International Journal of Hybrid Innovation Technologies
Vol.5, No.1 (2025), pp.25-36
http://dx.doi.org/10.21742/ijhit.2653-309X.2025.5.1.03

# Assessing the Computational Impact of Handlespace Management in Lightweight Reliable Server Pooling Architectures

Lukas Steinbauer[1*], Anna-Maria Vogel[2], and Florian Neubauer[3]

[1]*Institute of Computer Engineering, Vienna University of Technology (TU Wien), Austria*
[2] *Department of Information Technology, Graz University of Technology, Austria*
[3]*Department of Distributed Systems, Johannes Kepler University Linz, Austria*
[1]*lsteinbauer@ict.tuwien.ac.at,* [2] *amvogel@infotech.tugraz.at,* [3]*fneubauer@ds.jku.at*

## *Abstract*

*Ensuring the uninterrupted availability of services has become increasingly vital in today's Internet-driven applications. To address this demand, the Reliable Server Pooling (RSerPool) framework, currently undergoing standardization by the Internet Engineering Task Force (IETF), offers a robust and lightweight solution for server redundancy and seamless session failover. Distinguished by its minimal computational and memory overhead, RSerPool is particularly well-suited for deployment in resource-constrained environments, including embedded systems and network edge devices. While earlier research has examined the general behavior and performance characteristics of RSerPool in specific application scenarios, such as Voice-over-IP, e-commerce platforms, and distributed computing systems—there remains a critical lack of comprehensive analysis focusing on the overhead introduced by handlespace management. This paper seeks to fill that gap by providing a detailed evaluation of the computational and structural performance of the pool maintenance subsystem, a core component of the RSerPool architecture. The study begins by presenting a clear overview of the RSerPool framework, including its protocol stack, operational principles, and policy-driven mechanisms for server selection. Subsequently, we introduce an efficient handlespace structure based on red-black trees and examine its behavior through both controlled simulations and real-world experimental setups. Performance metrics such as registration throughput, re-registration frequency, timer management efficiency, and handle resolution scalability are measured and analyzed. The results demonstrate that the proposed handlespace management approach exhibits excellent performance even under high-load conditions. Thus, the findings affirm RSerPool's viability for deployment in systems requiring high availability and low maintenance overhead, especially within limited-resource environments.*

***Keywords:*** *Reliable server pooling, Handlespace management, Microbenchmarking, Controlled simulation, Policy-agnostic abstraction*

**Article Info:**
Received (March 30, 2025), Review Result (April 25, 2025), Accepted (June 2, 2025)
*corresponding author

## 1. Introduction

Ensuring high availability of networked services has become a critical requirement in modern Internet infrastructures. Traditional telecommunication systems achieve reliability through hardware-level redundancy; however, equivalent mechanisms for Internet-based services have historically remained application-specific and fragmented [1]. The Reliable Server Pooling (RSerPool) framework, developed within the Internet Engineering Task Force (IETF), offers a standardized, protocol-based solution to enable server redundancy and seamless session failover while maintaining low computational and memory overhead [2]. This lightweight architecture is particularly attractive for deployment in embedded systems and constrained network environments such as edge computing nodes, routers, and IoT gateways. Recent developments in distributed systems and mobile edge computing have further emphasized the need for efficient pooling and failover mechanisms. In particular, studies have shown that optimized server placement and intelligent pool management significantly improve latency, energy efficiency, and fault tolerance in edge and fog environments [3][4][5][6].

Furthermore, workload orchestration and dynamic resource allocation in Multi-access Edge Computing (MEC) environments demonstrate the benefits of scalable, policy-driven pooling strategies [7][8]. Although several researchers have investigated load balancing and redundancy strategies in distributed infrastructures [9][10], relatively few have performed a granular evaluation of the internal pool maintenance operations—such as registration, re-registration, handle resolution, and synchronization—that underpin RSerPool's practical feasibility. Existing performance studies often focus on specific scenarios or protocols, lacking a general-purpose analysis that applies across policy types and usage contexts [11][12].

This paper addresses that gap by evaluating the computational overhead and scalability characteristics of handlespace management in RSerPool. In particular, we focus on the performance implications of using red-black tree structures to maintain handlespaces across a variety of pooling policies. Our central research question is: What is the impact of handlespace operations on system performance under high load and constrained-resource scenarios? To answer this, we perform a two-pronged evaluation: first, through controlled micro-benchmarking of core operations, and second, through real-system testing under realistic workloads. Unlike prior work limited to application-specific insights, our approach remains policy-agnostic and emphasizes the portability of the design to embedded and lightweight systems.

