Technical Note



# A Scale-Up Optimized Layer7 Proxy Architecture with Deterministic Session Control and Logical Slicing

Daisuke Sugisawa *

Posted Date: 21 January 2026

doi: 10.20944/preprints202510.2163.v2

Keywords: Layer7 load balancer; reverse TCP flow; deterministic QPS; session control; scalability; NGINX

 doi:10.20944/preprints202510.2163.v2



*Technical Note*

# A Scale-Up Optimized Layer7 Proxy Architecture with Deterministic Session Control and Logical Slicing

**Daisuke Sugisawa**

Xander, LLC. Shibuya, Tokyo, Japan; daisuke.sugisawa.xander@gmail.com

**Abstract**

This paper proposes a bidirectional Layer7 Proxy architecture using NGINX and a minimal Acceptor module. In conventional Layer7 load balancers, NGINX issues connect() to the backend for each request, making context switches between kernel and user space structurally unavoidable due to TCP handshakes. In the proposed approach, service modules register sockets with completed TCP handshakes to the Acceptor in advance, and NGINX receives the file descriptors via UNIX domain sockets. This eliminates connect() calls and skips per-request TCP handshakes. Performance evaluation demonstrates that the proposed method directly translates context switch reduction into throughput (RPS) improvement and latency reduction. While the conventional method shows no correlation between context switches and RPS, the proposed method enables context switches to function as a "controllable performance parameter," forming the foundation for deterministic throughput control. Additionally, the Acceptor queue functions as a buffer, structurally limiting requests exceeding the service module's processing capacity during traffic spikes, thereby avoiding non-linear performance degradation. This approach enables dynamic reconfiguration and graceful restarts through socket file descriptor passing, without relying on health checks or frequent configuration reloads. It presents a new design guideline that achieves deterministic throughput control at the application layer, complementing scale-out dependent resource operations in cloud environments.

**Keywords:** Layer7 load balancer; reverse TCP flow; deterministic QPS; session control; scalability; NGINX

---

## 1. Introduction

In modern large-scale and dynamic web service environments, traditional L7 load balancers face increasing difficulty in sustaining predictable throughput due to TCP port exhaustion and connection imbalance between slow and fast terminals (e.g., smartphone applications).

To address these issues, we propose a bidirectional Layer7 Proxy architecture. In this architecture, service modules register sockets with completed TCP handshakes to the Acceptor in advance, and NGINX receives the file descriptors via UNIX domain sockets. This eliminates connect() calls from NGINX. This allows applications to share CPU cores efficiently and to achieve deterministic, spike-tolerant behavior without resorting to heavy health checks or frequent configuration reloads. The main contributions of this study are as follows:

- Proposal of an **architectural paradigm** that shifts TCP socket establishment to the service module side and eliminates connect() from NGINX
- **Conceptual definition and structural guarantee of deterministic QPS** based on the number of sessions registered by service modules
- **Design-level elimination** of indeterminacy factors inherent in conventional methods (connect() timing, TIME_WAIT accumulation, kernel FD distribution)
- Scale-up optimization guidelines that can be **combined with** cloud environment scale-out

 doi:10.20944/preprints202510.2163.v2



## 2. System Architecture

### *2.1. Overview*

As shown in Figure 1, the system consists of:

- **NGINX patched module** with capability to receive duplicated socket descriptors
- **Acceptor module** that transfers active connections via UNIX domain sockets
- **Service modules** that perform request handling as independent processes

This enables bidirectional transfer of TCP session ownership between processes. (See Figure 2: Layer7 Proxy Characteristics, Figure 3: Specific Control)

