DOI: 10.15514/ISPRAS-2025-37(5)-7

The Dynamic Adaptive Packet Buffering (DAPB) Algorithm for Service Mesh Performance Enhancement Based on eBPF

H-D. Djambong Tenkeu, ORCID: 0009-0002-4689-1665 <Dzhambong.T.K@hse.ru>
D.V. Alexandrov, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>
National Research University "Higher School of Economics"
11, Pokrovsky blvd, Moscow, 109028, Russia.

Abstract. This paper introduces the Dynamic Adaptive Packet Buffering (DAPB) algorithm. Designed to enhance data transfer efficiency in modern networking environments, it is built on the principles of Nagle's algorithm. DAPB addresses the limitations of existing buffering techniques by dynamically adjusting its behavior based on real-time network conditions, application requirements, and latency sensitivity. The algorithm incorporates context-sensitive buffering, adaptive timeout mechanisms, and machine learning-driven predictions to achieve a balance between efficiency, latency, and energy consumption. DAPB's context-aware buffering tailors its strategy to the specific needs of the application, minimizing buffering for latency-sensitive applications like VoIP and online gaming, while maximizing buffering for throughput-sensitive applications like file transfers. The adaptive timeout mechanism dynamically adjusts the waiting timeout based on network conditions such as round-trip time, packet loss, and jitter, ensuring optimal performance under varying workloads. Machine learning models are used to predict optimal buffer sizes and timeout values, leveraging historical data and real-time metrics to improve decision-making. The algorithm also features selective aggregation, intelligently deciding which packets to aggregate and which to send immediately. This ensures that urgent packets are transmitted without delay, while nonurgent packets are aggregated to reduce overhead. Additionally, DAPB prioritizes energy efficiency by optimizing buffer sizes and timeout values, making it suitable for energy-constrained environments like edge computing and IoT devices. The DAPB algorithm is expected to improve the data transfer performance in various scenarios. Compared to the standard Nagle algorithm, the DAPB algorithm is expected to reduce latency, improve throughput, and enhance energy efficiency. This paper is the result of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE University).

Keywords: dynamic adaptive packet buffering (DAPB); extended Berkeley packet filter (eBPF); kernel-level packet processing; service mesh

For citation: Djambong Tenkeu H-D., Alexandrov D.V. The Dynamic Adaptive Packet Buffering (DAPB) Algorithm for Service Mesh Performance Enhancement Based on eBPF. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 93-110. DOI: 10.15514/ISPRAS-2025-37(5)-7.

Алгоритм динамической адаптивной буферизации пакетов (DAPB) для повышения производительности Service Mesh на основе eBPF

Х-Д. Джамбонг Тенке, ORCID: 0009-0002-4689-1665 <Dzhambong.T.K@hse.ru>
Д.В. Александров, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>

Национальный исследовательский университет «Высшая школа экономики», Россия, 109028, г. Москва, ул. Покровский бульвар, д. 11.

Аннотация. Данная статья представляет алгоритм динамической адаптивной буферизации пакетов (Dynamic Adaptive Packet Buffering, DAPB). Разработанный для повышения эффективности передачи данных в современных сетевых средах, алгоритм основан на принципах алгоритма Нейгла. DAPB преодолевает ограничения существующих методов буферизации за счет динамической адаптации поведения на основе текущих сетевых условий, требований приложений и чувствительности к задержкам. Алгоритм сочетает контекстно-зависимую буферизацию, адаптивные механизмы таймаутов и прогнозирование на основе машинного обучения для оптимального баланса между эффективностью, задержкой и энергопотреблением. Контекстно-ориентированная буферизация адаптирует стратегию под конкретные приложения: минимизирует буферизацию для чувствительных к задержкам сервисов (VoIP, онлайн-игры) и максимизирует для throughput-ориентированных задач (передача файлов). Адаптивный механизм таймаутов динамически регулирует период ожидания с учетом времени кругового обхода (RTT), потерь пакетов и джиттера, обеспечивая оптимальную производительность при изменяющейся нагрузке. Модели машинного обучения предсказывают оптимальные размеры буфера и значения таймаутов, используя исторические данные и метрики реального времени. Алгоритм реализует селективную агрегацию пакетов, интеллектуально определяя какие пакеты следует агрегировать, а какие передавать немедленно. DAPB уделяет особое внимание энергоэффективности за счет оптимизации параметров буферизации, что делает его применимым в энергоограниченных средах (edge computing, IoT устройства). По сравнению со стандартным алгоритмом Нейгла, DAPB демонстрирует снижение задержек, увеличение пропускной способности и улучшение энергоэффективности. Исследование выполнено в рамках Программы фундаментальных исследований Национального исследовательского университета "Высшая школа экономики" (НИУ ВШЭ).

Ключевые слова: динамическая адаптивная буферизация пакетов (DAPB); расширенный фильтр пакетов Беркли (eBPF); обработка пакетов в ядре; service mesh

Для цитирования: Джамбонг Тенке Х-Д., Александров Д.В.. Алгоритм динамической адаптивной буферизации пакетов (DAPB) для повышения производительности Service Mesh на основе eBPF. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 93–110 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)-7.

1. Introduction

The proliferation of micro-services as the de-facto standard for building scalable and resilient applications has necessitated the evolution of underlying infrastructures that can adeptly manage the complexities of distributed systems. Service meshes have emerged as a critical component in the cloud-native ecosystem, offering a dedicated infrastructure layer that simplifies inter-service communication, enforces security policies, and provides observability across microservices. Istio, a leading service mesh implementation, exemplifies this by deploying a sidecar proxy alongside each microservice, thus abstracting the intricacies of network management from the application logic.

Despite the advantages conferred by service meshes, they are not without their challenges. The introduction of an intermediary proxy layer, while beneficial for manageability and control, inadvertently introduces additional overheads (like higher latency) in the communication path. These overheads are particularly pronounced within the Linux kernel network stack, where packet transmission is subject to several context switching and kernel-space to user-space communication.

As microservices continue to scale and the demand for low-latency, high-throughput systems grow, the need to address these overheads becomes increasingly critical.

One of the main approaches to solving these issues consists of performing traffic buffering. It allows optimizing data transmission and managing network congestion. One of the most popular buffering algorithms is Nagle's algorithm, introduced by John Nagle in 1984. Improves TCP communication by reducing the transmission of small packets over networks [1]. Designed to address inefficiencies caused by applications sending frequent, tiny data bursts, it mitigates network congestion by temporarily buffering small writes until either an acknowledgment (ACK) is received for previous data or enough data accumulates to form a full TCP segment. The algorithm ensures that only one small packet remains unacknowledged at a time, preventing the network from being flooded with tiny packets.

