МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, КОМПЛЕКСОВ И КОМПЬЮТЕРНЫХ СЕТЕЙ/MATHEMATICAL SOFTWARE FOR COMPUTERS, COMPLEXES AND COMPUTER NETWORKS

DOI: https://doi.org/10.60797/IRJ.2025.162.128

IMPACT OF NETWORK ISOLATION ON INTERCONNECTING COMMUNICATIONS IN MICROSERVICE SYSTEMS

Research article

Ilichev V.Y.[1, *], Vershinin Y.V.[2], Yermakov Y.V.[3], Fadeev V.O.[4]
[1] ORCID : 0000-0003-1017-5544;
[1, 2, 3, 4] Bauman Moscow State Technical University, Kaluga, Russian Federation

* Corresponding author (patrol8[at]yandex.ru)

Abstract

This study examines in detail the impact of network isolation implemented using Istio-based mesh services and Envoy proxies on inter-container communications within microservice systems. To perform experiments, two modes of operation were chosen: the first — without an additional level of isolation, the second — with an activated sidecar proxy. During the study, key network metrics such as latency, bandwidth, and packet loss were measured for each configuration. The conducted comparative analysis of the data obtained made it possible to delve into the understanding of how the implementation of the mesh service affects the characteristics of the interaction between services, as well as formulate practical recommendations for choosing the optimal level of isolation when working in network clusters. This approach provides a better choice of architectural solutions and significantly improves the efficiency of microservice applications.

Keywords: service mesh, Istio, Envoy, microservice architecture, network isolation, latency, bandwidth, packet loss.

ВЛИЯНИЕ СЕТЕВОЙ ИЗОЛЯЦИИ НА МЕЖКОНТЕЙНЕРНЫЕ КОММУНИКАЦИИ В МИКРОСЕРВИСНЫХ СИСТЕМАХ

Научная статья

Ильичев В.Ю.[1, *], Вершинин Е.В.[2], Ермаков Я.В.[3], Фадеев В.О.[4]
[1] ORCID : 0000-0003-1017-5544;
[1, 2, 3, 4] Московский государственный технический университет имени Н.Э. Баумана, Калуга, Российская Федерация

* Корреспондирующий автор (patrol8[at]yandex.ru)

Аннотация

В данной работе подробно исследуется влияние сетевой изоляции, реализованной с использованием сервис-меша на основе Istio и прокси Envoy, на осуществление межконтейнерных коммуникаций в рамках микросервисных систем. Для выполнения экспериментов были выбраны два режима работы: первый — без дополнительного уровня изоляции, второй — с активированным sidecar-прокси. В процессе исследования для каждой из конфигураций были измерены ключевые сетевые метрики, такие как время ответа (latency), пропускная способность (bandwidth) и потери пакетов (packet loss). Проведённый сравнительный анализ полученных данных позволил углубиться в понимание того, как внедрение сервис-меша воздействует на характеристики взаимодействия между сервисами, а также сформулировать практические рекомендации по выбору оптимального уровня изоляции при работе в сетевых кластерах. Такой подход обеспечивает более обоснованный выбор архитектурных решений и позволяет значительно повысить эффективность работы микросервисных приложений.

Ключевые слова: сетевой меш, Istio, Envoy, микросервисная архитектура, сетевая изоляция, latency, bandwidth, packet loss.

Introduction

Modern software systems based on microservice architecture impose stringent requirements on the stability, security, and efficiency of inter-service communication. As the number of components grows, network interactions become increasingly complex: traffic volume escalates, routing logic intensifies, and demands for monitoring, access control, and fault tolerance rise significantly [1]. In such distributed environments, the network layer plays a critical role not only in data transfer but also in defining system resilience, observability, and scalability.

To ensure reliable operation of microservices, modern architectures employ infrastructure layers such as service meshes, which enable centralized management of service-to-service communication, enforce security policies (e.g., mutual TLS), provide load balancing, telemetry collection, and failure recovery mechanisms [2]. These solutions abstract networking complexities from application code, improving maintainability and operational visibility. However, the introduction of an additional network processing layer — particularly through sidecar proxies — inevitably introduces performance overhead, the extent of which depends on configuration, workload characteristics, and the degree of network isolation enforced.