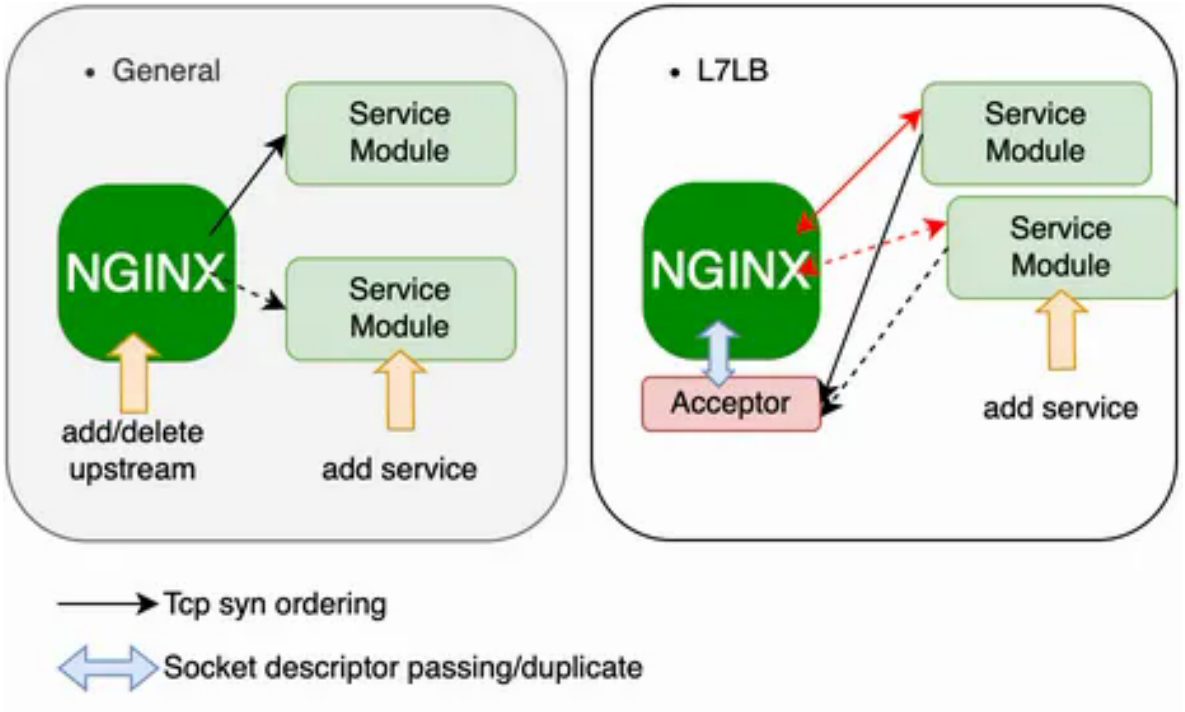


**Figure 1.** Layer7 Proxy Overview.

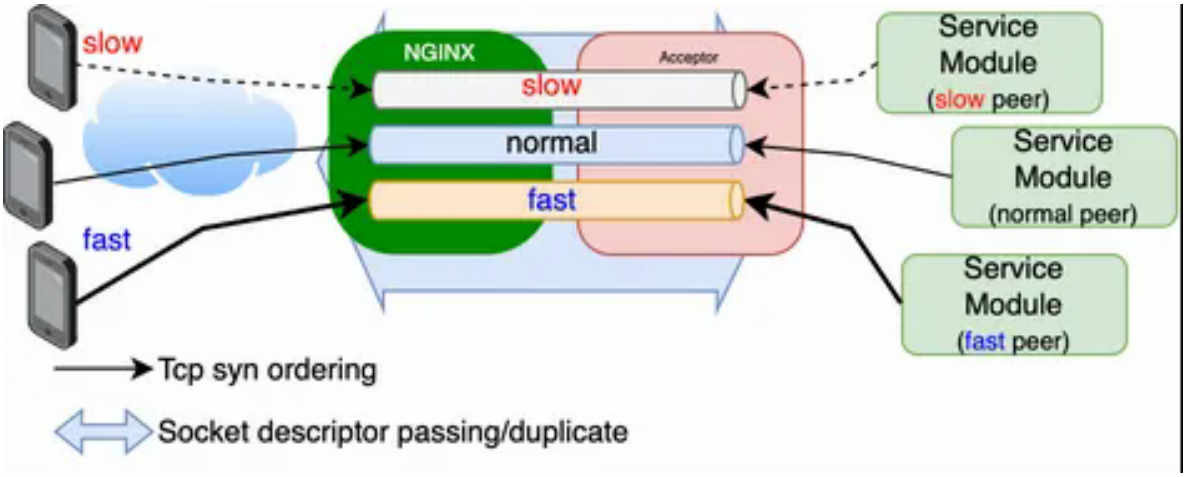


**Figure 2.** Layer7 Proxy Characteristics.

 doi:10.20944/preprints202510.2163.v2



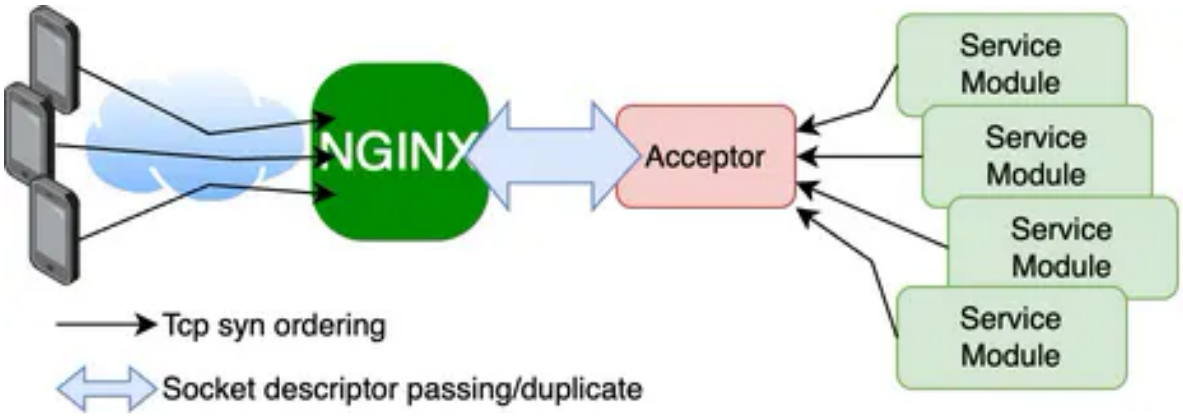


**Figure 3.** Layer7 Proxy Specific Control.

### *2.2. Advantages over Conventional L7LB*

The proposed approach addresses several long-standing issues:

- Port exhaustion mitigation by logically separating (slicing) session processing based on terminal characteristics, suppressing local port occupation by slow terminals
- Dynamic and configurable L7 load balancing without restarts
- No health-check overhead
- Graceful restart support via socket FD passing
- Reduced number of required application service processes

Auto-scaling, which has become common in cloud-native environments, is a powerful mechanism that automates resource scaling in response to traffic fluctuations. The proposed method, when combined with auto-scaling, improves spike resistance and resource predictability through deterministic session control at the application layer, enabling more efficient resource operation.

## 3. Implementation

### *3.1. NGINX Modification*

A lightweight patch adds NGX_X_BOTH_PROXY support to the NGINX core (nginx.patch), allowing it to exchange socket descriptors via UNIX domain sockets (/tmp/Xaccepted_socket). This modification enables bidirectional proxying between NGINX and the Acceptor module.

**(See Appendix A.1)**

### *3.2. Acceptor Module*

The Acceptor component manages sockets registered from service modules and provides socket FDs to NGINX upon request. It utilizes UNIX domain sockets to manage socket descriptor passing (sendmsg/recvmsg) to NGINX. The Acceptor maintains fairness across multiple service modules and dynamically rebuilds connection queues without terminating active sessions.

**(See Appendix A.2)**

### *3.3. Example Service Module*

A lightweight test service implemented in Go repeatedly establishes bidirectional connections with the proxy and serves HTTP responses. This validates the proxy's resilience under multi-core, multi-threaded concurrent stress.

**(See Appendix A.5)**.

## 4. Performance Evaluation

This section evaluates the performance characteristics of the proposed method in a loopback environment. The evaluation items are linear scaling characteristics with respect to CPU core count, and

 doi:10.20944/preprints202510.2163.v2



the relationship between context switches, throughput, and latency in comparison with conventional methods.