Although effective for bulk data transfers, Nagle's algorithm can introduce latency in interactive applications like gaming or SSH due to its interaction with TCP's delayed ACK mechanism, which waits up to 200 milliseconds to combine ACKs with outgoing data. This trade-off led to criticism [2]. The main point is that the strength of this algorithm (reducing small packets) is also its weakness. Modern systems often disable it for latency-sensitive applications (e.g., VoIP) using the TCP_NODELAY socket option, but it remains valuable for optimizing high-throughput workloads like file transfers.

This paper introduces a novel algorithm to improve data transfer efficiency by dynamically adapting to real-time network conditions and application needs. It does so through context-sensitive buffering, adaptive timeouts, and machine learning. Such an algorithm shall help improving performance, namely reducing latency, and improve energy efficiency in modern networking environments.

The remainder of the paper is organized as follows. Section 2 discusses the related works, while Section 3 presents the background and motivation of the new algorithm. Section 4 describes the new algorithm. Section 5 defines the performance metrics that can be used to assess the performance of the new algorithm. Section 6 contains the risks and limitations of the DAPB algorithm. Section 7 specifies the next steps of this research.

2. Related works

eBPF (extended Berkeley Packet Filter) enables the execution of user-defined programs within the Linux kernel. Its lineage begins with the Berkeley Packet Filter (BPF), introduced in 1993 by McCanne and Jacobson as a mechanism to efficiently capture network packets in user space [3]. Classic BPF (cBPF) employed a simple register-based virtual machine to execute filter programs in the kernel, reducing unnecessary data copying between the kernel and the user space.

The transition to eBPF began in 2014 with its integration into Linux kernel 3.18. This overhaul, led by Alexei Starovoitov, reimagined BPF as a general-purpose execution environment [4]. Key enhancements included a 64-bit register model, a Just-In-Time (JIT) compiler, and a richer instruction set, enabling eBPF programs to interact safely with kernel data structures. The most transformative applications of eBPF have emerged in networking. The 2018 introduction of XDP (eXpress Data Path) [5] marked a paradigm shift by enabling packet processing at the driver layer, bypassing the kernel network stack entirely.

Network congestion is a common issue in computer network engineering. To solve it, several buffering algorithms and techniques were created. The Sliding Window Protocol, as described in [6], is fundamental to TCP's flow control, allowing multiple packets to remain in transit before requiring acknowledgments. This approach maximizes throughput while preventing receiver overload by dynamically adjusting the window size based on network conditions.

Modern congestion control algorithms such as TCP BBR ([7] & [8]) represent another category, using bandwidth and latency measurements to dynamically optimize transmission rates. Meanwhile, at the hardware level, Direct Memory Access (DMA) [9] and Zero-Copy Buffering [10] minimize

CPU involvement by enabling direct data transfers between devices and memory, significantly reducing latency in high-speed networks.

Nagle's algorithm reduces TCP overhead by buffering small writes until either: (1) enough data accumulate to fill a packet or (2) all sent data are acknowledged. Although it minimizes "tinygrams" that waste bandwidth, it can increase latency [6], prompting many real-time systems to disable it via 'TCP_NODELAY'. The algorithm remains fundamental in throughput-latency tradeoff studies.

Compared to existing network (Table 1) enhancement algorithms, the DAPB algorithm introduces several novel improvements. Context-sensitive adaptability and machine learning-driven optimization as key novelties. Unlike traditional Nagle's algorithm, which uses fixed rules, DAPB dynamically adjusts buffer sizes and timeout mechanisms based on real-time network conditions. These conditions include (but are not limited to) packet round-trip time (RTT), the variability in packet arrival times (known as jitter), and the packet loss. The DAPB algorithm also considers application requirements (e.g. latency sensitivity). Using machine learning, it can predict optimal configurations [11], ensuring better performance in diverse scenarios. Additionally, DAPB incorporates selective buffering to prioritize urgent packets, reducing latency for real-time applications, while optimizing energy efficiency for resource-constrained environments like IoT. This makes DAPB more versatile, efficient, and adaptive than static or rule-based algorithms.

Table 1: Comparative analysis of buffering algorithms.

Characteristic	Nagle's	BBR	Circular	QUIC	DAPB
Adaptivity	Fixed	Congestion	Fixed	Per-conn	ML-driven
Latency	Poor	Moderate	Low	Excellent	Context
Throughput	Moderate	High	High	High	Adaptive
Energy	None	Partial	None	None	Optimized
Protocol	TCP	Transport	Generic	QUIC	Multi-protocol
Layer	Kernel	CC	User	User	Kernel + eBPF
ML	No	No	No	No	Yes
Priority	None	None	None	Stream	Urgency
Dynamic	No	RTT	No	Per-flow	RTT + Jitter + Packet loss
HW Accel	No	No	No	No	Partial

- Comparison includes classic (Nagle's, Circular) and modern (BBR, QUIC) approaches
- DAPB introduces ML-driven adaptivity and multi-protocol support
- Evaluated for cloud-native service mesh requirements

3. Background and motivation

3.1 Flow of traffic within the service mesh and Linux operating system

In a service mesh architecture, communication between application components occurs through a dedicated infrastructure layer composed of programmable proxies (Fig. **1**). These proxies, deployed as sidecars (e.g., Envoy, Linkerd-proxy), run alongside application containers in user space. Instead of applications directly managing network logic, they delegate tasks like service discovery, retries, or mutual Transport Layer Security (mTLS) to their sidecars via local inter-process communication (IPC) mechanisms such as Unix domain sockets.

The service mesh divides responsibilities between a control plane (e.g., Istio Pilot [12], Linkerd's control plane) and a data plane (sidecar proxies). The control plane acts as a centralized orchestrator, distributing policies, routing rules, and security configurations (e.g., certificates for mTLS) to data plane proxies. These proxies enforce rules at the application layer (Layer 7), enabling features like HTTP/2-based load balancing, circuit breaking, and header-based routing. Unlike the Linux kernel's IP/TCP-centric approach, service meshes prioritize protocols like HTTP, gRPC, or service-specific APIs.

