Technical Note

A Bidirectional Layer7 Proxy Load Balancer with Deterministic Session Control

Daisuke Sugisawa *

Posted Date: 28 October 2025

doi: 10.20944/preprints202510.2163.v1

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

 doi:10.20944/preprints202510.2163.v1

Technical Note

A Bidirectional Layer7 Proxy Load Balancer with Deterministic Session Control

Daisuke Sugisawa

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

Abstract

This paper presents a novel Bidirectional Layer7 Proxy Load Balancer (L7LB) architecture implemented using NGINX and a minimal Acceptor module. Unlike traditional TCP upstream models, which rely on unidirectional connections from client to server, the proposed design reverses the TCP session direction to achieve deterministic request-per-second (QPS) behavior and strong resilience against traffic spikes. This approach enables predictable throughput, reduced port exhaustion, and graceful degradation under high concurrency. The Layer7 Proxy architecture was verified through a proof of concept (PoC) conducted in July 2019. Performance measurements confirm scalability with multi-core CPUs and demonstrate practical advantages over conventional L7 load balancing in dynamic, service-modular 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 where the session control direction is reversed — the server-side module actively accepts upstream socket descriptors from the load balancer via UNIX-domain FD passing. 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.

2. System Architecture

2.1. Concept Overview

As shown in Figure 1, the system consists of:

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 October 2025 doi:10.20944/preprints202510.2163.v1

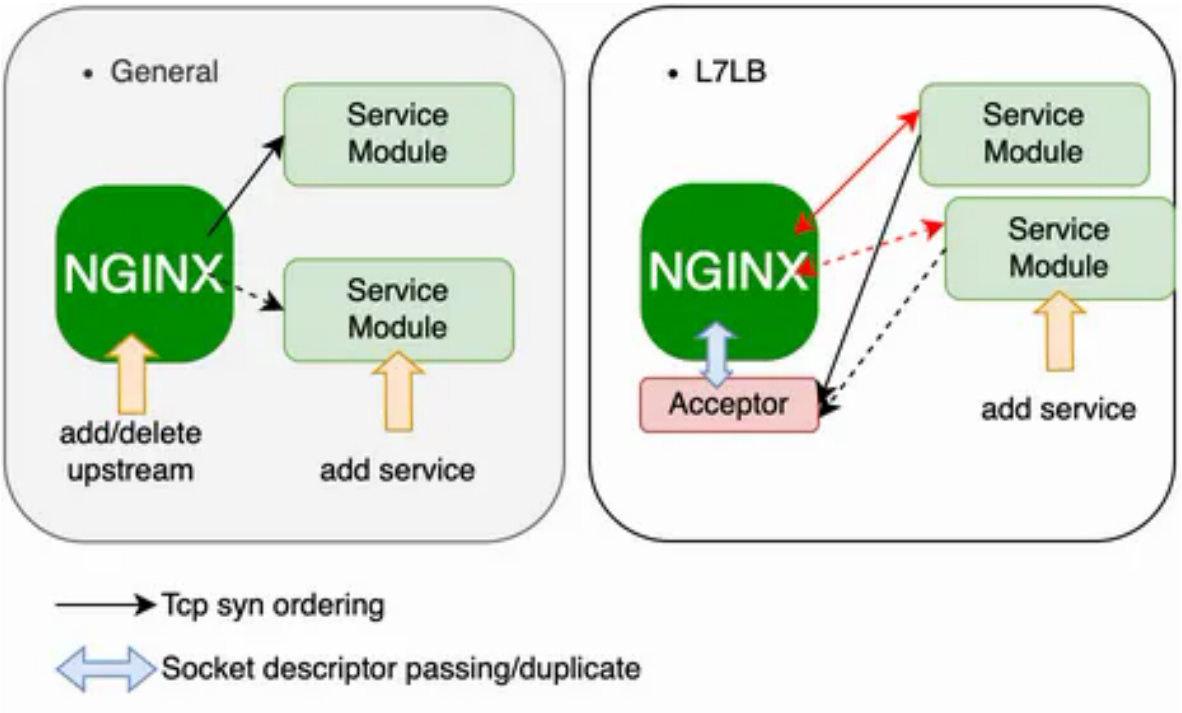

Figure 1. Layer7 Proxy Summary

- **NGINX patched module** with socket descriptor duplication.
- **Acceptor module** that transfers active connections via UNIX-domain sockets.
- **Service Modules** that perform request handling as independent processes.