**Table 1.** Measurement Environment.

| Item | Specification |
|---|---|
| Machine | MacBook Pro (Mac15,8) |
| Chip | Apple M3 Max |
| Core Count | 16 (P: 12, E: 4) |
| Memory | 64 GB |
| NGINX | 1.16.0 |
| worker_processes | 1 |
| worker_connections | 1024 |
| Benchmark Tool | ApacheBench (ab) |
| CS Measurement Command | top -stats csw -pid <PID> |
| CS Measurement Target | NGINX worker process (fixed PID) |
| CS Type | Total (voluntary + involuntary) |
| Sample Interval | 1 second |

**Table 2.** NGINX Build Options.

```
./configure -with-debug -prefix=../nginx
-with-cc-opt="-I/opt/homebrew/opt/pcre/include"
-with-ld-opt="-L/opt/homebrew/opt/pcre/lib"
```

Performance scaling tests confirm linear throughput growth with CPU core count (Figure 4).

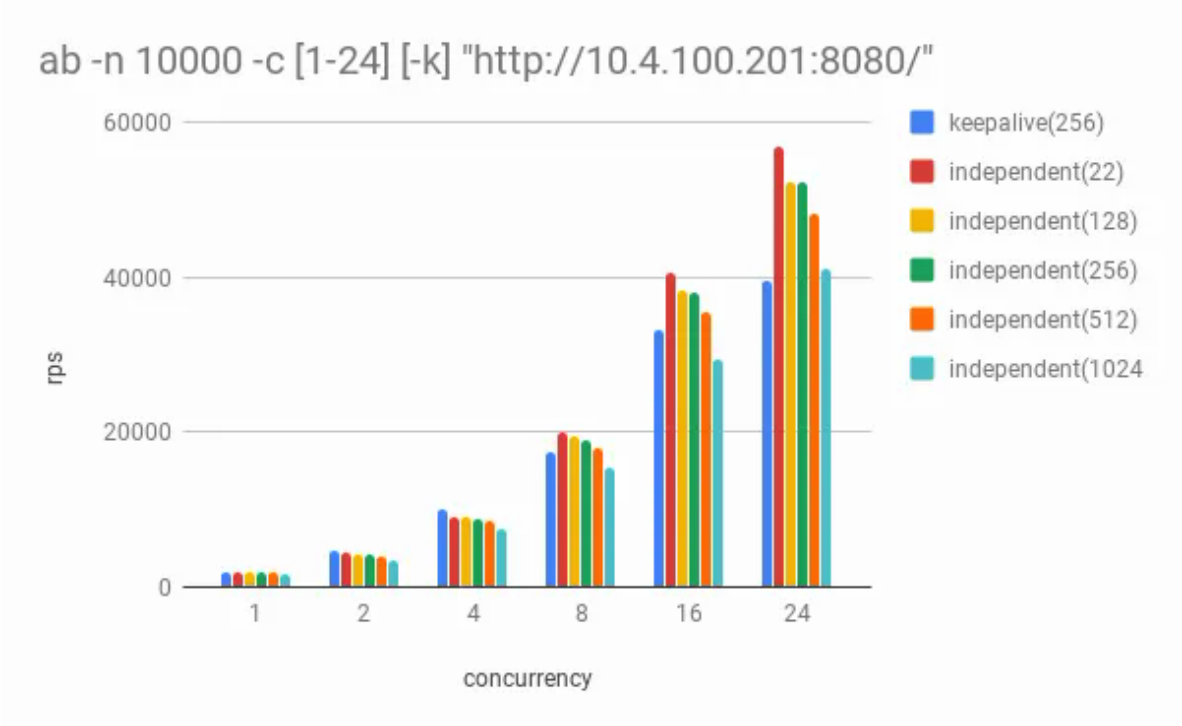


**Figure 4.** Layer7 Proxy Performance by Cores.

htop analysis (Figure 5) shows distributed CPU utilization across service modules, indicating effective socket-level load distribution.

 doi:10.20944/preprints202510.2163.v2




**Figure 5.** Layer7 Proxy Performance htop

*4.1. Comparison with Conventional Method*

4.1.1. Context Switches and RPS

Figure 6 shows the relationship between context switch count and RPS (Requests per Second) with respect to concurrency. This measurement reveals the essential difference between the proposed and conventional methods.