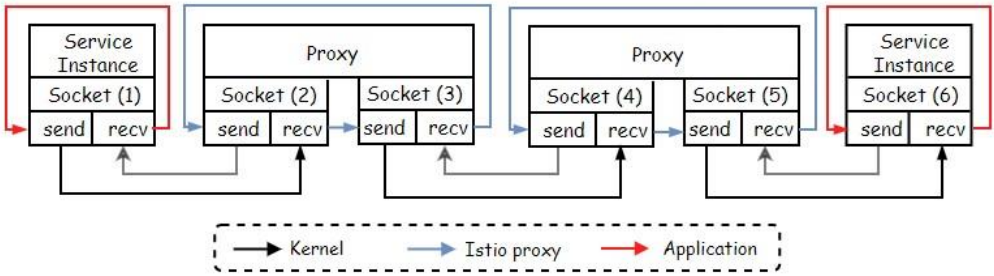

Fig. 1. Traffic flow within Istio service mesh.

Proxies intercept traffic using mechanisms like iptables rules or eBPF programs to redirect packets to the sidecar before reaching the application. For example, in Kubernetes, an init container may configure networking rules to ensure that all ingress/egress traffic flows through the proxy.

Within the Linux operating system (Fig. 2), data travels across multiple layers with distinct responsibilities. Applications in user space initiate communication using programming interfaces like sockets and system calls. For example, a web browser might use TCP socket functions from the standard C library to send HTTP requests. These requests get passed to the kernel via syscalls like sendto() or write(), which transition the execution from user mode to kernel mode.

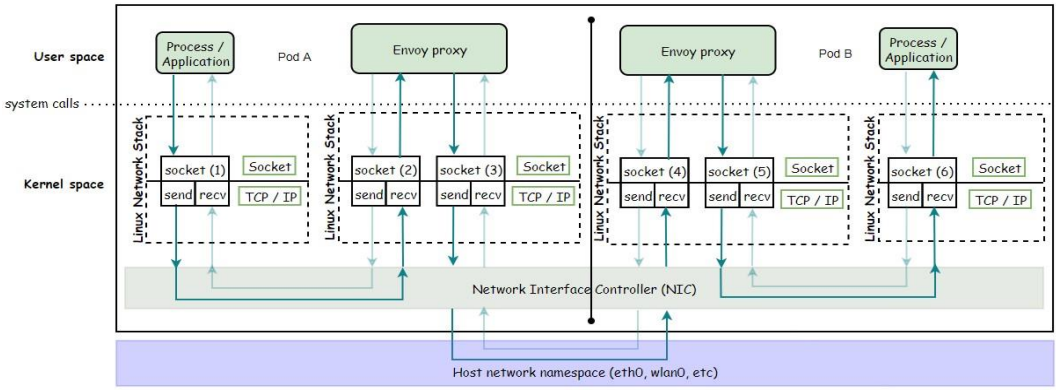

Fig. 2. Traffic flow in Linux OS running within a service mesh.

3.2 Nagle's algorithm

Nagle's algorithm is used to optimize TCP communication by decreasing the number of small packets transmitted over the network [12]. Introduced by John Nagle in 1984, it is particularly effective in situations where applications frequently send small amounts of data. Its main objective is to reduce the overhead that comes with sending numerous small packets, which can contribute to network congestion and inefficient bandwidth usage. Here is how it works:

- Buffering Small Packets: When an application transmits a small amount of data (less than the Maximum Segment Size (MSS), Nagle's algorithm temporarily stores those data in a buffer instead of sending them immediately as a separate packet. This design reduces overhead by minimizing the number of small packets, a trade-off between latency and efficiency noted in [14] and [15].
- Combining Packets: The algorithm waits for one of the following conditions to occur:
  - ✓ An acknowledgment (ACK) from the receiver for data that has already been sent, or
  - ✓ Additional data from the application that can be combined with the buffered data.

This approach, while effective for bulk data transfers, can introduce undesirable latency for interactive applications, as observed in [16] and [17].

- Sending Large Packets: Once one of these conditions is satisfied, the algorithm transmits the buffered data along with any new data as a single larger packet. This optimization leverages network efficiency by amortizing per-packet overhead, a principle further analyzed in [18] and [19].

3.3 The eBPF Technology

eBPF (extended Berkeley Packet Filter) is a Linux kernel innovation that enables developers to run custom, event-driven programs securely within the kernel space without modifying kernel source code or rebooting the system. Originally designed for network packet filtering, eBPF has evolved into a versatile framework to improve performance, observability, and security in modern computing environments [20].

eBPF programs are executed in a sandbox environment, ensuring safety by verifying the code before execution to prevent crashes or resource leaks. These programs attach to predefined hooks in the kernel, such as network events, system calls, or function entries/exits, allowing real-time data processing. For example, eBPF can intercept network packets to optimize routing, monitor application behavior for debugging, or enforce security policies by auditing system activity. It offers the following key advantages:

- Performance: by operating in-kernel, eBPF minimizes context switches and data copying, reducing overhead for tasks like packet processing or monitoring.
- Flexibility: developers can dynamically load programs to adapt to changing needs, such as scaling service mesh traffic or troubleshooting latency.
- Safety: a built-in verifier ensures that programs run without destabilizing the kernel, enforcing strict rules on memory access and loop structures.

To understand eBPF, it is essential to explore its core concepts, including program types, maps, and specialized frameworks such as XDP.

3.3.1 eBPF program types

eBPF supports a variety of program types, each designed for specific use cases. These program types determine where and how eBPF programs can be attached within the kernel. Some common eBPF program types include the following:

- Socket Filtering: Used for filtering and processing network packets at the socket level.
- Kprobes and Uprobes: allow tracing of kernel and user-space functions, respectively, for debugging and observability.
- Tracepoints: Attach to predefined kernel tracepoints to monitor system events.
- XDP (eXpress Data Path): a high-performance program type for processing network packets at the earliest possible point in the kernel's networking stack.
- TC (Traffic Control): used for advanced packet processing and traffic shaping in the kernel's networking subsystem.
- Perf Events: Enable monitoring of hardware and software performance events.

3.3.2 eBPF Maps

eBPF maps are key-value data structures that allow eBPF programs to store and share data between user space and kernel space, or between multiple eBPF programs. They are a fundamental building

block for creating complex and stateful eBPF applications. Common types of eBPF maps include the following:

- Hash Maps: store key-value pairs in a hash table for efficient lookups.
- Array Maps: use integer keys to store fixed-size values, providing fast access.
- Per-CPU Maps: maintain separate data for each CPU core, great for high-performance use cases.
- Ring buffer: a high-throughput data structure for passing data between eBPF programs and the user space.
- LRU (Least Recently Used) Maps: automatically evict least recently used entries to manage memory efficiently.