Despite the widespread adoption of service meshes like Istio, Linkerd, and Cilium, there remains a gap in empirical research quantifying the impact of network isolation mechanisms— implemented via service mesh data planes — on key communication metrics such as latency, throughput, and packet loss in containerized environments. Most existing studies focus on functional benefits or qualitative assessments of reliability and security [2], while systematic measurement of performance trade-offs under controlled conditions is limited. For example, MeshInsight provides comparative benchmarks across service mesh implementations but lacks granular analysis of transport-layer behavior under sustained load. Similarly, evaluations of eBPF-based alternatives (e.g., Cilium) highlight reduced overhead compared to traditional sidecars [1], yet direct comparisons with full-featured control planes like Istio remain incomplete.

Moreover, regulatory trends toward zero-trust architectures and stricter compliance requirements (e.g., GDPR, NIST SP 800-204) are driving increased deployment of network segmentation and policy enforcement in cloud-native systems. This makes it essential to understand how such isolation measures affect not only security but also performance — a multidimensional challenge at the intersection of network engineering and distributed systems design.

Given this context, the relevance of the study lies in addressing the growing need for evidence-based decision-making when integrating service meshes into production-grade microservice platforms. Understanding the quantitative impact of network isolation enables architects to balance security and observability against performance degradation, particularly in latency-sensitive or high-throughput applications.

Objective of the work: to assess the degree of influence of network isolation, implemented via a service mesh (Istio), on inter-container communications in a Kubernetes-based microservice system.

Research tasks:

1. Design and deploy a reproducible testbed for measuring network performance in a microservice environment with and without service mesh activation.
2. Quantify changes in key communication metrics — response time (latency), bandwidth (throughput), and packet loss — induced by the introduction of Istio's sidecar-based data plane.
3. Statistically validate observed differences using appropriate inferential methods.
4. Analyze the implications of measured overhead for real-world deployment scenarios and formulate optimization recommendations.

This study contributes to closing the empirical gap in understanding the performance-cost trade-off introduced by service mesh technologies, providing actionable insights for both researchers and practitioners in cloud-native computing.

Research methods and principles

Service mesh is an infrastructure layer designed to manage communications between individual components of microservice applications. It allows you to transparently and centrally control network interactions, provide load balancing, improve security, and simplify monitoring. This approach helps developers to abstract from the complexities of network interaction, reduces the risks of failures, improves the observability of services and simplifies the maintenance of distributed systems. Service mesh is implemented by integrating special components into the existing infrastructure that intercept and control all network traffic between services without having to change their code.

Istio was chosen as the service mesh under study. Istio is an add-on on top of Kubernetes that expands the cluster network itself to the level of "canvas" — mesh network, through which requests from all services pass [3]. A side proxy container Envoy is automatically inserted into each Pod; it is he who intercepts the outgoing and incoming connections of the application, establishes a secure mTLS tunnel for them, collects telemetry and applies routing rules. All Envoy instances form a data plane, and the "brain" of the system — the control plane Istio — works on top of it. It stores policies (traffic, authentication, quotas), issues and rotates service certificates and "conveys" new settings to sidecar proxies using the compressed xDS protocol. Thanks to this separation, applications "do not know" about security, balancing or tracing — they only communicate via HTTP/gRPC, and a transparent network provides end-to-end encryption, tracing and management.

Three metrics were used to assess the impact of network isolation on inter-container communications in microservice systems:

1. Response time (latency) is the time interval between sending a request from a client and receiving the first byte of a response from a service. This indicator reflects how quickly the system processes and responds to requests.
2. Bandwidth. Throughput is a network connection characteristic that reflects the amount of data that can be transferred between services per unit of time. This metric allows you to evaluate the efficiency of a data link in a microservice architecture, where the interaction between application components is carried out over the network. Throughput determines how quickly individual microservices can communicate under given load conditions and runtime configuration.
3. Packet loss. Packet loss is an indicator of the quality of network communication. They reflect the proportion of network packets that did not reach the destination during transmission. Losses can occur due to network congestion, equipment malfunctions, routing conflicts, or network infrastructure configuration issues. In microservice systems, the presence of packet loss can lead to retransmission of data, increase response time, and reduce the overall stability of communication between services.