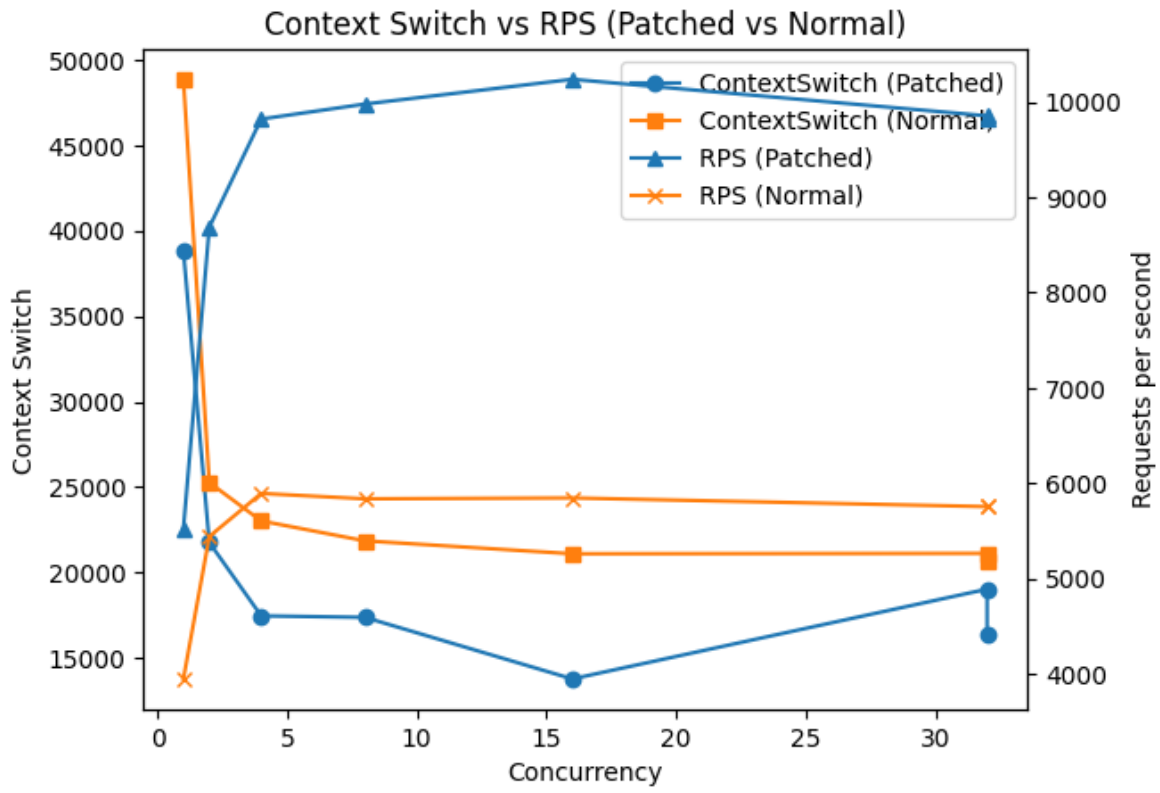


**Figure 6.** ContextSwitch x RPS.

In the **conventional method (Normal nginx)**, context switch count fluctuates between 20k and 50k, but RPS plateaus at approximately 5.7k req/s, showing no correlation between context switch variation and RPS. This indicates that in the conventional method, context switches occur merely as overhead and do not contribute to throughput improvement.

In contrast, in the **proposed method (Patched nginx)**, context switches are minimized (approximately 14k) in the Concurrency 8–16 range, at which point RPS reaches maximum (approximately

 doi:10.20944/preprints202510.2163.v2



10k req/s). Furthermore, even when context switches increase at Concurrency 32, RPS degradation remains minimal.

This result demonstrates a fundamental difference in causal structure between the two methods. In the conventional method, NGINX issues connect() to the backend for each request, making context switches between kernel and user space structurally unavoidable due to TCP handshakes (SYN/SYN-ACK/ACK). Therefore, no correlation emerges between CS and RPS.

In contrast, in the proposed method, service modules register sockets with completed TCP handshakes to the Acceptor in advance, and NGINX receives the file descriptors via UNIX domain sockets. Since the connect() call itself is eliminated, per-request TCP handshakes are skipped, and the associated context switches are structurally reduced. As a result, CS reduction directly contributes to improved execution efficiency and manifests as increased RPS. In other words, in this proposal, context switches function as a "controllable performance parameter," forming the foundation for deterministic throughput control.

#### 4.1.2. Context Switches and Request Time

Figure 7 shows the relationship between context switch count and Time per Request with respect to concurrency. While RPS in the previous section represents overall system throughput, this measurement reveals the response characteristics of individual requests.

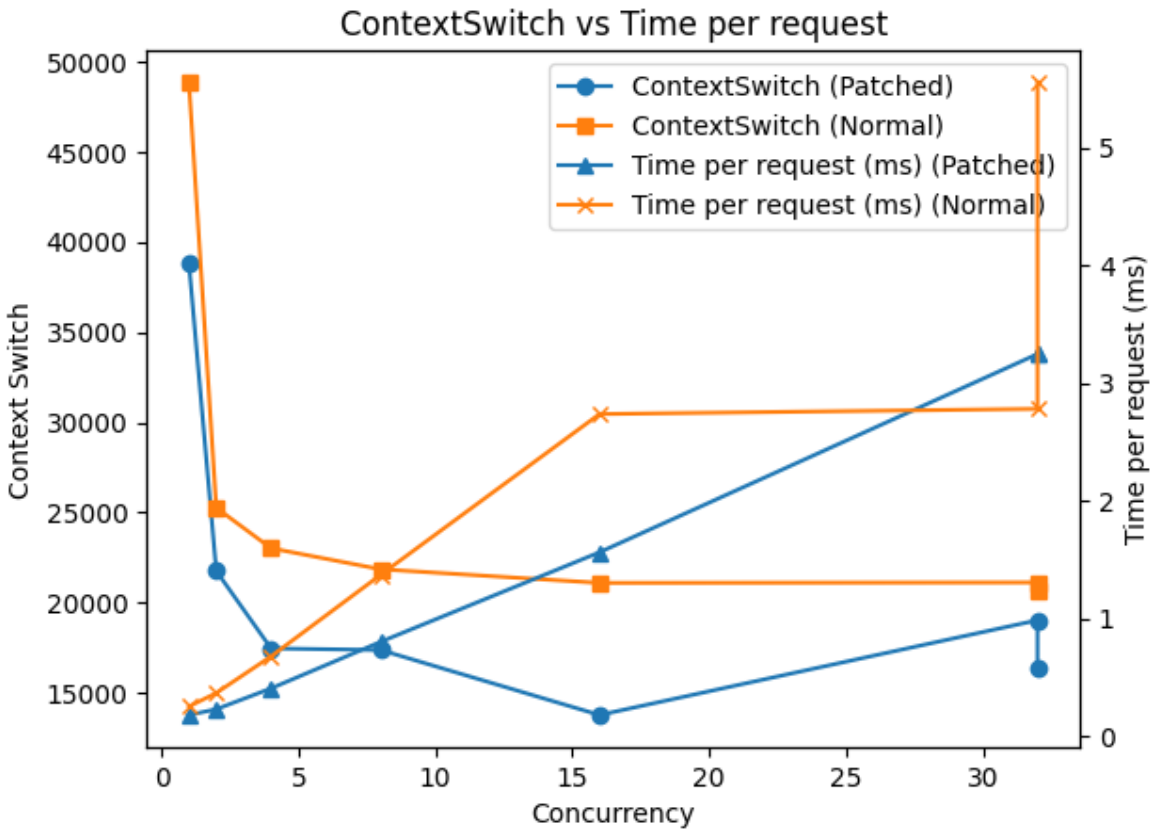


**Figure 7.** ContextSwitch x TPR.

In the **conventional method (Normal nginx)**, while context switch count remains high, Time per Request increases sharply in proportion to concurrency, exceeding 5ms at Concurrency 32. This suggests that the majority of request processing time is occupied by wait time originating from the kernel scheduler.