TCP session ownership can thus be transferred bidirectionally between processes. (See Figure 2: Layer7 Proxy Characteristics, Figure 3: Specific Control)

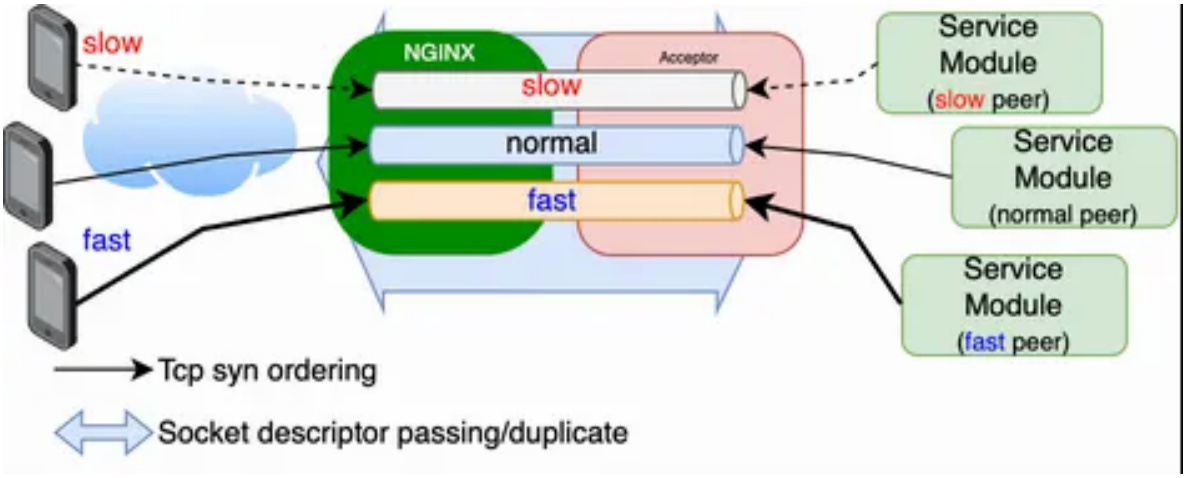

Figure 2. Layer7 Proxy Characteristics

 doi:10.20944/preprints202510.2163.v1

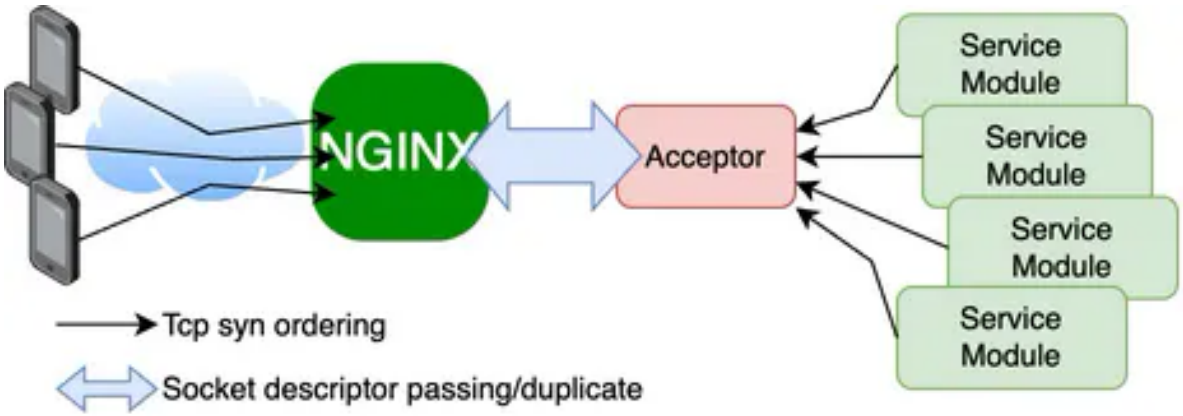

Figure 3. Layer7 Proxy Specific

2.2. Advantages over Conventional L7LB

The proposed approach addresses several long-standing issues:

- Local port exhaustion mitigation by offloading 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.

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 both-way proxying between NGINX and the Acceptor module.

(See Listing 1)

3.2. Acceptor Module

The Acceptor component duplicates accepted sockets and redistributes them to backend service modules. Implemented in C++, it utilizes a UNIX-domain server to manage socket descriptor passing (sendmsg/recvmsg) to NGINX. It maintains fairness across multiple service modules and dynamically rebuilds connection queues without terminating active sessions.