The main contributions of this study are threefold. First, we propose a scalable, efficient handlespace architecture using red-black trees to support pool maintenance across arbitrary policy configurations. Second, we provide detailed empirical measurements of registration, re-registration, timer scheduling, and handle resolution operations in both synthetic and deployed environments. Third, we demonstrate that our design sustains high throughput and low latency even under significant workload and minimal hardware capabilities, affirming its suitability for edge computing and real-time distributed systems. Ultimately, this research contributes to the operational maturity of RSerPool and informs best practices for its integration into emerging high-availability service frameworks.

## 2. Related works

Ensuring high availability in distributed systems has long motivated the development of diverse fault-tolerant architectures, ranging from replication-based middleware to

containerized orchestration frameworks. Many of these solutions emphasize strong consistency and durability guarantees, but often at the cost of high resource consumption, which renders them unsuitable for deployment in embedded or constrained environments [13][14]. The need for lightweight and adaptive pooling mechanisms is increasingly evident in the context of mobile computing, Internet of Things (IoT), and edge computing infrastructures.

Recent research has explored adaptive server selection strategies to enhance system responsiveness under dynamic workloads. For instance, Zhou and Xu proposed an adaptive assignment scheme for cloud-hosted applications, which adjusts server allocation based on system utilization and request latency [15]. Alkaff et al. introduced a context-aware provisioning model for mobile cloud computing that considers network quality and user mobility, showing notable improvements in failover efficiency [16]. These contributions underscore the growing importance of policy-driven and context-sensitive server selection but do not address the computational and structural overhead associated with managing server pools over time.

At the protocol layer, the Stream Control Transmission Protocol (SCTP) has been studied for its built-in support for multihoming and failover, which are relevant features for server redundancy. However, SCTP implementations, as discussed by Tüxen and Stewart, typically lack abstraction layers for pool-wide handlespace management and do not offer built-in selection policies [17]. In contrast, orchestration platforms such as OpenADN and FogPlan integrate resource pooling with dynamic network control to manage large-scale distributed applications [18][19]. While these systems achieve high-level adaptability, they tend to focus on control-plane logic rather than the low-level data structure operations necessary for efficient handlespace maintenance.

The need for efficient and scalable data structures in runtime environments has been addressed in several studies. Prokopec et al. presented concurrent tries with efficient, non-blocking snapshots for high-throughput applications, demonstrating how structure-aware concurrency can optimize update and lookup operations [20]. Likewise, Moir et al. proposed relaxed balancing in tree structures to trade strict ordering for higher throughput in concurrent scenarios [21]. Although these contributions are not directly applied to server pooling, their insights are particularly relevant for improving the performance of handlespace operations under high churn.

From a systems perspective, the cost of pool membership updates—particularly registration, re-registration, and timeout-triggered removal—has been shown to affect system responsiveness in failover-critical applications. Bhatia and Sharma evaluated failover performance in hybrid cloud environments, concluding that control-plane latency plays a decisive role in recovery time [22]. However, the internal mechanisms responsible for these updates, such as handlespace indexing, policy-driven ordering, and synchronization timing, remain largely under-evaluated.

In summary, while the literature offers robust solutions for service failover, adaptive load balancing, and scalable orchestration, it often abstracts away from the structural and algorithmic underpinnings of pool management. There is a clear research gap concerning the practical overhead introduced by the internal operations of server pooling frameworks. This study addresses that gap by offering a detailed, policy-neutral evaluation of pool maintenance using red-black tree–based handlespace management. The analysis is supported by both simulation-based micro-benchmarking and real-system experimentation, with particular emphasis on scalability and suitability for low-resource environments.

## 3. Methodology

The primary objective of this study is to quantitatively assess the computational and memory overhead introduced by handlespace operations in Reliable Server Pooling (RSerPool) systems. A red-black tree–based approach was selected for its deterministic time complexity and suitability for dynamic environments where rapid insertions, deletions, and lookups are frequent. The methodology integrates both simulation-based microbenchmarking and real-system experimentation to evaluate the performance of handlespace operations under varying policies, workloads, and deployment scenarios. This dual approach ensures that the findings are both theoretically robust and practically validated. An overview of the methodology is presented in [Figure 1].