In contrast, in the **proposed method (Patched nginx)**, Time per Request is minimized in the region where context switches are minimized, and increases nearly linearly (slope approximately

 doi:10.20944/preprints202510.2163.v2



0.1ms/concurrency) with increasing concurrency. This gradual increase indicates that wait time originates from actual processing rather than the scheduler.

This result reveals the **qualitative difference in latency** between the two methods. In the conventional method, because context switches are not reduced, the dominant factor of latency is scheduler wait, causing non-linear degradation with increasing concurrency. In contrast, in the proposed method, the main component of latency is actual processing time, maintaining predictable linear response even under increasing load.

The important point is not that "it became faster because context switches decreased," but that **"by changing to an architecture that can structurally reduce context switches, the nature of latency itself has changed."** This structural change is the source of deterministic latency characteristics.

### *4.2. Structural Advantages*

In addition to the performance differences shown in the previous section, this proposal has structural advantages under the following conditions:

- **During traffic spikes:** In conventional methods, NGINX issues connect() to the backend for each request and TCP handshakes occur, causing connect() wait to accumulate during spikes and throughput to degrade non-linearly. In this proposal, since handshake-completed sockets are received via UNIX domain sockets, connect() is unnecessary, and the Acceptor queue functions as a buffer, structurally limiting requests exceeding the service module's processing capacity.
- **When slow clients are mixed:** In conventional methods, slow clients occupy NGINX's local ports for extended periods, potentially reducing connection acceptance capacity for all clients due to port exhaustion. In this proposal, since session ownership is on the service module side, NGINX's port consumption is limited to the service module's concurrent processing capacity.
- **TIME_WAIT accumulation during long-term operation:** In conventional methods, the NGINX process repeatedly calling connect()/close() causes TIME_WAIT socket accumulation, potentially causing ephemeral port exhaustion. In this proposal, since connect() calls from the NGINX process are eliminated, this problem is structurally avoided.

## 5. Discussion

### *5.1. io-uring*

Recent advances in the Linux kernel introduced **io_uring**, which shares submission and completion rings between user space and the kernel, thereby reducing system call overhead, data copies, and context switches. In networking, several implementations and studies have demonstrated zero-copy RX and high-speed receive paths using this mechanism [1], making it a promising approach for data-plane optimization.

The proposed bidirectional Layer-7 Proxy Load Balancer (L7LB) is orthogonal yet complementary to **io_uring**. While **io_uring** focuses on improving I/O path efficiency in the data plane, the L7LB shifts TCP socket establishment to the service module side in the control plane, eliminating connect() from NGINX. That is, service modules register handshake-completed sockets to the Acceptor in advance, and Layer-7 requests are dispatched strictly within those predetermined bounds. As a result, the QPS becomes inherently governed by the declared concurrency (*Deterministic QPS*), providing predictable throughput behavior.

These two approaches can be combined to achieve both deterministic scheduling and low-overhead I/O. For example, using `io_uring` on the Acceptor or service side would enable deterministic slot allocation at the control layer, while maintaining zero-copy and efficient I/O operations in the data layer.

### *5.2. Deterministic QPS and Spike Resistance*

Unlike conventional L7LBs that issue connect() for each inbound TCP initiation, the proposed method has service modules register sockets in advance, so connection bursts are absorbed by Acceptor

 doi:10.20944/preprints202510.2163.v2



queues before reaching service modules. This decouples request inflow from application readiness, allowing stable deterministic QPS behavior even under spikes. Fig. 8 shows details of the Acceptor queue.

### *5.3. Definition and Analysis of Deterministic QPS*

Deterministic QPS refers to request-per-second behavior that is completely governed by the direction of TCP initiation, that is, by the internal concurrency of the service application itself. Unlike conventional Layer-7 load balancers where TCP SYN packets are initiated by clients and passively accepted by the backend application, the proposed architecture allows the service application to actively register its own parallel TCP sessions to the Acceptor module. **Because each service process knows its own concurrency capacity**, the total number of registered TCP sessions can precisely reflect the system's actual processing potential.

#### 5.3.1. Mechanism

Through this architecture, each service process proactively registers a number of handshake-completed sockets equivalent to its parallel processing capability to the Acceptor. Then, NGINX (proxy) dispatches requests only within the limited pool of sessions registered in the Acceptor. Consequently:

- The application never receives more simultaneous requests than it can process.
- The proxy never accumulates excessive pending ACCEPT/LISTEN queues, even under burst traffic.
- connect() from NGINX becomes unnecessary, eliminating the round-trip wait time of TCP handshakes (SYN→SYN-ACK→ACK).
- QPS becomes deterministic and directly measurable as a function of the application's concurrency configuration.

#### 5.3.2. Analysis

This deterministic coupling between session allocation and known service concurrency enables linear scalability with respect to CPU-core count (as demonstrated in Fig. 4). By decoupling connection inflow from instantaneous application latency, the proxy-acceptor pair forms a predictable control surface for throughput regulation. Transient service delays no longer propagate back into the proxy's socket backlog; instead, they are locally absorbed by controlled Acceptor queues. As a result, the system achieves predictable throughput, queue-bounded stability, and graceful degradation even under severe traffic spikes—characteristics that are unattainable in traditional client-initiated TCP flows.

#### 5.3.3. Difference from Conventional Concurrency Limits

Deterministic QPS appears superficially similar to fixed concurrency limits in conventional proxies, but is fundamentally different.

Conventional concurrency limits are values that the proxy side **estimates and statically sets** based on guessing the backend's processing capacity. However, when there are additional services or databases beyond the backend server, they also have their own concurrency limits, and **the true processing capacity is unknowable from the proxy side**. This gap between the configured value and actual capacity causes non-deterministic behavior under overload.

In contrast, in this proposal, the service module itself **dynamically registers** "the number of sessions it can currently process" to the Acceptor. Since services can adjust the registration count based on their internal state (including the load status of dependencies), the number of registered sessions **can accurately reflect the actual processing capacity at that point in time**. Furthermore, since NGINX does not call connect() and only uses FDs provided by the Acceptor, when the limit is exceeded, there is "no FD available for allocation," structurally preventing overload propagation.

 doi:10.20944/preprints202510.2163.v2



In other words, while the conventional method is "static setting of estimated values by the proxy side," this proposal fundamentally differs in achieving "dynamic declaration of actual capacity by the service side."