Maps enable eBPF programs to maintain state, aggregate data, and communicate with user space applications, making them indispensable for advanced use cases like network monitoring and security enforcement.

3.3.3 eXpress Data Path (XDP)

XDP is a high-performance eBPF program type designed to process network packets at the earliest possible point in the kernel's networking stack, often before they reach the kernel's network layer. This makes XDP ideal for use cases requiring low-latency packet processing, such as:

- DDoS mitigation: dropping malicious packets before they consume system resources.
- Load balancing: distributing network traffic across multiple servers with minimal overhead.
- Packet filtering: implementing custom filtering logic at line rate.
- Protocol parsing: extracting and processing custom protocol headers efficiently.

XDP programs are typically attached to network interfaces and operate in one of three modes:

- Native Mode: runs the XDP program directly on the network interface card (NIC) driver.
- Off-loaded mode: offloads the XDP program to the NIC hardware for maximum performance.
- Generic Mode: runs the XDP program in the kernel as a fallback when hardware offloading is not available.

3.3.4 Use cases of eBPF

eBPF has found applications in a wide range of domains. Some notable use cases include:

- Networking: eBPF is widely used to optimize network performance by enabling efficient packet filtering, load balancing, and traffic shaping. For example, tools such as Cilium [21] leverage eBPF to implement high-performance Kubernetes networking and security policies.
- Observability: eBPF provides deep visibility into system and application behavior without requiring invasive instrumentation. Tools such as BPF Compiler Collection (BCC) and bpftrace allow developers to trace system calls, monitor file I/O, and analyze performance bottlenecks in real time.
- Security: eBPF enables runtime security enforcement by monitoring system calls, file access, and network activity. It can detect and prevent malicious behavior, such as privilege escalation attempts or unauthorized data exfiltration.
- Tracing and Profiling: eBPF can be used to trace function calls, measure latency, and profile applications, making it invaluable for debugging and performance tuning.

3.4 Problem statement

Service meshes in distributed systems face significant inefficiencies in data transfer due to the prevalence of small, unaggregated packets. These inefficiencies manifest as such:

- High Latency: Frequent small-packet transmissions introduce delays from protocol overhead (e.g., TCP headers, ACKs) and kernel processing.
- Low Throughput: Low network bandwidth caused by excessive packet fragmentation and interrupt handling.
- Energy Overhead: Increased CPU cycles for per-packet processing, increasing power consumption in data centers.

Current buffering algorithms are static and do not adapt to dynamic network conditions (e.g., variable RTT, congestion) or application-specific requirements (e.g., latency-sensitive vs. batch traffic).

Research Gap: Lack of adaptive, context-aware buffering mechanisms capable of dynamically balancing these trade-offs based on real-time network state and traffic patterns.

4. The Dynamic Adaptive Packet Buffering (DAPB) algorithm

4.1 The DAPB algorithm's technical architecture

The DAPB algorithm is a novelty designed to improve data transfer efficiency in modern networking environments. It operates within 4 cornerstones (Fig. 3):

4.1.1 Data Collection and Learning Layer

The architecture begins with metric collectors in user space, which gather real-time network data (e.g., latency, throughput, packet loss) from the Linux network stack and service instances. These metrics feed into a reinforcement learning model that optimizes buffering policies through continuous interaction with the environment. Historical data are stored for history-based recommendations, while an optimization neural network, tuned via differential evolution, refines decision-making parameters. This layer ensures DAPB adapts dynamically to changing network conditions and application needs.

4.1.2 Decision and Control Plane

The decision model synthesizes inputs from the learning layer to generate adaptive buffering policies. It balances competing objectives (e.g., latency vs. throughput) using the reinforcement model's predictions. The control plane enforces these policies across the system, coordinating with the policy applier to translate decisions into actionable rules. This centralized intelligence allows DAPB to adjust buffer sizes, timeouts, and aggregation strategies in real time, tailored to specific traffic patterns (e.g., prioritizing VoIP packets over file transfers).

4.1.3 Kernel-Level Execution

Policies are executed in kernel space via eBPF data structures, which enable efficient packet processing without modifying the kernel. The eBPF components intercept traffic at the socket and TCP/IP layers, applying buffering rules while minimizing overhead. By operating close to the Network Interface Controller (NIC), DAPB reduces context switches and leverages kernel bypass techniques when possible. This design ensures low-latency processing while maintaining compatibility with existing Linux networking infrastructure.

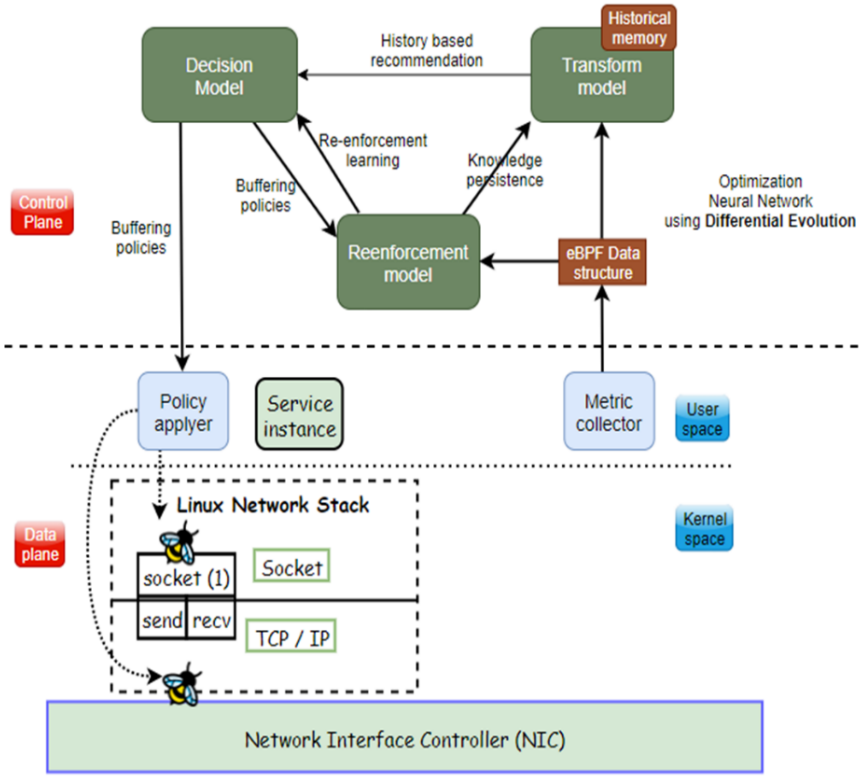

Fig. 3. The DAPB algorithm's technical architecture.

