking research, where it laid the foundation for Software-Defined Networking (SDN) [9]. Beyond networking, this pattern has been widely adopted in distributed storage systems [17, 58, 70]. Cloud database systems such as Snowflake [11] apply a related principle by separating query planning and optimization from distributed query execution. In contrast to this coarse-grained architectural separation, the bi-channel paradigm applies the idea at a finer granularity: it separates control and data paths within the networking layer of DBMS operators. This systems-level interpretation is closer in spirit to kernel-bypass operating systems such as Arrakis [49] and Demikernel [75].

8 Conclusion & Future Work

We presented the *bi-channel paradigm*, a new architectural principle for network-intensive database systems that separates communication into two coordinated planes: a high-performance, user-space data path and a reliable, kernel-based control path. This design enables systems to fully exploit modern NICs and cloud networking hardware without re-implementing TCP semantics in user space.

Our evaluation demonstrated that this paradigm combines performance and simplicity: a distributed shuffle operator saturates 200 Gbit/s using only three CPU cores, and a replicated key-value store sustains tens of millions of messages per second with predictable latency. Beyond raw performance, the bi-channel approach yields a clearer division of concerns, simplifying the design of distributed operators. We therefore view the bi-channel paradigm not merely as an optimization, but as a reusable systems pattern for database networking on emerging cloud hardware.

In this work, we chose user-space UDP for the fast path. Future work will explore integrating the paradigm with kernel-native fast paths such as `AF_XDP`, which can provide similar performance without external dependencies like DPDK.

9 Artifacts

The artifacts for reproducing our experiments are available at: https://github.com/GeorgKreuzmayr/bi-channel

The repository includes source code, experiment scripts, and reproduction instructions.

Acknowledgments

Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Amazon Web Services. 2023. Compute optimized instances. Amazon Elastic Compute Cloud Documentation. https://docs.aws.amazon.com/ec2/latest/instancetypes/co.html Accessed: 2026-01-31.

[2] Amazon Web Services. 2025. Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html Accessed: 2026-01-31.

[3] Amazon Web Services. 2025. DPDK driver for Elastic Network Adapter (ENA). https://github.com/amzn/amzn-drivers/tree/master/userspace/dpdk Accessed: 2026-01-31.

[4] Austin Appleby. 2011. SMHasher. https://github.com/aappleby/smhasher Accessed: 2026-01-31.

[5] Shivnath Babu and Herodotos Herodotou. 2013. Massively Parallel Databases and MapReduce Systems. *Found. Trends Databases* 5, 1 (2013), 1–104. doi:10.1561/1900000036

[6] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *NSDI*. USENIX Association, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (2016), 528–539. doi:10.14778/2904483.2904485

[9] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: taking control of the enterprise. In *SIGCOMM*, Jun Murai and Kenjiro Cho (Eds.). ACM, 1–12. doi:10.1145/1282380.1282382

[10] Mark A Cusack, John Adamson, Mark Brinicombe, Neil A Carson, Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, and Robert Wipfel. 2024. Yellowbrick: An Elastic Data Warehouse on Kubernetes. In *CIDR*. https://www.cidrdb.org/cidr2024/papers/p2-cusack.pdf

[11] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. doi:10.1145/2882903.2903741

[12] DPDK Project, Linux Foundation. 2026. Data Plane Development Kit. https://github.com/DPDK/dpdk/blob/main/drivers/net/mana/rx.c Accessed: 2026-01-31.

[13] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[14] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, Ethan L. Miller and Steven Hand (Eds.). ACM, 54–70. doi:10.1145/2815400.2815425

[15] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 20, 2 (mar 2023), 31:1–31:25. doi:10.1145/3587096

[16] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, Felipe Huici and Giuseppe Bianchi (Eds.). ACM, 1:1–1:12. doi:10.1145/2716281.2836086

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*, Michael L. Scott and Larry L. Peterson (Eds.). ACM, 29–43. doi:10.1145/945445.945450

[18] Google Cloud. 2025. Network bandwidth. https://cloud.google.com/compute/docs/network-bandwidth Accessed: 2026-01-31.