### *5.4. Port Exhaustion Mitigation*

Port Exhaustion Mitigation refers to a mechanism that optimizes port-resource utilization within the proxy layer by decoupling upstream and downstream TCP session lifecycles. In conventional architectures, the proxy immediately terminates client connections and, for every inbound SYN, establishes a corresponding upstream TCP session toward the service application—irrespective of the application's processing latency or state. This leads to unnecessary socket allocation and can result in port depletion when a large fraction of clients are slow or long-lived.

#### 5.4.1. Problem Context

Under such traditional TCP-upstream models:

- Slow or idle clients occupy proxy ports for extended periods.
- Application-side and client-side latency indirectly causes TIME_WAIT and CLOSE_WAIT accumulation in the proxy.
- Eventually, ephemeral-port exhaustion limits connection acceptance capacity and induces cascading failures.

#### 5.4.2. Proposed Mechanism

In the bidirectional Layer-7 Proxy introduced in this paper, the NGINX process maintains independent management of downstream (client-side) and upstream (service application-side) sessions. A new upstream connection toward the service application is acquired by NGINX only upon explicit allocation from the Acceptor. This pull-based session creation model confines the proxy's port usage strictly within the active concurrency limits defined by the service application itself.

#### 5.4.3. Analysis

By making the application the source of truth for permissible concurrent sessions, the proxy's port utilization becomes stable and predictable regardless of client speed or traffic variability. This architectural decoupling yields several tangible benefits:

- **Stable port consumption:** proportional to declared backend concurrency rather than client volume.
- **Reduced TIME_WAIT/CLOSE_WAIT pressure:** preventing socket-table inflation during long uptimes.
- **Improved multi-service coexistence:** isolation between independent service modules avoids cross-service port contention.

#### 5.4.4. Considerations

The mitigation mechanism attains its maximum effect when combined with the deterministic QPS framework described above. Together, they form a two-layer cooperative control model in which applications define the upper bound of concurrency, and the proxy enforces efficient utilization of network and port resources. This synergy ensures long-term stability and scalability of Layer-7 load-balancing infrastructures under high-density, latency-variable client populations.

### *5.5. Session Fairness and Multi-Core Affinity*

By using shared queues (Figure 8: Acceptor Ring), the system ensures fair distribution among service modules, preventing starvation and improving CPU cache locality.

 doi:10.20944/preprints202510.2163.v2



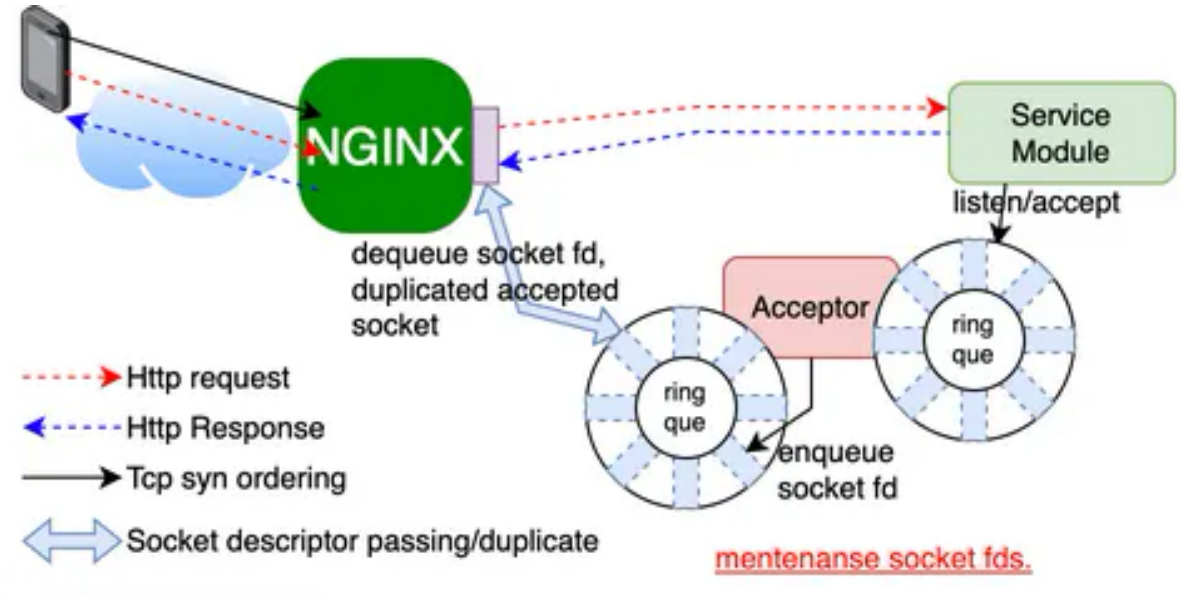


**Figure 8.** Layer7 Proxy Acceptor Ring.

## 6. Comparison to Conventional Architectures

Table 3 shows a comparison of characteristics, but the essence of this proposal lies in the structural elimination of indeterminacy factors inherent in conventional methods. Table 4 shows the indeterminacy factors in conventional methods and how they are eliminated in this proposal.