(See Listing 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 Listing 3).

 doi:10.20944/preprints202510.2163.v1

4. Performance Evaluation

4.1. Environment

Table 1. Nginx Environment

Name	Description
nginx worker process	4
Benchmark tool	ApacheBench(ab)
ab -n 10000 -c 4 -k "http://127.0.0.1:8080/"	http keep alive
ab -n 10000 -c 4 "http://127.0.0.1:8080/"	no keep alive

4.2. Results

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

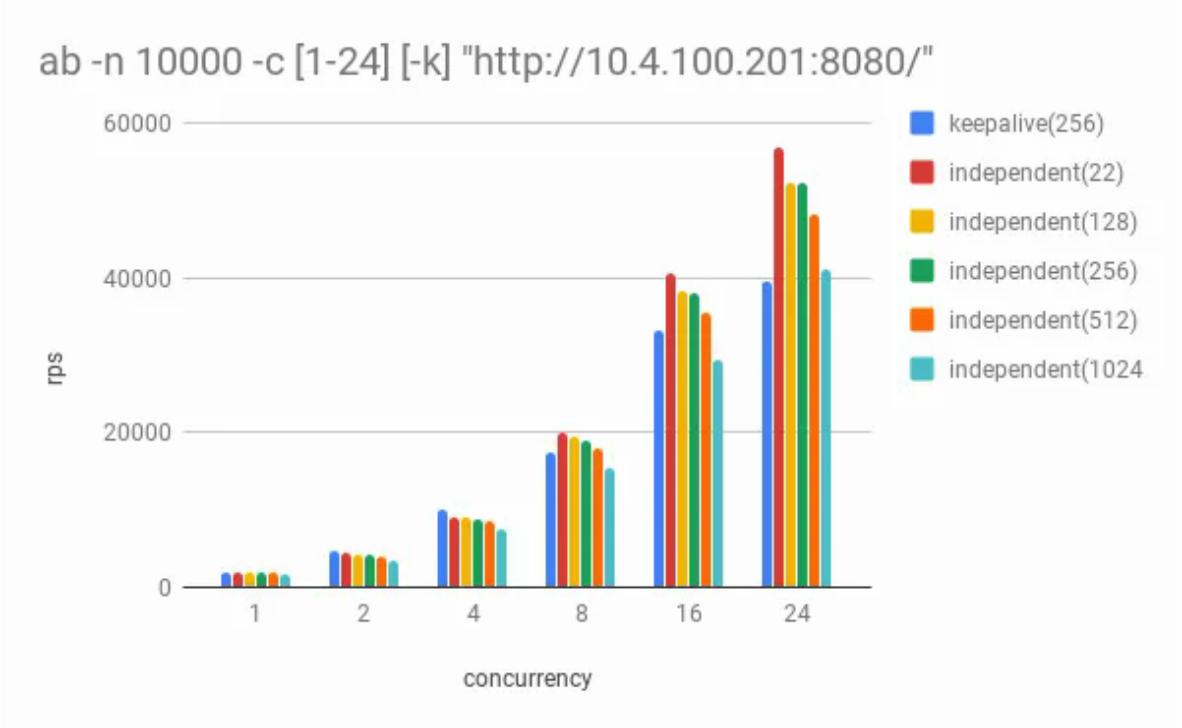

Figure 4. Layer7 Proxy Performance by Cores

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

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 October 2025 doi:10.20944/preprints202510.2163.v1

Figure 5. Layer7 Proxy Performance htop

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 reverses the direction of session ownership in the control plane: the application proactively registers its concurrency slots 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, which handle inbound TCP initiation, the reversed-direction approach ensures that connection bursts are absorbed by Acceptor queues before reaching service modules. This decouples request inflow from application readiness, allowing stable deterministic QPS behavior even under spikes.

5.3. Definition and Analysis of Deterministic QPS

5.3.1. Definition

In this paper, the term Deterministic QPS refers to a request-per-second behavior that is completely governed by the direction of TCP initiation and thus 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 precisely reflects the system's actual processing potential.

 doi:10.20944/preprints202510.2163.v1

5.3.2. Mechanism

Through this reversed session-control direction, the application side proactively establishes a number of upstream connections equivalent to its parallel processing capability. The proxy (NGINX) then dispatches requests only within that bounded pool of registered sessions. 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.
- QPS becomes deterministic and directly measurable as a function of the application's concurrency configuration.

5.3.3. Analysis

This deterministic coupling between session allocation and known service concurrency enables linear scalability with respect to CPU-core count (as demonstrated in Figure 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.

5.4. Port Exhaustion Mitigation

5.4.1. Definition

Port Exhaustion Mitigation denotes 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.

5.4.2. Problem Context

Under such traditional TCP-upstream models:

- Slow or idle clients occupy proxy ports for extended periods.
- Application-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.