4.1.4 Feedback and Optimization Loop

The architecture closes the loop with knowledge persistence, where outcomes of applied policies (e.g., actual latency improvements) are logged and fed back into the learning layer. This continuous feedback enables the system to refine its models, ensuring long-term adaptability. The integration of differential evolution further optimizes neural network weights, while the control plane orchestrates iterative policy updates. Together, these components create a self-tuning system that evolves with network demands, achieving optimal performance across diverse service mesh environments.

As a result, the DAPB algorithm can be deployed and operate at the level of a whole service mesh installation (Fig. 4), managing buffering simultaneously for all containers within the installation.

4.2 The DAPB algorithm's features

The DAPB algorithm introduces several innovative features to address the limitations of existing buffering techniques.

4.2.1 Context-sensitive buffering

Unlike Nagle's algorithm, which uses a one-size-fits-all approach, DAPB tailors its buffering strategy to the specific needs of the application. For example, in latency-sensitive applications such as VoIP or online gaming, DAPB minimizes buffering to reduce delays. In contrast, in throughput-sensitive applications such as file transfers, it maximizes buffering to improve efficiency.

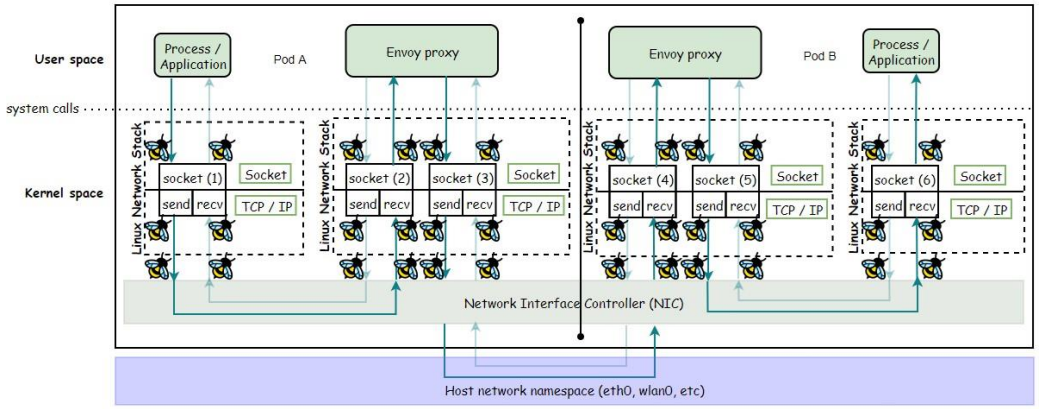

Fig. 4. Packet buffering using eBPF in Linux Kernel throughout service mesh.

Considering the following:

- $B(t)$ – the buffer size at time t, in bytes.
- $L(t)$ – the latency sensitivity of the application at time t (e.g., $L(t) = 1$ for latency-sensitive applications, $L(t) = 0$ for throughput-sensitive applications).
- $C(t)$ – the network conditions at time t, including round-trip time (RTT), packet loss rate ρ, jitter (J).

$$B(t) = \begin{cases} B_{min} & \text{if } L(t) = 1 \text{ (latency-sensitive)} \\ B_{max} & \text{if } L(t) = 0 \text{ (throughput-sensitive)} \\ f(C(t)) & \text{otherwise} \end{cases}, \quad (1)$$

where:

- B_{min} – the minimum buffer size for latency-sensitive applications (in bytes).
- B_{max} – the maximum buffer size for throughput-sensitive applications (in bytes).
- $f(C(t))$ – a function that adjusts the buffer size based on network conditions.

$$f(C(t)) = B_{min} + \alpha \cdot \text{RTT}(t) + \beta \cdot \rho(t) + \gamma \cdot J(t), \quad (2)$$

where α, β, *and* γ are weighting factors.

4.2.2 Adaptive timeout

While Nagle's algorithm relies on a fixed timeout, DAPB dynamically adjusts the waiting timeout based on real-time network conditions. If the network is congested, DAPB increases the timeout to allow more data buffering, thereby improving efficiency. However, if the network is underutilized, it reduces the timeout to minimize latency. This dynamic approach ensures that the DAPB strikes the right balance between efficiency and responsiveness. The timeout $T(t)$ is adjusted dynamically based on network conditions:

$$T(t) = T_{base} + \delta \cdot RTT(t) + \epsilon \cdot \rho(t) + \xi \cdot J(t), \quad (3)$$

where:

- $T(t)$ – the timeout period at time t (in ms).
- T_{base} – the base timeout value (in ms).
- δ, ϵ, *and* ξ – weighting factors that determine the influence of RTT, packet loss, and jitter on timeout.

4.2.3 Machine learning-driven predictions

DAPB incorporates machine learning-driven predictions to optimize its performance. The algorithm uses an AI model to predict network traffic. It provided a solid foundation for making informed decisions about buffer sizes and timeout values. For example, in a video streaming application, the AI component might analyze past transfer patterns to predict the best buffer size for a given video quality. This predictive capability ensures that DAPB remains effective even as network conditions and application requirements evolve. The predicted optimal buffer size $P(t)$ is derived from a machine learning model MM:

$$P(t) = M(C(t), H(t)), \tag{4}$$

where:

- $P(t)$ – predicted optimal buffer size at time t, derived from an AI model (in ms).
- $C(t)$ – current network conditions.
- $H(t)$ – historical data (e.g., past buffer sizes, network conditions, and performance metrics).