To improve reproducibility, experiments were conducted on a dedicated, server-class virtual dedicated server (VDS) with the following configuration: 8 vCPU (≥ 3.0 GHz), 16 GiB RAM, and a 40 GiB SSD. The operating system was Ubuntu 22.04 LTS (Linux x86_64). Docker [6] and kind (Kubernetes-in-Docker) [8] were installed on the host, atop which a Kubernetes cluster was deployed in a 1 control-plane + 3 worker topology. Container images were built locally and published to a registry accessible by the nodes; the worker nodes then pulled these images. Intra-cluster networking relied on standard Kubernetes mechanisms [7] (ClusterIP services and the built-in DNS).

The test application followed a microservices' architecture; measurements targeted the "entry-point HTTP node → registration processing service" path. The entry point was an Nginx instance serving static client assets; the target component was the pod responsible for processing registrations (implemented in Python with the uWSGI application server). Access to the target component was performed via a Kubernetes ClusterIP service, which allowed measuring network latency and throughput at the service-to-service level within the cluster, without involving external load balancers.

To assess TCP throughput and behavior under load, auxiliary (sidecar) containers were added to the working pods. The pod handling registrations ran iperf3 [5] in client mode (using the standard iperf3 TCP port), while the Nginx pod ran iperf3 in server mode. A separate Kubernetes ClusterIP service was exposed for iperf3 so that the test traffic followed the same network path and was subject to the same network policy rules as application calls.

HTTP-path load testing was performed with wrk [4], launched from a dedicated pod. The parameters were: 4 threads, 100 concurrent connections, a 30-second duration, and latency distribution reporting; the target URL was the internal DNS name of the ClusterIP service proxying to the registration-processing pod.

The study used the Istio service-mesh platform (version 1.25.1), installed with the demo profile. The control plane was deployed in the istio-system namespace and included the standard set of components for traffic routing and telemetry. Automatic injection of Envoy proxy sidecars (sidecar pattern) was enabled for the test services by labeling the namespace with istio-injection=enabled.

Traffic between containers is intercepted by node-level networking rules and routed through the local Envoy proxy in each pod. As a result, all service-to-service calls are governed at the service-mesh layer: encryption policies, tracing, metrics, and unified default timeouts/retries are applied. In the default installation, Auto mTLS is enabled: if both ends of a connection have sidecars, the connection is transparently established over mutual TLS (mTLS) for the application code; if a sidecar is absent on one side, traffic is permitted in plaintext (PERMISSIVE compatibility mode). No custom DestinationRule/VirtualService resources were defined in the tests, so default timeouts and routing behavior were in effect.

Main results

The comparison was performed in two modes—without the service mesh and with Istio enabled—over twenty independent runs per mode (each run lasting 30 s). According to wrk, the mean response time without Istio was 11.66 ms with a mean spread of 1.68 ms; the maximum observed latency reached 41.34 ms. With Istio enabled, the mean response time increased to 15.00 ms and the spread to 3.88 ms, while the maximum was similar at 41.36 ms. Thus, the average latency rose by 28.6%, and the variability by roughly 2.3×, reflecting additional proxy-layer processing.

According to iperf3, the mean throughput without Istio was 14.53 Gbit/s (about 1,014.8 GiB transferred across the series), whereas with Istio it decreased to 7.43 Gbit/s (about 518.8 GiB), i.e., a reduction of approximately 49%.

Auxiliary TCP quality indicators showed that the total number of retransmissions across the series without Istio was 64,904, which corresponds to ~64.0 retransmits per 1 GiB and an estimated loss of ~0.0087% at MSS = 1460 B; with Istio, 157 retransmissions were recorded (~0.30 per 1 GiB) with an estimated loss of ~0.000041%.

These differences in retransmissions should be interpreted in light of the fact that the reduced target sending rate in the presence of the proxy is accompanied by a more conservative operating regime of the transport stack and fewer retransmissions per unit of traffic.