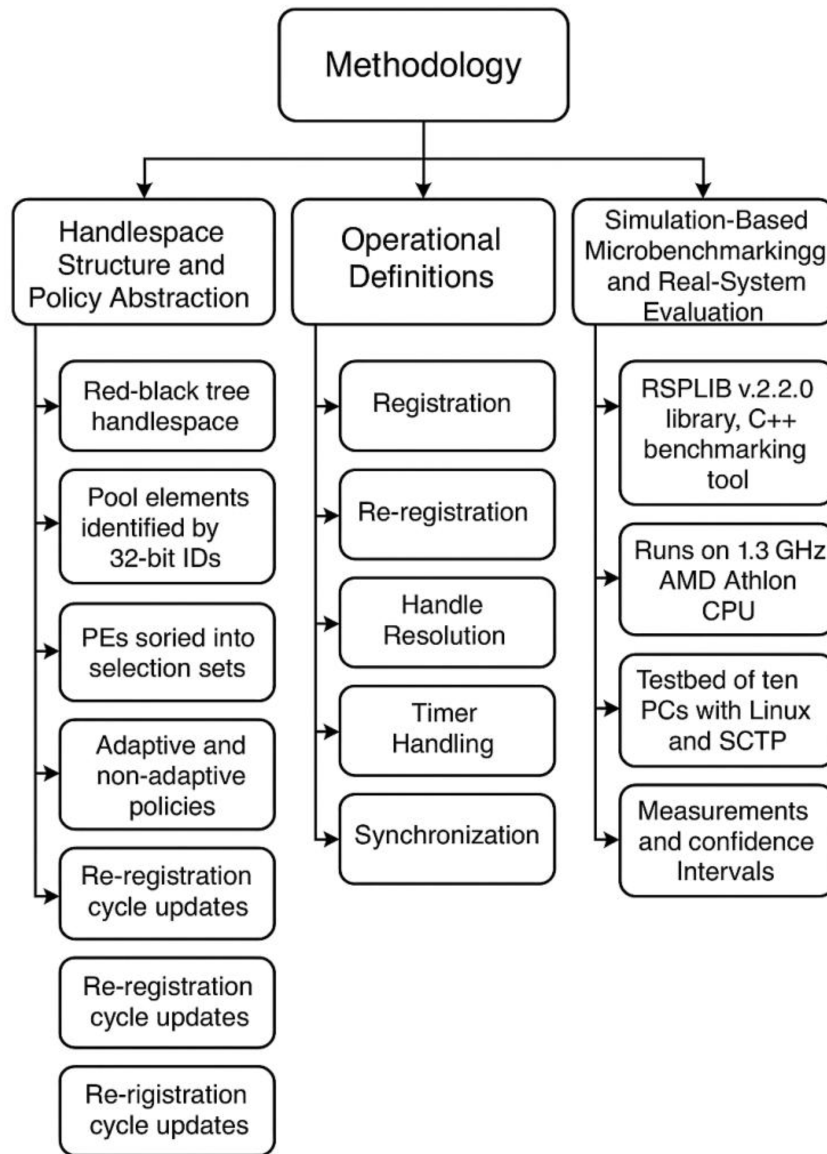


Figure 1. Methodology overview for handlespace performance evaluation in RSerPool systems

### 3.1. System architecture and handlespace design

The RSerPool architecture comprises Pool Registrars (PRs), Pool Elements (PEs), and Pool Users (PUs), where PRs maintain the handlespace. This data structure stores pool handles (PHs) and their associated registered PEs. Each PH can map to a dynamically varying number of PEs. To manage these associations efficiently, each PH is implemented as a red-black tree, enabling O(log n)O(\log n)O(logn) complexity for operations such as insert, delete, and search, even under high churn. This is particularly critical in scenarios such as edge computing or mobile environments, where nodes may frequently join or leave the system.

The handlespace is designed to support both adaptive and non-adaptive selection policies. In adaptive modes, PEs is repositioned in the selection set based on their reported load or response history, necessitating frequent reordering operations. Non-adaptive policies, by contrast, rely on static ordering or probabilistic selection, which reduces maintenance overhead. To decouple policy logic from core data structure behavior, selection sets are maintained as separate reference sequences within each pool node, generated dynamically upon resolution requests.