The buffer size $B(t)$ is then updated based on the prediction:

$$B(t) = min(B_{max}, max(B_{min}, P(t))). \tag{5}$$

4.2.4 Selective buffering

The algorithm intelligently decides which packets to buffer (and aggregate) and which to send immediately. Small packets that are part of a larger data stream are aggregated to reduce overhead, while urgent packets (e.g., control messages) are sent immediately to minimize latency. This selective approach ensures that the DAPB maintains high performance without compromising responsiveness. The average urgency of packets in the buffer at time t is given by:

$$U(t) = \frac{1}{N(t)} \sum_{i=1}^{N(t)} u_i(t), \tag{6}$$

where:

- $u_i(t)$ is the urgency of the i-th packet at time t ($0 \le u_i(t) \le 1$),
- $U(t)$ is the average urgency at time t,
- $N(t)$ is the number of packets in the buffer at time t,

The urgency $u_i(t)$ can be established either by using protocol headers or the application context. The first approach consists of extracting priority flags (e.g., HTTP/2 stream priorities, DSCP/ToS bits in IP headers, or gRPC metadata). The second consists of integrating with service mesh APIs (e.g., Istio virtual service) to label latency-sensitive traffic (e.g., VoIP, gaming) as high urgency. The decision to send the buffer is based on the following condition: send buffer if $U(t) > U_{th}$ or $B(t) \ge B_{max}$, where U_{th} is a threshold for the urgency of the packet.

4.2.5 Energy efficiency

Energy efficiency is also a priority for DAPB. By optimizing buffer sizes and timeout values, the algorithm reduces unnecessary resource consumption, making it particularly valuable for energy constrained environments like edge computing and IoT devices. For example, in an IoT sensor network, DAPB can minimize buffer sizes during periods of low activity, saving energy without compromising performance. The energy cost $E(t)$ is modeled as:

$$E(t) = \eta \cdot B(t) + \theta \cdot T(t), \tag{7}$$

where η and θ are weighting factors that represent the energy cost of buffering and waiting, respectively.

5. Performance Metrics

After applying the DAPB algorithm to service mesh using eBPF in Linux, 6 performance metrics should be collected (evaluated) and analyzed. Each of them is presented below.

5.1 Reduction in small packets

This metric quantifies the decrease in the number of small data packets transmitted over the network due to buffering. By holding data in a buffer until a predefined size or timeout is reached, fewer packets are sent, reducing overhead and eliminating network congestion. For example, if an application generates 1000 small packets per second but transmits only 200 after aggregation, 800 packets are eliminated, lowering processing demands on network hardware. Considering the following variables:

- N_{reduced} – the reduction in small packets (in bytes).
- $\lambda(t)$ – the arrival rate of the packets at time t (in packets/sec).
- $N_{\text{DAPB}}(t)$ – the number of packets transmitted after buffering, at time t (in bytes).

$$N_{\text{reduced}} = \sum_{t=0}^{T} (\lambda(t) - N_{\text{DAPB}}(t)) \tag{8}$$

5.2 Buffer efficiency

Evaluates how effectively the allocated buffer capacity is used to aggregate the data. High efficiency means that the buffer is consistently filled to its maximum capacity before transmission, minimizing wasted space. Lower efficiency indicates frequent early transmissions (e.g. due to timeouts), which may under-utilize buffer resources and reduce potential throughput gains. This is important because high efficiency directly means a reduction in latency (fewer waiting for partial fills). Considering the following:

- $B_{\text{max}}(t)$ – the maximum buffer size at time t (in bytes).
- $B(t)$ – the actual data accumulated in the buffer at time t (in bytes).
- $Tr_{\text{DAPB}}(t)$ – the transmissions triggered by buffer-full or timeout events (in ms).
- η – the buffer efficiency.

$$\eta = \frac{\sum_{t=0}^{T} (B_{\text{used}}(t) \cdot Tr_{\text{DAPB}}(t)}{\sum_{t=0}^{T} B_{\text{max}}(t)} \tag{9}$$

5.3 End-to-end latency

Estimates the time it takes for a request to traverse all nodes in the service mesh, including both network delays and buffering pauses. Highlights the bottlenecks where buffering dominates latency, guiding changes such as adjusting buffer sizes or timeouts to maintain responsiveness across distributed services. For each node v, considering the following:

- Q_v as a queue with capacity B_{max}.
- Edges e_{uv} have a transmission delay d_{uv}.

The Nagle-inspired policy modifies the dequeue behavior of Q_v. Packets are dequeued only when $\|Q_v\| \geq B_{\text{max}}$, or τ expires. After applying the DAPB algorithm, for a path $P = (v_1, v_2, \ldots, v_n)$, the end-to-end latency becomes:

$$L_P = \sum_{i=1}^{n-1} \left(d_{v_i, v_{i+1}} + \Pi_{Q_{v_i} < B_{\max}} \cdot \tau \right), \quad (10)$$

where:

- $d_{vi,vi+1}$ — the network delay between nodes v_i and v_{i+1} (in ms).
- Π — an indicator function for delayed transmissions (in ms).
- $\Pi_{Qvi<Bmax} \cdot \tau$ — the buffering delay at node v_i if its buffer is not yet full (in ms).

5.4 Additional delay

This metric estimates the additional delay introduced by buffering packets before transmission. While aggregation improves throughput, it inherently adds waiting time, either until the buffer fills or a timer expires. Applications sensitive to delays (e.g., real-time systems) must balance this trade-off carefully to avoid degrading user experience. Considering the following variables:

- τ – the acknowledgment timeout (in ms).
- t_{fill} – the time to fill the buffer to $B_{\max}$ (in ms).
- L_{added} – the added latency increase (in ms).

$$L_{added} = \mathbb{E}[\min(\tau, t_{\text{fill}})] \quad (11)$$

The increase in latency can be useful to estimate the effect of buffering on latency-sensitive applications (e.g., real-time APIs). For example, if $t_{\text{fill}} = 150ms$ and $\tau = 200ms$, the added latency is 150 ms. However, if $t_{\text{fill}} = 250ms$, the added latency is 200 ms (the timeout triggers transmission).

5.5 Throughput gain

The throughput gain reflects the improvement in data transmission rates achieved by sending larger aggregated packets instead of smaller ones. Larger packets reduce header overhead and improve network utilization, enabling more efficient bandwidth use. For example, combining 100 small packets into one large packet minimizes repetitive header transmissions, increasing throughput.