For each configuration (without Istio / with Istio), we obtained n = 20 independent runs. For latency, we analyzed wrk mean-latency values averaged across runs; for throughput, iperf3 values (Gbit/s) averaged across runs. The comparison was performed on the means using Welch's two-sample t-test (unequal variances); 95% confidence intervals (CIs) are reported for the difference of means.

Latency (ms).

$\bar{x}_{no} = 11.66$, $s_{no} = 0.25$, $n_{no} = 20$; $\bar{x}_{istio} = 15.00$, $s_{istio} = 0.22$, $n_{istio} = 20$.

Difference of means: +3.34 ms; standard error ≈ 0.08 ms; $t \approx 41.8$, $df \approx 38.5$. 95% CI: [+3.18; +3.50] ms. The difference is statistically significant at $\alpha = 0.05$ (p d 0.001). Cohen's ≈ 13 — very large.

Throughput (Gbit/s).

$\bar{x}_{no} = 14.53$, $s_{no} = 0.90$, $n_{no} = 20$; $\bar{x}_{istio} = 7.43$, $s_{istio} = 0.53$, $n_{istio} = 20$.

Difference of means: −7.10 Gbit/s (~49% decrease); standard error ≈ 0.54 Gbit/s; $t \approx -13.2$, $df \approx 19.4$. 95% CI: [−8.24; −5.96] Gbit/s. The difference is statistically significant ($p \ll 0.001$).

Retransmissions / loss. For TCP retransmissions, the distribution of counts is discrete and skewed; we therefore did not apply a strict t-test. Instead, we used a normalized metric — retransmits per 1 GiB: 64.0/GiB (without Istio) vs 0.30/GiB (with Istio). With similar run durations, this indicates a substantially lower retry rate with the service mesh enabled in this environment, consistent with the estimated loss fractions (0.0087% vs 0.000041%). This should be interpreted with the caveat that the target sending rate also decreased along with retransmissions (average throughput roughly halved).

Discussion

For illustrative comparison, the experimental results are presented in Table 1.

Table 1 - Tests results

DOI: https://doi.org/10.60797/IRJ.2025.162.128.1

Metric	Without Istio	With Istio
Mean response time, ms	11.66	15.00
Response-time variability (mean σ per run), ms	1.68	1.75
Throughput, Gbit/s	14.53	7.43
Data transferred per run, GB	50.74	25.94
Retransmissions (Retr), count	3245	7.9
Estimated packet loss*, %	0.00896	0.000041
Retransmissions per 1 GB, count/GB	65.9	0.304

Note: TCP estimate: loss ≈ Retr × MSS / data volume (MSS = 1460 B)

With the service mesh enabled, the mean response time increased from 11.66 ms to 15.00 ms (~+28.6%), while the mean within-run variability remained at a comparable level (~1.7–1.8 ms). This indicates a moderate but consistent increase in latency due to the additional proxy hop and network processing. TCP throughput decreased by roughly a factor of two (from 14.53 to 7.43 Gbit/s), which aligns with the reduction in the amount of data transferred over a fixed time window. At the transport layer, we observed a marked decrease in the number of retransmissions and the derived estimate of packet loss with Istio enabled; in this environment, that can be interpreted as a more conservative sending regime: lowering the target sending rate reduces queue buildup and the frequency of retransmissions.

Published studies consistently show that an additional proxy hop, encryption, and telemetry collection in service meshes introduce measurable overhead; the magnitude of the effect depends on workload profile, connection duration, and the depth of traffic filtering. A representative example is the MeshInsight systematic study (ACM SoCC 2023), which reports increased latency and reduced throughput when sidecar proxies are enabled, as well as sensitivity of metrics to configuration and traffic type [9]. In practice, the size of the overhead varies across implementations: early independent benchmarks comparing Istio and Linkerd already noted latency advantages for “lightweight” stacks, especially for short requests. In parallel, an alternative class of “sidecar-less”/eBPF approaches is emerging: reports on Cilium Service Mesh indicate that moving parts of the data plane into the Linux kernel (eBPF) can markedly reduce the cost of packet interception and lower network-path overhead compared to the classic sidecar architecture [10]. For Istio, the ecosystem itself documents factors that influence performance (telemetry volume, encryption policy, Envoy filter configuration) and recommends practices to reduce load, including metrics optimization and hierarchical (federated) scraping via Prometheus [11]. Overall, external results align with our observations: enabling the mesh’s traffic-management layer increases median/quantile latencies and reduces throughput, explained by additional proxy processing and mTLS cryptography.