5.4.3. Proposed Mechanism

In the bidirectional Layer-7 Proxy introduced in this paper, the proxy maintains independent management of upstream (client-side) and downstream (application-side) sessions. A new upstream connection toward the service application is established only when required—either upon explicit allocation from the Acceptor or when no cached session is available. When idle or reusable sessions exist, the proxy reuses them rather than spawning new TCP connections. This pull-based session creation model confines the proxy's port usage strictly within the active concurrency limits defined by the service application itself.

5.4.4. 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:

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 October 2025 doi:10.20944/preprints202510.2163.v1

- **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.

5.4.5. 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 6: Acceptor Ring), the system ensures fair distribution among service modules, preventing starvation and improving CPU cache locality.

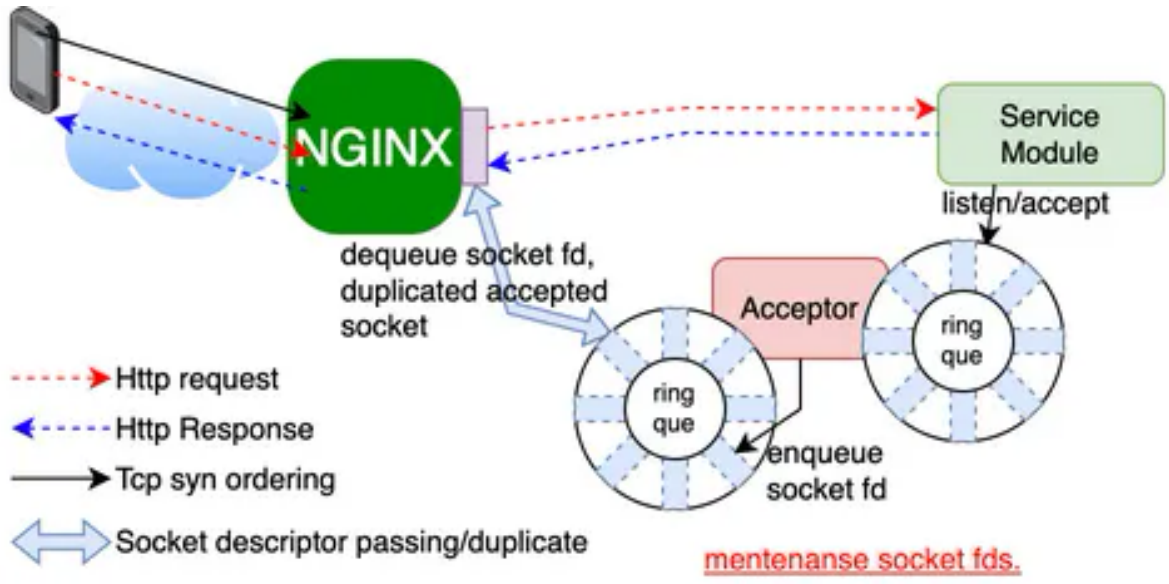

Figure 6. Layer7 Proxy Acceptor Ring

6. Comparison to Conventional Architectures

Table 2. Traditional L7LB vs Proposed Layer7 Proxy

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

 doi:10.20944/preprints202510.2163.v1

6.1. 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.

6.2. Comparison with TCP Splicing

Earlier works such as Cohen et al. (1999) proposed TCP splicing to keep both sides of a proxied connection entirely within the kernel after an initial L7 decision, eliminating context switches and buffer copies [3]. While effective under late-1990s hardware constraints, this approach has become obsolete on modern CPUs and NICs. Today, the cost of a user <-> kernel transition is negligible compared with the flexibility gained by terminating connections in user space, where TLS offload, header parsing, and adaptive routing can be performed. Consequently, systems like NGINX or Envoy deliberately perform full TCP termination instead of kernel-resident splicing, accepting minimal overhead in exchange for programmability and observability.

Our proposed bidirectional Layer-7 proxy differs fundamentally: it retains user-space control but introduces a deterministic connection-handoff mechanism between the Acceptor and service processes, ensuring fairness, burst isolation, and restart safety—benefits that TCP splicing, confined to transport-layer forwarding, never addressed. Thus, rather than restoring kernel-level splicing, our design achieves logical splice-like efficiency while preserving complete L7 visibility and control.