5.6 eBPF overhead cost

While eBPF optimizes kernel-level processing, its operations consume additional CPU cycles. This metric assesses the computational cost of using eBPF to manage packet aggregation. The cost must remain low enough to avoid negating the benefits of aggregation, ensuring net performance gains. It can be estimated as the additional processing time, memory increase, and energy consumption introduced by eBPF hooks to intercept, buffer, and redirect packets. Considering the following variables:

- $X^{\text{proc}}_{\text{native}}$ – the time (or memory or energy) to process a packet in the native Linux stack.
- $X^{\text{proc}}_{\text{eBPF}}$ – the time (or memory or energy) of all eBPF logics (e.g., aggregation, buffering).
- C_{eBPF} – the overhead cost induced by eBPF.

$$C_{eBPF} = X^{\text{proc}}_{\text{eBPF}} - X^{\text{proc}}_{\text{native}} \quad (13)$$

An important aspect of this indicator is that it allows to verify that eBPF enhancements are not less than the cost due to increased resource usage. For example, if C_{eBPF} = 5 μs/packet, and the packet arrival rate λ(t) = 1000 packets/sec the overhead is about 5 seconds of CPU time per second.

5.7 Performance metrics conclusion

The metrics can be interpreted holistically as follows:

- Reduction in Small Packets (N_{reduced}) and Throughput Gain (T_{gain}) measure improvements in network efficiency from aggregation. These are critical for throughput-sensitive applications (e.g., file transfers).
- End-to-End Latency (L_P) and Latency Increase (L_{added}) capture responsiveness trade-offs, vital for real-time systems (e.g., VoIP).
- The buffer efficiency (η) reflects resource utilization, indicating how well DAPB adapts buffer usage to dynamic conditions.

The context-specific guidance for the ML models is as follows:

- In throughput-sensitive context: prioritize N_{reduced}, T_{gain}, and η. The cost (C_{eBPF}) is tolerable if the gains exceed it.
- In latency-sensitive context: minimize L_P and L_{added}; tolerate lower η or higher C_{eBPF}.
- In energy-constrained context (Edge/IoT, etc.): favor η and low C_{eBPF}.

6. Risks and limitations

The DAPB algorithm introduces significant improvements over static buffering approaches, but its adaptive and machine learning-driven nature presents several challenges that must be carefully mitigated.

1. Prediction Inaccuracies in Dynamic Environments

   Risk: the machine learning model's reliance on historical data and real-time metrics may yield suboptimal predictions under sudden network changes (e.g., flash crowds, DDoS attacks). Noisy or incomplete data (e.g., inaccurate RTT measurements due to asymmetric routes) could degrade performance.

   Mitigation: incorporate ensemble methods (e.g., random forests [23]) to reduce variance and fallback mechanisms (e.g., reverting to Nagle-like static thresholds when prediction confidence is low).

2. Overhead from Adaptive Mechanisms

   Risk: the computational cost of dynamically adjusting buffer sizes and timeouts may offset throughput gains, especially in resource-constrained edge/IoT environments. The eBPF overhead metric must be monitored to ensure net benefits.

   Mitigation: profile the algorithm's CPU/memory footprint under varying loads and optimize the eBPF bytecode (e.g., reducing redundant calculations in the $f(C(t))$ and $T(t)$ functions).

3. Misclassification of Application Context

   Risk: incorrectly labeling an application as latency-sensitive ($L(t) = 1$) or throughput-sensitive ($L(t) = 0$) could lead to inappropriate buffering. For example, misclassifying VoIP traffic as batch processing would introduce unacceptable delays.

   Mitigation: implement hybrid labeling (e.g., allow applications to declare their sensitivity via API) and validate classifications using runtime telemetry (e.g., packet inter-arrival times).

4. Energy Trade-offs in Adaptive Buffering

   Risk: although DAPB optimizes energy use, frequent buffer resizing $B(t)$ and timeout adjustments $T(t)$ may increase CPU cycles, negating energy savings in low-power devices.

Mitigation: introduce hysteresis in adjustments (e.g., change $B(t)$ only when network conditions $C(t)$ shift beyond a threshold) to reduce computational churn.

5. Scalability in Large-Scale Deployments

   Risk: the centralized control plane in service meshes may struggle to propagate real-time network conditions $C(t)$ to all proxies, causing inconsistent buffering decisions across nodes.

   Mitigation: decentralize partial decision-making (e.g., let each proxy compute B(t) locally) and use lightweight consensus protocols for critical updates [24].

6. Security Implications of eBPF Dependencies

   Risk: eBPF's kernel-level access exposes DAPB to potential exploits (e.g., buffer overflow in eBPF programs). Maliciously crafted packets could trigger excessive buffering, leading to resource exhaustion.

   Mitigation: apply eBPF hardening techniques (e.g., verifier-based bounds checking, ratelimiting buffer allocations) and audit the DAPB eBPF code with tools like BPFKit [25].

7. Interoperability with Legacy Systems

   Risk: older kernels or non-Linux environments may lack eBPF support, limiting DAPB's applicability. Hybrid deployments (e.g., partial service meshes) could experience performance asymmetry.

   Mitigation: provide a fallback mode using socket-level buffering (e.g., TCP_CORK) with reduced adaptability, and document compatibility matrices.

7. Next steps of the research

1. Implement the new algorithm using eBPF.
2. Engineer the ML model.
3. Prepare multiple testing environments. Make sure to have most common architectures:
   - 2 containers inside a pod with one Istio sidecar (Intra-pod)
   - 2 containers inside 2 pods with 2 Istio sidecars (Inter-pod)
   - n containers inside n pods with n Istio sidecars (Inter-pod), where $2 < n < \infty$.

   There are two environments for each architecture. The new algorithm is applied to the first one, and it is not applied to the second one.
4. Collect basic system metrics. They include, but are not limited to, the CPU, the memory (RAM), and the disk usage (in bytes).
5. Integrate the new algorithm into the corresponding testing environments.
6. Collect and store performance metrics. These are the metrics described in section 5.
7. Analyze and assess the baseline system and performance metrics.
8. Conclude the work.

8. Conclusion

This paper introduces the Dynamic Adaptive Packet Buffering (DAPB) algorithm. It is designed to enhance data transfer efficiency in service mesh environments by leveraging eBPF. DAPB improves upon existing buffering algorithms like Nagle's algorithm by dynamically adjusting buffer sizes and timeout values based on real-time network conditions, application requirements, and machine learning predictions. Key features include context-sensitive buffering, adaptive timeout mechanisms, selective aggregation, and energy efficiency optimizations, making it suitable for diverse scenarios such as latency-sensitive applications and resource-constrained IoT devices.

Performance metrics can be used to assess the reduction in small packets, the improved throughput, and the added latency

References

[1]. J. Nagle, Congestion control in IP/TCP internetworks, RFC Editor, RFC 896, Jan. 1984, Obsoleted by RFC 1122, but foundational to Nagle's algorithm. [Online]. Available: https: //tools.ietf.org/html/rfc896.