[19] Google Cloud. 2026. View Google Cloud packet loss dashboard. https://docs.cloud.google.com/network-intelligence-center/docs/performance-dashboard/how-to/view-google-cloud-packet-loss Accessed: 2026-01-31.

[20] Google Cloud Platform. 2026. Linux kernel driver for Compute Engine Virtual Ethernet. https://github.com/GoogleCloudPlatform/compute-virtual-ethernet-linux Accessed: 2026-03-31.

[21] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*. ACM, 29–42. doi:10.1145/3098822.3098825

[22] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *DaMoN*. ACM, 14:1–14:5.

[23] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2025. High-Performance DBMSs with io_uring: When and How to use it. *CoRR* abs/2512.04859 (2025). doi:10.48550/ARXIV.2512.04859 arXiv:2512.04859

[24] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *SIGCOMM*, Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy (Eds.). ACM, 295–306. doi:10.1145/2619239.2626299

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. *login Usenix Mag.* 41, 3 (2016). https://www.usenix.org/publications/login/fall2016/kalia

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *NSDI*. USENIX Association, 1–16.

[28] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *EuroSys*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 24:1–24:16. doi:10.1145/3302424.3303985

[29] Michael Kerrisk. 2025. io_uring_prep_send_zc. https://man7.org/linux/man-pages/man3/io_uring_prep_send_zc.3.html Accessed: 2026-01-31.

[30] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. 2023. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *NSDI*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 275–292. https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun

[31] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *CIDR*. CIDR Foundation. https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf

[32] H. T. Kung and Robert Morris. 1995. Credit-based flow control for ATM networks. *IEEE Netw.* 9, 2 (1995), 40–48.

[33] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[34] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. doi:10.1145/2588555.2610507

[35] LF Projects, LLC. 2025. Data Plane Development Kit. https://www.dpdk.org/ Accessed: 2026-01-31.

[36] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 355–370. doi:10.1145/2882903.2882949

[37] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*. ACM, 48–63. doi:10.1145/3064176.3064202

[38] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. 2004. High Performance RDMA-Based MPI Implementation over InfiniBand. *Int. J. Parallel Program.* 32, 3 (2004), 167–198. doi:10.1023/B:IJPP.0000029272.69895.C1

[39] Zirui Liu, Xian Niu, Wei Zhou, Yisen Hong, Zhouran Shi, Tong Yang, Yuchao Zhang, Yuhan Wu, Yikai Zhao, Zhuochen Fan, and Bin Cui. 2025. Extendible RDMA-based Remote Memory KV Store with Dynamic Perfect Hashing Index. In *ICDE*. IEEE, 1745–1758. doi:10.1109/ICDE65448.2025.00134

[40] Hendrik Makait, Bonaventura Del Monte, and Tilmann Rabl. 2024. Ghostwriter: a Distributed Message Broker on RDMA and NVM. In *ADMS@VLDB*. VLDB.org. https://vldb.org/workshops/2024/proceedings/ADMS/ADMS24_04.pdf

[41] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. doi:10.14778/3415478.3415568

[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[43] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (1986), 378–396. doi:10.1145/7239.7266

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 221–235. doi:10.1145/3230543.3230564

[45] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *NSDI*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon

[46] NVIDIA Corporation. 2025. sockperf. https://github.com/Mellanox/sockperf Accessed: 2026-01-31.

[47] John K. Ousterhout. 2022. It's Time to Replace TCP in the Datacenter. *CoRR* abs/2210.00714 (2022). doi:10.48550/ARXIV.2210.00714 arXiv:2210.00714

[48] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *CDMS@VLDB*, Satyanarayana R. Valluri and Mohamed Zaït (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper1_MoshaPasumansky.pdf

[49] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *OSDI*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[50] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (2015), 228–239. doi:10.14778/2856318.2856319

[51] Alireza Sanaee, Vahab Jabrayilov, Ilias Marinos, Anuj Kalia, Divyanshu Saxena, Prateesh Goyal, Kostis Kaffes, and Gianni Antichi. 2025. Fast Userspace Networking for the Rest of Us. arXiv:2502.09281 [cs.NI] https://arxiv.org/abs/2502.09281

[52] Michael Scharf and Sebastian Kiesel. 2006. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *GLOBECOM*. IEEE. doi:10.1109/GLOCOM.2006.333

[53] ScyllaDB. 2019. Seastar. https://seastar.io/ Accessed: 2026-01-31.