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

Maglev [4] 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.

 doi:10.20944/preprints202510.2163.v1

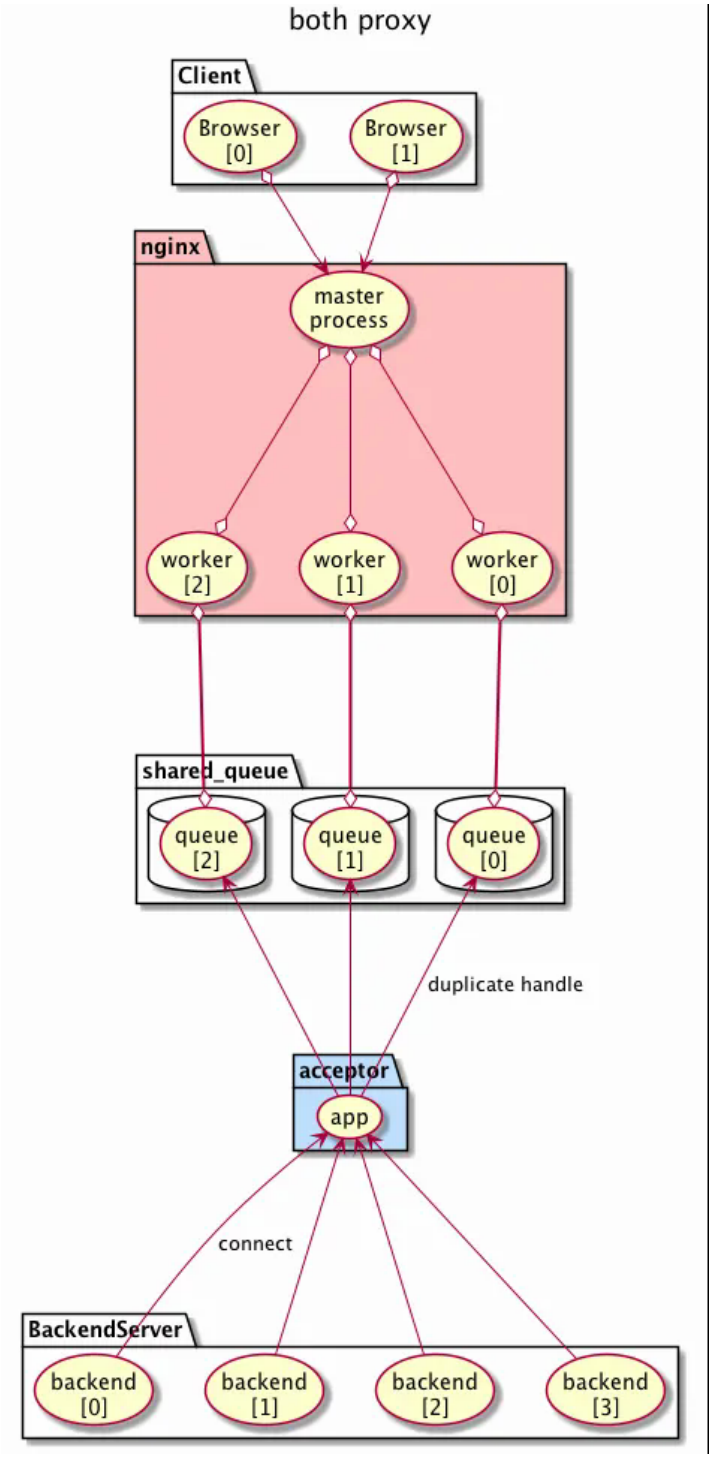

Figure 7. Layer7 Proxy Payload Sequence

7. Conclusions

This paper introduced a bidirectional Layer7 Proxy Load Balancer leveraging NGINX with minimal patching and an Acceptor coordination module. The architecture inverts the conventional TCP session direction to achieve 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.