Conclusion

The results of this study demonstrate that the implementation of a service mesh — specifically Istio with Envoy sidecars — has a significant and measurable impact on inter-container communication performance in a Kubernetes-based microservice architecture. Under controlled experimental conditions, enabling the service mesh led to:

1. An increase in average response time from 11.66 ms to 15.00 ms (+28.6%), with statistically significant difference ($p \ll 0.001$, Cohen’s $d \approx 13$);
2. A reduction in TCP throughput from 14.53 Gbit/s to 7.43 Gbit/s (−49%), confirmed as highly significant ($p \ll 0.001$);
3. A dramatic decrease in retransmissions — from ~64 per GiB to ~0.3 per GiB — indicating improved transport-layer stability despite lower sending rates.

These findings clearly reveal a performance-security trade-off: while the service mesh enhances system observability, security (via mTLS), and fault tolerance, it simultaneously imposes substantial network-level overhead due to proxy mediation, encryption, and policy enforcement.

Novelty and originality of the work lie in the comprehensive, statistically validated measurement of multiple network metrics under identical hardware and software conditions, with particular emphasis on transport-layer behavior (retransmissions, effective loss estimation) rarely reported in prior literature. Unlike many existing studies that rely on synthetic benchmarks or qualitative claims, this research provides quantitative, reproducible evidence of performance degradation in a realistic microservice interaction path, supported by rigorous statistical analysis (Welch’s t-test, confidence intervals).

Furthermore, the integration of auxiliary tools (iperf3, wrk) within the same cluster topology ensures that test traffic traverses the same network policies and routing rules as application traffic, enhancing ecological validity.

In summary, this work delivers a clear, empirically grounded assessment of the cost of network isolation in modern microservice systems. The results underscore the necessity of careful architectural planning when deploying service meshes, especially in performance-critical domains such as fintech, real-time analytics, or edge computing. Future work may explore optimization strategies — including eBPF offloading, telemetry sampling, and hierarchical control planes — to mitigate performance penalties while preserving security and observability benefits.

Конфликт интересов

Не указан.

Рецензия

Все статьи проходят рецензирование. Но рецензент или автор статьи предпочли не публиковать рецензию к этой статье в открытом доступе. Рецензия может быть предоставлена компетентным органам по запросу.

Conflict of Interest

None declared.

Review

All articles are peer-reviewed. But the reviewer or the author of the article chose not to publish a review of this article in the public domain. The review can be provided to the competent authorities upon request.

Список литературы / References

1. Ильичев В.Ю. Сравнительный анализ технологий REST и SOAP для решения интеграционных задач / В.Ю. Ильичев, В.Е. Драч, А.С. Кушнир // Системный администратор. — 2022. — № 4(233). — С. 86–89.
2. Ильичев В.Ю. Сравнительный анализ VPN-протоколов Wireguard и OpenVPN / В.Ю. Ильичев, В.Е. Драч // Системный администратор. — 2022. — № 5(234). — С. 86–89.