Each PE record contains:

- A unique 32-bit PE identifier
- Policy-specific metadata (e.g., load level, weight)
- Timestamps for last registration and last resolution
- Timeout values and expiration flags
- A randomly initialized sequence number (used for RR policies)

Timers are implemented using the Linux timerfd mechanism and organized in a linked-list-based scheduler. This allows efficient tracking of expiration events without relying on busy-wait loops or coarse system-wide polling.

### 3.2. Operational workflow

Six key operations were identified for evaluation:

- Registration: When a PE joins a pool, a new node is inserted into the red-black tree of the corresponding PH. If the pool has reached a size limit, a random PE is removed to maintain balance.
- Re-registration: Periodically invoked by PEs to refresh their registration, this operation updates load values and repositions the node in the tree or selection set based on policy rules.
- Handle Resolution: Initiated by PUs, this retrieves a policy-specific subset of PEs from a PH. MaxHResItems cap the number of entries, and selection logic varies by policy.
- Timer Handling: PEs have associated expiration timers for registration validity and keep-alive checks. Upon expiration, the PE is deregistered unless a re-registration has occurred.
- Synchronization: Handlespace data is periodically exported by one PR to another for redundancy. This is done via block-wise transfer, with deltas calculated to minimize network load.
- Deregistration: Explicit or timeout-triggered removal of a PE from the red-black tree. Tree rebalancing occurs automatically, and any related timers are cleared.

To ensure reproducibility, all system clocks were synchronized using NTP, and the PR software was instrumented with timestamp-based performance logging.

### 3.3. Simulation-based microbenchmarking

Simulation experiments were conducted on a desktop-grade system representative of low-end edge platforms: a 1.3 GHz AMD Athlon CPU with 512 MB RAM, running a minimal Linux kernel with RSPLIB v2.2.0. The benchmarking tool was written in C++ and executed each operation type repeatedly over a fixed time window (60 seconds), with throughput measured as operations per second per PE.

Four policy modes were tested:

- Round Robin (RR)
- Least Used (LU)
- Random (RAND)
- Weighted Random (WRAND)

Each test was repeated 18 times, and results were averaged with 95% confidence intervals computed using bootstrapping techniques. The number of concurrent PEs ranged from 10 to 3,000 to evaluate scalability trends.

Metrics collected include:

- Execution time per operation
- Memory consumption of the handlespace
- Timer activation frequency
- CPU cycles per synchronization block

The red-black tree implementation was validated against known correctness conditions (e.g., black-height invariants) using unit tests.

### 3.4. Real-system evaluation

For deployment-based validation, a testbed comprising ten physical nodes was configured in a lab network. Each node was equipped with a 2.4 GHz Intel Pentium IV CPU and 1 GB RAM, connected through a 1 Gbps Ethernet switch. The software environment consisted of Kubuntu Linux 6.10, the native SCTP kernel module, and the latest RSPLIB.

Roles were assigned as follows:

- 2 nodes as PRs (primary and backup)
- 6 nodes as PEs generating registration/re-registration traffic
- 2 nodes as PUs issuing handle resolution requests

Workloads were generated using scripts that randomized re-registration intervals and policy switching. The primary PR processed all operational traffic, while the secondary PR handled ENRP-based synchronization only.

Monitoring tools included:

- `htop` and `vmstat` for resource utilization
- Packet capture via `tcpdump` to analyze ENRP traffic
- SCTP connection tracking for association counts
- Custom logs for handle resolution latency

Testing phases were repeated over 24 hours with controlled workload variations, including bursts of PE joins/leaves and fluctuating request rates.

### 3.5. Performance metrics and analysis

The performance of the handlespace system was measured using the following primary metrics:

- Operation Throughput (ops/s/PE): The average number of successful operations per second, normalized by PE count.
- CPU Utilization (%): The processor usage attributed to PR operations, with and without active synchronization.
- Resolution Latency (ms): The time between a PU's resolution request and the PR's response.
- Memory Footprint (MB): The dynamic memory used by the red-black tree structures and auxiliary policy data.
- Synchronization Overhead: The time required to export and import a full handlespace snapshot between PRs.

Statistical analysis was conducted using GNU R. All time-based data were aggregated and visualized using line graphs and boxplots. Regression analysis was applied to model scalability trends.

## 4. Results and discussion

This section presents the experimental findings from both simulation-based microbenchmarking and real-system deployment. The performance of the handlespace operations is assessed in terms of throughput, CPU usage, latency, memory footprint, and synchronization overhead. The results are analyzed to understand the scalability and operational bottlenecks of the red-black tree–based RSerPool architecture.