 doi:10.20944/preprints202510.2163.v1

Listing 1: nginx.patch

```
diff --git a/src/core/nginx.h b/src/core/nginx.h
index 71377bf1..f3af40d1 100644
--- a/src/core/nginx.h
+++ b/src/core/nginx.h
@@ -22,5 +22,7 @@
    #define NGINX_VAR          "NGINX"
    #define NGX_OLDPID_EXT     ".oldbin"

+#define NGX_X_BOTH_PROXY    (1)
+

    #endif /* _NGINX_H_INCLUDED_ */
diff --git a/src/core/ngx_connection.c b/src/core/ngx_connection.c
index 33682532..a5b63524 100644
--- a/src/core/ngx_connection.c
+++ b/src/core/ngx_connection.c
@@ -1193,8 +1193,11 @@ ngx_close_connection(ngx_connection_t *c)

        if (!c->shared) {
            if (ngx_del_conn) {
+#ifdef NGX_X_BOTH_PROXY
+            ngx_del_conn(c, 0);
+#else
                ngx_del_conn(c, NGX_CLOSE_EVENT);
-
+#endif
            } else {
                if (c->read->active || c->read->disabled) {
                    ngx_del_event(c->read, NGX_READ_EVENT, NGX_CLOSE_EVENT);
diff --git a/src/event/ngx_event_connect.c b/src/event/ngx_event_connect.c
index 1ffa7984..5b16c875 100644
--- a/src/event/ngx_event_connect.c
+++ b/src/event/ngx_event_connect.c
@@ -16,6 +16,14 @@ static ngx_int_t ngx_event_connect_set_transparent(ngx_peer_connection_t *pc,
        ngx_socket_t s);
    #endif

+#ifdef NGX_X_BOTH_PROXY
+extern ngx_int_t ngx_recv_fd(ngx_peer_connection_t *,int, int*);
+extern ngx_int_t ngx_send_proxy_request(ngx_peer_connection_t *,int, char);
+
+ngx_socket_t ngx_x_acceptor_socket = -1;
+
+#endif
+

    ngx_int_t
    ngx_event_connect_peer(ngx_peer_connection_t *pc)
@@ -30,6 +38,7 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)
        ngx_socket_t       s;
        ngx_event_t       *rev, *wev;
        ngx_connection_t  *c;
+    ngx_socket_t       accepted_fd = -1;

        rc = pc->get(pc, pc->data);
        if (rc != NGX_OK) {
@@ -38,6 +47,52 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)

        type = (pc->type ? pc->type : SOCK_STREAM);

+#ifdef NGX_X_BOTH_PROXY
+    {
```

 doi:10.20944/preprints202510.2163.v1


```
+        ngx_msec_t delta = ngx_current_msec;
+        int sock;
+        struct sockaddr_un addr;
+
+        memset(&addr, 0, sizeof(struct sockaddr_un));
+        addr.sun_family = AF_UNIX;
+        snprintf(addr.sun_path,sizeof(addr.sun_path)-1, "/tmp/Xaccepted_socket"/* FIXME: from config.
  */);
+        if (ngx_x_acceptor_socket < 0){
+            if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) >= 0){
+                if (connect(sock, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) != 0) {
+                    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "connect() failed: %s:%d", strerror
  (errno), errno);
+                    close(sock);
+                }else{
+                    ngx_x_acceptor_socket = sock;
+                }
+            }
+        }
+        delta = ngx_current_msec - delta;
+        ngx_log_debug1(NGX_LOG_DEBUG_EVENT, pc->log, 0, "both proxy - connect to acceptor server : %lu\
  n", delta);
+
+        if (ngx_x_acceptor_socket > 0){
+            delta = ngx_current_msec;
+            if (ngx_send_proxy_request(pc, ngx_x_acceptor_socket, 0x80) != 0){
+                ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "ngx_send_proxy_request() failed: %d: %
  s", strerror(errno), ngx_x_acceptor_socket);
+                close(ngx_x_acceptor_socket);
+                ngx_x_acceptor_socket = -1;
+            }else{
+                if (ngx_recv_fd(pc, ngx_x_acceptor_socket, &accepted_fd) == 0){
+                    delta = ngx_current_msec - delta;
+                    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "both proxy - recieve from acceptor
   server : %lu/%d", delta, accepted_fd);
+                    // FIXME: cached peer socket fd.
+                }else{
+                    ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "ngx_recv_fd() failed: %d: %s",
  delta, ngx_x_acceptor_socket);
+                    close(ngx_x_acceptor_socket);
+                    ngx_x_acceptor_socket = -1;
+                }
+            }
+        }
+    }
+    //
+    err = 0;
+    level = 0;
+    s = accepted_fd;
+#else
        s = ngx_socket(pc->sockaddr->sa_family, type, 0);

        ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "%s socket %d",
@@ -48,6 +103,7 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)
                          ngx_socket_n " failed");
            return NGX_ERROR;
        }
+#endif  /* NGX_X_BOTH_PROXY */


        c = ngx_get_connection(s, pc->log);
@@ -93,7 +149,8 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)
        }
```