3. Istio : официальная документация. — URL: https://istio.io/latest/docs/ (дата обращения: 18.05.2025).
4. Гайдамако В.В. Моделирование сервера в инфраструктуре облачного центра обработки данных на базе платформы SIMGRID / В.В. Гайдамако // Проблемы автоматики и управления. — 2023. — № 3(48). — С. 81–92.
5. Одоевский С.М. Особенности решения задач оценки требуемой пропускной способности каналов связи для мультимедийного трафика / С.М. Одоевский // Современное состояние и перспективы развития инфокоммуникационных сетей связи специального назначения : материалы науч.-практ. конф. — Санкт-Петербург, 2024. — С. 67–72.
6. Malyshev D.O. Analysis of packet losses in MPLS networks / D.O. Malyshev // Youth. Society. Modern science, technology, and innovations. — 2021. — № 20. — P. 352–354.
7. Инструмент нагрузочного тестирования HTTP. — URL: https://github.com/wg/wrk (дата обращения: 18.05.2025).
8. Инструмент для измерения производительности сети. — URL: https://iperf.fr/iperf-doc.php (дата обращения: 18.05.2025).
9. Основы работы с контейнерами Docker. — URL: https://www.docker.com (дата обращения: 18.05.2025).
10. Основы Kubernetes. — URL: https://kubernetes.io (дата обращения: 18.05.2025).
11. Быстрый старт с kind (Kubernetes IN Docker). — URL: https://kind.sigs.k8s.io (дата обращения: 18.05.2025).

Список литературы на английском языке / References in English

1. Il’ichev V.Yu. Sravnitel’nyi analiz tekhnologii REST i SOAP dlya resheniya integratcionnykh zadach [Comparative Analysis of REST and SOAP Technologies for Solving Integration Problems] / V.Yu. Il’ichev, V.E. Drach, A.S. Kushnir // Sistemyi administrator [System Administrator]. — 2022. — № 4(233). — P. 86–89. [in Russian]
2. Il’ichev V.Yu. Sravnitel’nyi analiz VPN-protokolov Wireguard i OpenVPN [Comparative Analysis of Wireguard and OpenVPN VPN Protocols] / V.Yu. Il’ichev, V.E. Drach // Sistemyi administrator [System Administrator]. — 2022. — № 5(234). — P. 86–89. [in Russian]
3. Istio : oficial'naya dokumentaciya [Istio : Official Documentation]. — URL: https://istio.io/latest/docs/ (accessed: 18.05.2025). [in Russian]
4. Gaidamako V.V. Modelirovanie servera v infrastrukture oblachnogo tsentra obrabotki dannykh na baze platformy SIMGRID [Server Modeling in the Cloud Data Center Infrastructure Based on the SIMGRID Platform] / V.V. Gaidamako // Problemy avtomatiki i upravleniya [Problems of Automation and Control]. — 2023. — № 3(48). — P. 81–92. [in Russian]
5. Odoevskiy S.M. Osobennosti resheniya zadach otsenki trebuemoi propusknoi sposobnosti kanalov svyazi dlya mul’timediinogo trafika [Features of Solving the Problems of Estimating the Required Throughput of Communication Channels for Multimedia Traffic] / S.M. Odoevskiy // Sovremennoe sostoyanie i perspektivy razvitiya infokommunikatsionnykh setei svyazi spetsial’nogo naznacheniya [Current State and Prospects of Development of Special-Purpose Infocommunication Networks] : Materials of the Scientific and Practical Conference. — St. Petersburg, 2024. — P. 67–72. [in Russian]
6. Malyshev D.O. Analysis of packet losses in MPLS networks / D.O. Malyshev // Youth. Society. Modern science, technology, and innovations. — 2021. — № 20. — P. 352–354.
7. Instrument nagruzochnogo testirovaniya HTTP [HTTP Load Testing Tool]. — URL: https://github.com/wg/wrk (accessed: 18.05.2025). [in Russian]
8. Instrument dlya izmereniya proizvoditel’nosti seti [Network Performance Measurement Tool]. — URL: https://iperf.fr/iperf-doc.php (accessed: 18.05.2025). [in Russian]
9. Osnovy raboty s konteinerami Docker [Fundamentals of Working with Docker Containers]. — URL: https://www.docker.com (accessed: 18.05.2025). [in Russian]
10. Osnovy Kubernetes [Kubernetes Basics]. — URL: https://kubernetes.io (accessed: 18.05.2025). [in Russian]
11. Bystryi start s kind (Kubernetes IN Docker) [Quick Start with kind (Kubernetes IN Docker)]. — URL: https://kind.sigs.k8s.io (accessed: 18.05.2025). [in Russian]