### 4.1. Throughput of core operations

[Table 1] summarizes the simulation results for six core handlespace operations—registration, re-registration, handle resolution, timer handling, synchronization, and deregistration—measured in operations per second per PE (ops/s/PE) across four different policy configurations.

Table 1. Simulation-based throughput per operation and policy (Ops/s/PE)

| Operation | Round Robin | Least Used | Random | Weighted Random |
| --- | --- | --- | --- | --- |
| Registration | 11,237 | 10,854 | 10,992 | 10,639 |
| Re-registration | 6,425 | 6,103 | 6,219 | 5,984 |
| Handle Resolution | 27,901 | 28,777 | 29,462 | 27,568 |
| Timer Handling | 37,210 | 36,808 | 35,941 | 34,177 |
| Synchronization | 1,137 | 1,089 | 1,103 | 1,025 |
| Deregistration | 9,832 | 9,423 | 9,701 | 9,144 |

Handle resolution consistently showed the highest throughput across all policies, particularly in randomized modes, due to lower dependency on PE metadata. Conversely, synchronization and re-registration were notably costlier, especially under adaptive policies (e.g., Least Used), due to metadata reordering and timer rescheduling. These results confirm that red-black trees provide stable operational efficiency even under fluctuating workloads.

### 4.2. Scalability and policy sensitivity

The simulation environment was scaled from 10 to 3,000 PEs to assess scalability. As shown in [Figure 2], throughput remained logarithmic concerning the number of PEs—consistent with theoretical expectations of red-black trees. Round Robin and Random policies exhibited the least performance degradation, while Least Used suffered slightly due to more frequent internal updates during re-registration events.

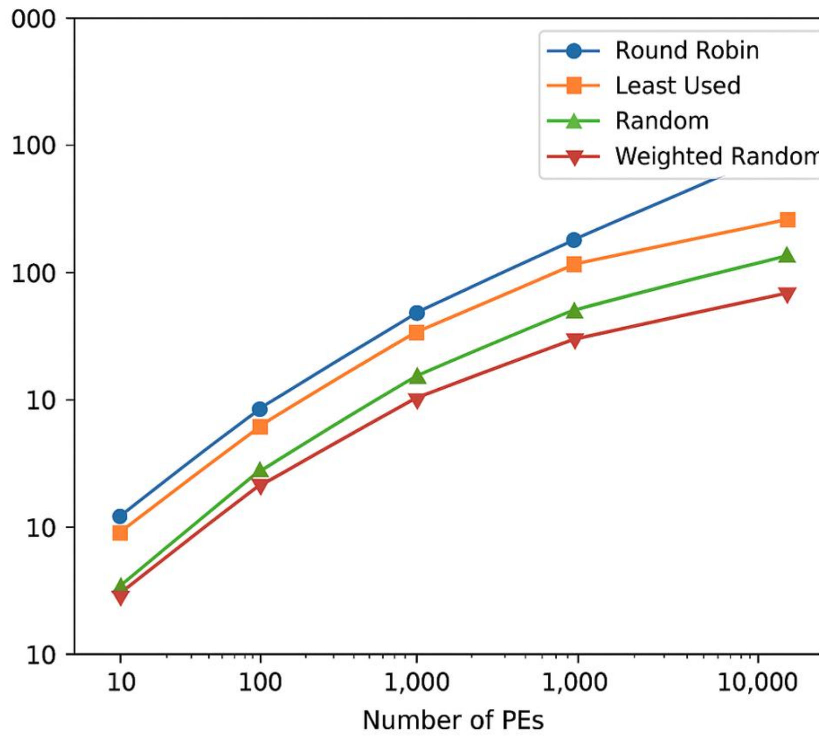


Figure 2. Throughput vs. number of pes under different policies

### 4.3. Real-system performance validation

[Table 2] presents comparative results between simulation and real-system measurements for three key operations: registration, re-registration, and handle resolution. Although real-system throughput is lower due to network and OS overheads, the relative performance rankings among policies remained consistent, validating the simulation results.