**Table 3.** Traditional L7LB vs Proposed Layer7 Proxy.

| **Feature** | **Traditional L7LB** | **Proposed Layer7 Proxy** |
|---|---|---|
| TCP Direction | Client → Server | Server ↔ Acceptor |
| Upstream Port Exhaustion | Frequent | Suppressed |
| Health Checks | Required | Not required |
| Session Fairness | Limited | Deterministic |
| Restart Overhead | High | Graceful |
| QPS Determinism | Low | High |

**Table 4.** Indeterminacy Factors in Conventional Methods and Elimination in This Proposal.

| **Indeterminacy Factor** | **Occurrence Mechanism in Conventional Method** | **Elimination Method in This Proposal** |
|---|---|---|
| connect() completion timing | Depends on backend load; wait time accumulates during spikes | Eliminates connect() call itself from NGINX |
| TIME_WAIT accumulation | Socket state accumulates as NGINX repeatedly calls connect()/close() | Avoids TIME_WAIT generation by eliminating connect() from NGINX |
| Kernel FD distribution | OS-dependent scheduling such as SO_REUSEPORT | Deterministic control in user space by Acceptor |
| Client-induced port occupation | Long-term connections from slow terminals occupy NGINX's local ports | Transfers session ownership to service module side |

### *6.1. Comparison with Upstream Keepalive*

NGINX's upstream keepalive is a mechanism that maintains and reuses TCP connections to specific upstream servers. However, in environments that assume backend distribution by load balancers or per-request routing, the destinations for reusing maintained connections often do not

 doi:10.20944/preprints202510.2163.v2



match, and consequently the problems of connect() calls and TIME_WAIT accumulation are not resolved.

In the proposed method, service modules declaratively register "connections as processing capacity" to the Acceptor, and NGINX receives file descriptors before upstream selection. In other words, while keepalive is server-centric connection reuse, this method is capacity-centric resource provision, fundamentally different in that the judgment of connection reusability itself does not exist.

### *6.2. Comparison with Katran*

Katran (Meta, 2018) [2] is a software load balancer implementing traditional 5-tuple Layer-4 switching using XDP/eBPF, focusing on transport-level scalability and throughput. While Katran efficiently handles massive connection distribution, it does not address application-level determinism or latency-aware dispatching. Our work complements such L4 systems by providing deterministic, latency-adaptive session control at Layer-7, where real-time response characteristics directly influence load balancing decisions.

### *6.3. Relation to L4 Load Balancers (e.g., Maglev)*

Maglev [3] represents a state-of-the-art Layer-4 software load balancer, achieving scalable and reliable packet-level distribution through consistent hashing and connection tracking. Operating entirely at the transport layer, Maglev efficiently balances TCP and UDP flows across backend servers but does not provide application-level visibility or adaptive session control. Its design prioritizes throughput and fault tolerance rather than connection fairness or latency adaptation.

In contrast, the proposed Bidirectional Layer-7 Proxy Load Balancer (L7LB) is positioned as an upper-layer complement to such L4 systems. After a Layer-4 load balancer (e.g., Maglev) distributes flows, the L7LB operates at the application layer to enforce deterministic per-session fairness, mitigate RTT-induced imbalance, and absorb excessive SYN or connection bursts through bidirectional session control. This architecture bridges the gap between transport-level scalability and application-level stability, ensuring predictable QPS behavior under high concurrency without kernel-level dependencies.

## 7. Conclusion

As shown in the sequence in Figure 9, in this method, connect() calls from the NGINX process are completely eliminated, and TCP socket creation and allocation are performed within the Acceptor module. That is, session ownership owner(FD) always resides in the user-space service module, and NGINX does not create new sockets. This removes the "kernel-scheduler-dependent FD distribution" seen in conventional accept-thread or SO_REUSEPORT approaches. In conventional thread pool or queue separation approaches, there is still a need to judge FD generation possibility or connection validity, which becomes the starting point of non-determinism. This method structurally removes this judgment process by eliminating connect() calls from the NGINX process, guaranteeing deterministic QPS.

This paper introduced a bidirectional Layer7 Proxy Load Balancer leveraging NGINX with minimal patching and an Acceptor coordination module. This architecture eliminates connect() calls from NGINX, achieving deterministic QPS, spike resilience, and graceful scaling across CPU cores.

It offers a practical and implementable enhancement to standard NGINX deployments for applications requiring high determinism and minimal latency under variable client conditions. [1] In simple terms, it is the structural removal of congestion causes by "making the nginx (proxy) process not call connect().".