 doi:10.20944/preprints202510.2163.v1


```
        if (pc->local) {
-
+#ifdef NGX_X_BOTH_PROXY
+#else
    #if (NGX_HAVE_TRANSPARENT_PROXY)
            if (pc->transparent) {
                if (ngx_event_connect_set_transparent(pc, s) != NGX_OK) {
@@ -155,6 +212,7 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)

                goto failed;
            }
+#endif  /* NGX_X_BOTH_PROXY */
        }

        if (type == SOCK_STREAM) {
@@ -202,6 +260,10 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)
        ngx_log_debug3(NGX_LOG_DEBUG_EVENT, pc->log, 0,
                       "connect to %V, fd:%d #%uA", pc->name, s, c->number);

+#ifdef NGX_X_BOTH_PROXY
+//    c->shared = 1;
+
+#else
        rc = connect(s, pc->sockaddr, pc->socklen);

        if (rc == -1) {
@@ -244,6 +306,7 @@ ngx_event_connect_peer(ngx_peer_connection_t *pc)
                return NGX_DECLINED;
            }
        }
+#endif  /* NGX_X_BOTH_PROXY */

        if (ngx_add_conn) {
            if (rc == -1) {
@@ -429,3 +492,66 @@ ngx_event_get_peer(ngx_peer_connection_t *pc, void *data)
    {
        return NGX_OK;
    }
+
+ngx_int_t
+ngx_recv_fd(ngx_peer_connection_t* pc, int sock, int *fd)
+{
+    union {
+        struct cmsghdr h;
+        char   control[CMSG_SPACE(sizeof(int))];
+    } buffer;
+
+    struct msghdr msghdr;
+    char nothing;
+    struct iovec nothing_ptr[1];
+    struct cmsghdr *cmsg;
+
+    nothing_ptr[0].iov_base = &nothing;
+    nothing_ptr[0].iov_len = 1;
+    msghdr.msg_name = NULL;
+    msghdr.msg_namelen = 0;
+    msghdr.msg_iov = nothing_ptr;
+    msghdr.msg_iovlen = 1;
+    msghdr.msg_flags = 0;
+    msghdr.msg_control = buffer.control;
+    msghdr.msg_controllen = sizeof(struct cmsghdr) + sizeof(int);
+    cmsg = CMSG_FIRSTHDR(&msghdr);
+    cmsg->cmsg_len = msghdr.msg_controllen;
+    cmsg->cmsg_level = SOL_SOCKET;
```

 doi:10.20944/preprints202510.2163.v1


```
+    cmsg->cmsg_type = SCM_RIGHTS;
+    ((int *)CMSG_DATA(cmsg))[0] = -1;
+
+    if(recvmsg(sock, &msghdr, 0) < 0){
+        ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "recvmsg() failed: %d: %s", sock, strerror(
  errno));
+        return(-1);
+    }
+    (*fd) = ((int *)CMSG_DATA(cmsg))[0];
+    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, pc->log, 0, "ngx_recv_fd() : %d", (*fd));
+
+    return((*fd)<0?-1:0);
+}
+ngx_int_t
+ngx_send_proxy_request(ngx_peer_connection_t* pc, int sock, char proxy_req)
+{
+    struct msghdr msghdr;
+    struct iovec nothing_ptr[1];
+    char   arg = proxy_req;
+
+    nothing_ptr[0].iov_base = &arg;
+    nothing_ptr[0].iov_len = 1;
+    msghdr.msg_name = NULL;
+    msghdr.msg_namelen = 0;
+    msghdr.msg_iov = nothing_ptr;
+    msghdr.msg_iovlen = 1;
+    msghdr.msg_flags = 0;
+    msghdr.msg_control = NULL;
+    msghdr.msg_controllen = 0;
+    //
+    if(sendmsg(sock, &msghdr, 0) < 0){
+        ngx_log_debug2(NGX_LOG_DEBUG_EVENT, pc->log, 0, "sendmsg() failed: %d: %s", sock, strerror(
  errno));
+        return(-1);
+    }
+    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, pc->log, 0, "ngx_send_proxy_request() : %02x", proxy_req);
+
+    return(0);
+}
```

Listing 2: accepter.cc

```
static void* _udssrv_session(void* arg){
    auto ses = (session_context_ptr)arg;
    std::map<uint32_t, uint32_t> &available_address = *((std::map<uint32_t, uint32_t>*)ses->ctx->
  available_address);
    client_ptr  cur, curtmp;
    int peer_fd = 0;
    char request_code = 0;
    int sock_err = 0;
    socklen_t sockopt_len = 0;
    int is_exists = 0;
    unsigned address = 0;

    //
    while(!__halt){
        bzero(&peer_fd, sizeof(peer_fd));
        // wait for request.
        if (acceptor_recieve_req(ses->sock , &request_code) != 0){
            LOG(LOG_ERR, "acceptor_recieve_req . failed.(%d)", ses->sock);
            goto fin;
        }
        // delegate tcp sock to nginx.
        // if empty tcp client -> wait.
```