[2]. J. Nagle, "Congestion control in IP/TCP internetworks," RFC Editor, RFC 896, (Jan. 1984), Obsoleted by RFC 1122, but foundational to Nagle's algorithm. [Online]. Available: https: //tools.ietf.org/html/rfc896.

[3]. TCP/IP Illustrated, Volume 1: The Protocols ([1994]), W. R. Stevens, Addison-Wesley Professional, isbn: 978-0201633467.

[4]. "The BSD packet filter: a new architecture for user-level packet capture" ([1993]), S. McCanne et al., (In: Proceedings of the USENIX Winter 1993 Conference), pp. 259–270.

[5]. "BPF: In-kernel Virtual Machine" ([2015]), A. Starovoitov, (In: Linux Plumbers Conference).

[6]. "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel" ([2018]), T. Hoiland-Jorgensen et al., (In: Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies), pp. 54–66, doi: 10.1145/3281411.3281443.

[7]. Computer Networks ([2011]), A. S. Tanenbaum et al., Pearson.

[8]. "BBR: Congestion-based congestion control" ([2016]), N. Cardwell, Y. Cheng, C. S. Gunn, et al., ACM Queue, 14, 5, pp. 20–53, doi: 10.1145/3012426.3022184.

[9]. N. Cardwell, Y. Cheng, S. H. Yeganeh, et al., "Tcp bbr v2 alpha/release history," IETF, RFC 8962, (2021). [Online]. Available: https://tools.ietf.org/html/rfc8962.

[10]. Understanding Linux Network Internals ([2005]), C. Benvenuti, O'Reilly, isbn: 9780596002558.

[11]. "Efficient data transfer through zero copy" ([2006]), W. Ma et al., (In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing), pp. 1–12, doi: 10.1145/1188455.1188583.

[12]. "Differential evolution optimization for constrained routing in Wireless Mesh Networks" ([2014]), M. Sanni et al., (In: International Conference on Frontiers of Communications, Networks and Applications (ICFCNA 2014 - Malaysia)), pp. 1–6, doi: 10.1049/cp.2014.1397.

[13]. Istio, Istio: A service mesh for microservices, Official documentation, (2023). [Online]. Available: https://istio.io/latest/docs/concepts/what-is-istio/.

[14]. "Congestion control in IP/TCP internetworks" ([1984]), J. Nagle, ACM SIGCOMM Computer Communication Review, 14, 4, pp. 11–17, doi: 10.1145/1024908.1024910.

[15]. "Congestion avoidance and control" ([1988]), V. Jacobson, ACM SIGCOMM Computer Communication Review, 18, 4, pp. 314–329, doi: 10.1145/52325.52341.

[16]. R. Braden, "Requirements for internet hosts—communication layers," IETF, RFC 1122, (1989). [Online]. Available: https://tools.ietf.org/html/rfc1122.

[17]. "Reducing web latency: the virtue of gentle aggression" ([2013]), T. Flach et al., (In: Proceedings of the ACM SIGCOMM 2013 Conference), pp. 159–170, doi: 10.1145/2486001.2486030.

[18]. "Evaluating the impacts of alternative TCP congestion control algorithms" ([2008]), S. Ha et al., (In: IEEE International Conference on Network Protocols), pp. 49–58, doi: 10.1109/ICNP. 2008.4697036.

[19]. "TCP Vegas: End to end congestion avoidance on a global internet" ([1995]), L. S. Brakmo et al., IEEE Journal on Selected Areas in Communications, 13, 8, pp. 1465–1480, doi: 10.1109/ 49.464716.

[20]. Computer Networking: A Top-Down Approach ([2021]), J. F. Kurose et al., Pearson, isbn: 9780135928615.

[21]. Learning eBPF ([Mar. 2023]), L. Rice, O'Reilly, isbn: 978-1-098-13887-5.

[22]. T. Graf et al., "Cilium: eBPF-based networking, security, and observability," Isovalent, Tech. Rep., (2023), Official documentation. [Online]. Available: https://docs.cilium.io/en/stable/index.html.

[23]. "Network Shortcut in Data Plane of Service Mesh with eBPF" ([Jan. 2024]), W. Yang et al., Journal of Network and Computer Applications, 222, 1, p. 103805, doi: 10.1016/j.jnca. 2023.103805.

[24]. A. Cutler et al., "Random forests," in Research Gate, (Jan. 2011), vol. 45, pp. 157–176, isbn:978-1-4419-9325-0. doi: 10.1007/978-1-4419-9326-7_5.

[25]. "Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms" ([Jan. 2024]), G. Zhang et al., ACM Comput. Surv., 56, 5, doi: 10.1145/3636553.

[26]. Gui774ume, eBPFKit: A rootkit and intrusion detection system based on ebpf, https:// github.com/Gui774ume/ebpfkit, GitHub repository, (2021). [Online]. Available: https://github.com/Gui774ume/ebpfkit.

Информация об авторах / Information about authors

Ханк-Дебэн ДЖАМБОНГ ТЕНКЕ – магистр программной инженерии, аспирант НИУ “Высшая Школа Экономики”, приглашенный преподаватель на факультет компьютерных наук НИУ “Высшая Школа Экономики”. Сфера научных интересов: инженерия программного обеспечения и компьютерных систем.

Hank-Debain DJAMBONG TENKEU – Master of Science in Software Engineering, postgraduate student at the National Research University “Higher School of Economics”, invited lecturer at the Faculty of Computer Science of NRU “Higher School of Economics”. Research interests: Software and Computer Systems Engineering.

Дмитрий Владимирович АЛЕКСАНДРОВ является Профессором в департаменте программной инженерии факультета компьютерных наук у НИУ “Высшая Школа Экономики”. Он также является заведующим научно-учебной лаборатории облачных и мобильных технологий. Его научные интересы включают методы и технологии искусственного интеллекта, машинное обучение и анализ данных, iOS разработка, разработка мобильных приложений, разработка программного обеспечения, indoor-навигация, базы данных, разработка игр.

Dmitry Vladimirovich ALEXANDROV is a Professor in the Department of Software Engineering, Faculty of Computer Science, National Research University “Higher School of Economics”. He is also the Head of the Research and Educational Laboratory of Cloud and Mobile Technologies. His research interests include methods and technologies of artificial intelligence, machine learning and data analysis, iOS development, mobile application development, software development, indoor navigation, databases, game development.