[1] https://nginx.org/download/nginx-1.16.0.tar.gz

 doi:10.20944/preprints202510.2163.v2



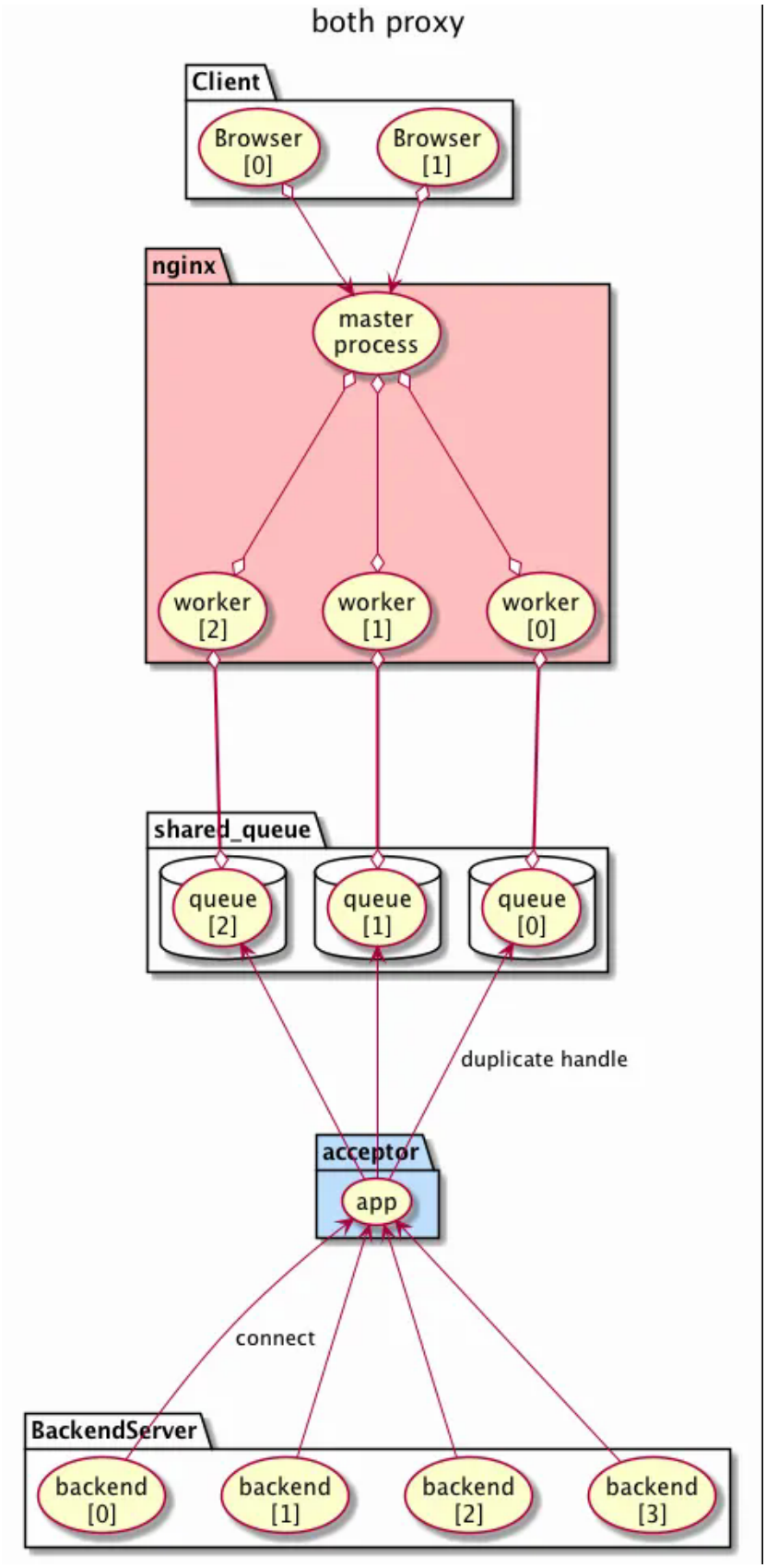


**Figure 9.** Layer7 Proxy Payload Sequence.

*7.1. Future Work*

This paper quantitatively demonstrated linear scalability and context switch reduction effects in steady state, but experimental verification of the essential advantage of the proposed method—

 doi:10.20944/preprints202510.2163.v2



"deterministic behavior under conditions where conventional methods break down"—has not been achieved.

The performance evaluation in this paper is an approximate evaluation on a loopback interface, and the following comparative experiments via actual NICs are listed as future work:

- Comparison of connect() wait accumulation vs Acceptor queue absorption under spike load
- Comparison of port occupation and slow/fast interference when slow clients are mixed
- Time-series comparison of TIME_WAIT/CLOSE_WAIT transitions during long-term operation

In the current Linux kernel, for normal NIC reception without using Poll Mode Driver, NAPI is scheduled by hardware interrupts triggered by packet arrival, and then packet processing is performed in softirq context. Therefore, compared to loopback interfaces where processing completes within the transmission context without execution context transitions due to interrupts and softirq, the difference in execution overhead is expected to become more pronounced.

**Author Contributions:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing-Original Draft, Writing-Review and Editing, Visualization, Supervision, Project administration: Y.Sugisawa, D.Sugisawa.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflicts of interest.

**Use of Artificial Intelligence:** The authors used a large language model ChatGPT, OpenAI to assist in proofreading, grammar checking, and improving the clarity of expressions. The authors reviewed and are responsible for the final content.

## Appendix A. Implementation Code

The implementation code for this proposal is available at the following GitHub repository.

- **Repository:** `https://github.com/xander-jp/mxl7lb`[2]

An overview of the main source files is provided below.

### *Appendix A.1. nginx.patch*

A patch to the NGINX core (`src/event/ngx_event_connect.c`) that adds file descriptor passing functionality via UNIX domain sockets using the `NGX_X_BOTH_PROXY` macro. It bypasses conventional `connect()` calls and directly uses socket FDs received from the Acceptor, enabling deterministic session control.

### *Appendix A.2. acceptor/*

The core implementation of the Acceptor module (C++), which redistributes TCP sessions registered from service modules to NGINX in a round-robin manner. It performs FD transfer using SCM_RIGHTS via `sendmsg`/`recvmsg` and thread-safe queue management. Main files include `main.cc`, `acceptor_sendfd.cc`, `queue.h`, etc.

### *Appendix A.3. acceptor_cli/*

A CLI tool (C++) that controls route add/del/list to the Acceptor via shared memory. For example, `./acceptor_cli add 192.168.0.0 24` is a configuration that permits FD transfer to NGINX for client sessions from 192.168.0.0/24.

[2] https://github.com/xander-jp/mxl7lb

 doi:10.20944/preprints202510.2163.v2



*Appendix A.4. natlibs/*

An IPC communication library using shared memory (C++), used for passing configuration and statistics information between the Acceptor and CLI tools. Main files include `nat_shared_memory_config.cc` (configuration management), `nat_shared_memory_statistics.cc` (statistics management), etc.

*Appendix A.5. client/client.go*

A test service module implemented in Go, where 32 parallel goroutines maintain bidirectional connections with the proxy while returning HTTP responses. Used for proxy resilience verification in multi-core environments.

*Appendix A.6. client_large_resp.go*

A test client using HAProxy protocol (PROXY protocol), which creates multiple concurrent connections (default 32 connections) and sends HTTP responses with variable-length payloads. The `-s` flag controls the number of services, and the `-l` flag controls the payload length.

*Appendix A.7. client_large_resp_server.go*

A lightweight HTTP server for testing, which returns dynamically generated HTML responses to requests. By default, it listens on port 4200, and the address can be changed with the `-a` flag. Used as a backend server during performance measurements.

## References

1. Neal Cardwell et al.: Fast ZC Rx Data Plane using io_uring, Netdev 0x17 (2023). https://netdevconf.info/0x17/docs/netdev-0x17-paper24-talk-paper.pdf
2. A. Agarwal, A. Nikolaev, D. Rybkin, *et al.*, Katran: A Scalable Layer-4 Load Balancer using XDP and eBPF, Meta (Facebook) Engineering Blog, May 2018. Available at: https://engineering.fb.com/2018/05/22/open-source/katran-a-scalable-network-load-balancer/. Open source implementation: https://github.com/facebookincubator/katran
3. D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, Maglev: A Fast and Reliable Software Network Load Balancer, *USENIX NSDI*, 2016.