 doi:10.20944/preprints202510.2163.v1


```
        // round robin dispatch.
        pthread_mutex_lock(&ses->ctx->tcp_clients_mutex);
        TAILQ_FOREACH_SAFE(cur, &ses->ctx->tcp_clients, link, curtmp) {
            sock_err = 0;
            sockopt_len = sizeof(sock_err);
            if (getsockopt(cur->sock, SOL_SOCKET, SO_ERROR, &sock_err, &sockopt_len) != 0){
                TAILQ_REMOVE(&ses->ctx->tcp_clients, cur, link);
                free(cur);
            }else if (sock_err != 0 && sock_err != EISCONN){
                TAILQ_REMOVE(&ses->ctx->tcp_clients, cur, link);
                free(cur);
            }else{
                is_exists = 1;
                address = (*((uint32_t*)(&cur->proxy_packet[sizeof(proxy_header_t)])));
                pthread_mutex_lock(&ses->ctx->available_address_mutex);
                if (available_address.find(address) == available_address.end()){
                    is_exists = 0;
                }
                pthread_mutex_unlock(&ses->ctx->available_address_mutex);
                //
                if (is_exists){
                    peer_fd = dup(cur->sock);
                    //LOG(LOG_INFO, "%p : %d: %s:%u --> %d", pthread_self(), cur->sock, cur->remote, cur
  ->port, peer_fd);
                    auto p = (client_ptr)malloc(sizeof(client_t));
                    bzero(p, sizeof(client_t));
                    p->sock = cur->sock;
                    memcpy(p->remote, cur->remote, sizeof(p->remote));
                    p->port = cur->port;
                    p->ctx = cur->ctx;
                    p->client_event = cur->client_event;
                    memcpy(p->proxy_packet, cur->proxy_packet, sizeof(p->proxy_packet));
                    //
                    TAILQ_INSERT_TAIL(&ses->ctx->tcp_clients, p, link);
                    TAILQ_REMOVE(&ses->ctx->tcp_clients, cur, link);
                    free(cur);
                    if (peer_fd != 0){
                        break;
                    }
                }else{
                    shutdown(cur->sock, SHUT_RDWR);
                    close(cur->sock);
                    TAILQ_REMOVE(&ses->ctx->tcp_clients, cur, link);
                    free(cur);
                }
            }
        }
        pthread_mutex_unlock(&ses->ctx->tcp_clients_mutex);
        //
        if (peer_fd == 0){
            LOG(LOG_ERR, "empty fd. . failed.(%d)", peer_fd);
            goto fin;
        }
        if (acceptor_sendfd(ses->sock, &peer_fd) != 0){
            LOG(LOG_ERR, "acceptor_sendfd . failed.(%d : %d)", ses->sock, peer_fd);
            goto fin;
        }
        close(peer_fd);
    }
fin:
    shutdown(ses->sock, SHUT_RDWR);
    close(ses->sock);
    free(ses);
    pthread_detach(pthread_self());
```

 doi:10.20944/preprints202510.2163.v1


```
    return(NULL);
}
```

Listing 3: example.go

```
package main

import (
    "fmt"
    "log"
    "net"
    "runtime"
    "strconv"
    "strings"
    "time"
)

func main() {
    for n := 0; n < 32; n++ {
        go startconn(n)
    }
    counter := 0
    for {
        time.Sleep(1 * time.Second)
        if counter%30 == 0 {
            fmt.Printf(".... : %+v\n", counter)
        }
        counter = counter + 1
    }
}

func startconn(id int) {
    for {
        conn, err := net.Dial("tcp", "127.0.0.1:9999")
        if err != nil {
            log.Printf("%+v", err)
            break
        }
        defer conn.Close()
        fmt.Printf("context : %+v :  %+v\n", id, goid())
        for {
            res := make([]byte, 2048)
            _, err := conn.Read(res)
            if err != nil {
                log.Printf("read ... %+v", err)
                break
            }
            reshtml := "HTTP/1.1 200 OK\r\n" + "Content-Type: text/html\r\n" + "Content-Length: 22\r\n\r
  \n" + "<h1>Hello World" + fmt.Sprintf("%02d", id) + "</h1>"
            _, err2 := conn.Write([]byte(reshtml))
            if err2 != nil {
                log.Printf("failed.write ... %+v", err2)
                break
            }
        }
    }
}
func goid() int {
    var buf [64]byte
    n := runtime.Stack(buf[:], false)
    idField := strings.Fields(strings.TrimPrefix(string(buf[:n]), "goroutine "))[0]
    id, err := strconv.Atoi(idField)
    if err != nil {
        panic(fmt.Sprintf("cannot get goroutine id: %v", err))
```

 doi:10.20944/preprints202510.2163.v1


```
    }
    return id
}
```

Conflicts of Interest: The author declares no conflicts of interest.

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

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

Funding: This research received no external funding.

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. Ely, S. M. Bellovin, and M. Richardson, "TCP Splice: Asymmetric TCP Connection Splicing for Application Layer Gateways," *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'99)*, USENIX Association, 1999.
4. 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.