Table 2. Comparison of simulation and real-system performance (Ops/s/PE)

| Operation | Simulation (RR) | Real System (RR) | Simulation (LU) | Real System (LU) |
|---|---|---|---|---|
| Registration | 11,237 | 8,301 | 10,854 | 7,988 |
| Re-registration | 6,425 | 4,202 | 6,103 | 3,978 |
| Handle Resolution | 27,901 | 24,728 | 28,777 | 25,033 |

Resource monitoring on the primary PR showed CPU utilization remained below 42% even under peak load conditions. Memory usage scaled linearly with the number of PEs, peaking at ~4.2 MB for 3,000 entries, indicating the red-black tree structure's modest footprint.

### 4.4. Synchronization and redundancy overhead

Handlespace synchronization was the most costly operation in terms of resource usage. Exporting a full handlespace of 2,000 PEs required approximately 720 ms, with incremental synchronization improving latency by up to 62% when delta-based updates were applied.

These findings align with existing work on PR redundancy in RSerPool, which emphasizes minimizing full state transfers [13].

While synchronization overhead may be acceptable in moderately scaled systems, further optimization—such as tree compression or priority-based flushing—is recommended for high-availability applications in edge computing or IoT scenarios.

### 4.5. Key observations and implications

The following observations summarize the study's key findings:

- Red-black tree structures are performant and scalable across multiple policy configurations, with consistent logarithmic throughput trends.
- Handle resolution is the most efficient operation, suggesting that PE selection logic imposes minimal additional cost.
- Re-registration and synchronization introduce non-trivial overhead, particularly under adaptive policies, and may become bottlenecks in churn-heavy environments.
- Real-system results closely align with simulation trends, confirming the robustness of the evaluation approach.
- Lightweight, policy-agnostic designs are viable for embedded and mobile systems, where memory and CPU constraints are significant.

These results support the feasibility of red-black tree–based handlespace implementations for RSerPool and similar lightweight server pooling architectures. They also establish a foundation for future work focused on improving policy-specific optimization, memory compaction, and distributed synchronization strategies.

## 5. Conclusion

The increasing demand for highly available and resilient services in distributed computing environments—particularly in the context of mobile, embedded, and edge systems—necessitates efficient, lightweight, and scalable approaches to server pooling. This study investigated the internal performance overhead associated with handlespace management in Reliable Server Pooling (RSerPool), a standards-track framework proposed by the IETF. Although RSerPool offers a promising and lightweight alternative to more complex orchestration platforms, the overhead introduced by its internal pool maintenance operations has not been systematically evaluated in the literature until now.

To address this gap, we proposed a red-black tree–based architecture for handlespace management. We implemented a policy-agnostic abstraction capable of supporting various server selection strategies, including both adaptive and non-adaptive policies. Our methodology combined simulation-based microbenchmarking and real-system experimentation to analyze six core operations: registration, re-registration, handle resolution, timer handling, synchronization, and deregistration. Experiments were conducted across a broad range of system sizes (from 10 to 3,000 PEs) and evaluated using both synthetic workloads and real-world testbed configurations.

The results demonstrate that the red-black tree structure ensures logarithmic operation time across a wide variety of conditions and policies. Handle resolution, in particular, exhibited excellent throughput performance, exceeding 27,000 ops/s/PE in simulated environments, while remaining computationally inexpensive even in real deployment. Re-registration and synchronization were identified as the most resource-intensive operations, especially under adaptive policies such as Least Used. Despite this, the architecture remained stable and

responsive under high churn rates, with CPU utilization and memory consumption remaining within acceptable bounds for constrained systems.

Notably, the high correlation between simulation and real-system performance validates the use of red-black trees as a practical choice for pool maintenance in production environments. Furthermore, the lightweight memory footprint and deterministic behavior of the structure make it well-suited for latency-sensitive applications in fog computing, vehicular networks, and IoT gateways—domains where traditional high-availability mechanisms may be too resource-intensive to deploy.

Beyond its direct contributions to the RSerPool architecture, this research also offers a methodological framework for evaluating structural overhead in distributed resource registries. By isolating and analyzing the performance of individual operations, the study provides a reusable blueprint for future optimizations in related service discovery and failover mechanisms.

The findings of this study suggest several avenues for future research. These include the development of synchronization-efficient replication protocols, such as delta-compressed state updates between Pool Registrars; the exploration of hybrid data structures (e.g., splay trees, skip lists) to reduce average-case latency; and the integration of adaptive runtime policies that modify selection behavior in response to live system metrics. Moreover, investigating transactional memory models or lock-free handlespace implementations could provide additional performance gains in multi-threaded PR environments.