[54] ScyllaDB. 2025. ScyllaDB. https://www.scylladb.com/ Accessed: 2026-01-31.

[55] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC. *IEEE Micro* 40, 6 (2020), 67–73. doi:10.1109/MM.2020.3016891

[56] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 433–448. doi:10.1145/3299869.3300069

[57] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *NSDI*. USENIX Association, 87–102.

[58] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *MSST*, Mohammed G. Khatib, Xubin He, and Michael Factor (Eds.). IEEE Computer Society, 1–10. doi:10.1109/MSST.2010.5496972

[59] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*. ACM, 183–197.

[60] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *EuroSys*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 1–15. doi:10.1145/3064176.3064189

[61] Tencent Cloud. 2025. F-Stack. https://www.f-stack.org/ Accessed: 2026-01-31.

[62] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2022. DFI: The Data Flow Interface for High-Speed Networks. *SIGMOD Rec.* 51, 1 (2022), 15–22. doi:10.1145/3542700.3542705

[63] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. doi:10.14778/3681954.3682031

[64] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1041–1052. doi:10.1145/3035918.3056101

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[66] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD*. ACM, 1033–1048. doi:10.1145/3514221.3517824

[67] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *CoRR* abs/2207.03027 (2022). doi:10.48550/arXiv.2207.03027 arXiv:2207.03027

[68] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. 2020. Efficient Usage of One-Sided RDMA for Linear Probing. In *ADMS*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–13. http://www.adms-conf.org/2020-camera-ready/ADMS20_06.pdf

[69] Johannes Wehrstein, Roman Heinrich, Mihail Stoian, Skander Krid, Martin Stemmer, Andreas Kipf, Carsten Binnig, and Muhammad El-Hindi. 2025. Redbench: Workload Synthesis From Cloud Traces. *CoRR* abs/2511.13059 (2025).

[70] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 307–320. http://www.usenix.org/events/osdi06/tech/weil.html

[71] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2019. Catfish: Adaptive RDMA-enabled R-Tree for Low Latency and High Throughput. In *ICDCS*. IEEE, 164–175. doi:10.1109/ICDCS.2019.00025

[72] Channy Yun. 2025. New Amazon EC2 C8gn instances powered by AWS Graviton4 offering up to 600Gbps network bandwidth. AWS News Blog. https://aws.amazon.com/blogs/aws/new-amazon-ec2-c8gn-instances-powered-by-aws-graviton4-offering-up-to-600gbps-network-bandwidth/ Accessed: 2026-01-31.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[74] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The End of a Myth: Distributed Transactions Can Scale. *CoRR* abs/1607.00655 (2016).

[75] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 195–211. doi:10.1145/3477132.3483569

[76] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. 2025. OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck. In *CIDR*. www.cidrdb.org. https://www.cidrdb.org/cidr2025/papers/p17-zhou.pdf

[77] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. 2025. Tux: Efficient Drop-in Networking for Database Systems. *Proc. VLDB Endow.* 19, 3 (2025), 334–347. doi:10.14778/3778092.3778096

[78] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. 2023. Deploying User-space TCP at Cloud Scale with LUNA. In *USENIX ATC*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 673–687. https://www.usenix.org/conference/atc23/presentation/zhu-lingjun

[79] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD*. ACM, 685–699. doi:10.1145/3514221.3526187

[80] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?. In *DaMoN*, Spyros Blanas and Norman May (Eds.). ACM, 10:1–10:5. doi:10.1145/3533737.3538506