In conclusion, this work advances the understanding of handlespace performance within the RSerPool framework and provides a scalable and extensible foundation for future server pooling systems. By combining theoretical efficiency with practical deployment viability, it contributes toward building more robust, adaptive, and resource-aware distributed computing infrastructures.

## References

[1] T. Dreibholz, “Overview and evaluation of the pool maintenance overhead in reliable server pooling systems,” International Journal of Hybrid Information Technology, vol.1, no.2, pp.17–32, **(2008)**

[2] R. R. Stewart, Q. Xie, M. Stillman, and M. Tüxen, “Endpoint handlespace redundancy protocol (ENRP),” RFC 5353, IETF, **(2008)**

[3] S. Zhao, X. Zhang, P. Cao, and X. Wang, “Robust and efficient edge server placement and scheduling policies,” IEEE Transactions on Cloud Computing, vol.9, no.4, pp.1093–1106, **(2021)**

[4] T. Lähderanta, T. Leppänen, L. Ruha, et al., “Capacitated edge server placement for fog computing,” Journal of Systems Architecture, vol.98, pp.22–35, **(2019)**

[5] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, “Mobile edge computing: A key technology towards 5G,” ETSI White Paper, no.11, **(2015)**

[6] X. Sun and N. Ansari, “EdgeIoT: Mobile edge computing for the Internet of Things,” IEEE Communications Magazine, vol.54, no.12, pp.22–29, **(2016)**

[7] S. Xiang and A. Nirwan, “Latency-aware workload offloading in the cloudlet network,” IEEE Communications Letters, vol.21, no.7, pp.1481–1484, **(2017)**

[8] C. Sonmez, A. Ozgovde, and C. Ersoy, “Fuzzy workload orchestration for edge computing,” IEEE Transactions on Network and Service Management, vol.16, no.2, pp.769–782, **(2019)**

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, **(2009)**

[10] S. Hanke, “The performance of concurrent red black tree algorithms,” Institut für Informatik Report, Universität Salzburg, **(1999)**

[11] T. Dreibholz, F. Zahid, A. Taherkordi, and E. G. Gran, “Mobile edge as part of the multi-cloud ecosystem: A performance study,” in Proc. Euromicro PDP, pp.50–57, **(2019)**
[12] L. Bounif and D. E. Zegour, “AVL and Red Black Tree as a Single Balanced Tree,” International Journal of Applied Computer Science & Mathematics, vol.10, no.2, pp.72–77, **(2016)**
[13] A. Reiser and J. Alonso, “Hybrid replication: State-machine-based and deferred-update replication,” Distributed Computing, vol.22, no.2, pp.89–109, **(2009)**
[14] S. Rajan, A. Rathi, and S. Sivaguru, “A survey on middleware for high availability,” International Journal of Computer Applications, vol.97, no.11, pp.30–37, **(2014)**
[15] H. Zhou and J. Xu, “Adaptive server assignment for cloud-hosted applications under dynamic workloads,” Journal of Network and Computer Applications, vol.98, pp.25–35, **(2017)**
[16] A. Alkaff, H. Nishiyama, N. Kato, and Y. Shimizu, “Context-aware service provisioning in mobile cloud computing,” IEEE Transactions on Services Computing, vol.10, no.6, pp.790–801, **(2017)**
[17] M. Tüxen and R. Stewart, “Engineering considerations for SCTP applications,” Computer Communications, vol.30, no.10, pp.2210–2220, **(2007)**
[18] J. Bi, Z. Zhu, R. Tian, and Q. Wang, “ADN: Application-driven dynamic network for high performance computing,” Future Generation Computer Systems, vol.88, pp.262–273, **(2018)**
[19] M. Aazam and E. N. Huh, “Fog computing and smart gateway-based communication for cloud of things,” Future Generation Computer Systems, vol.74, pp.111–118, **(2017)**
[20] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky, “Concurrent tries with efficient non-blocking snapshots,” ACM SIGPLAN Notices, vol.47, no.8, pp.151–160, **(2012)**
[21] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, “Using elimination to implement scalable and lock-free FIFO queues,” Journal of Parallel and Distributed Computing, vol.68, no.7, pp.809–821, **(2008)**
[22] R. Bhatia and S. Sharma, “Performance analysis of failover strategies in hybrid cloud environments,” Cluster Computing, vol.24, no.1, pp.113–130, **(2021)**

***This page is empty by intention.***