TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 19 November 2026

W. Wang
A. Wang
China Telecom
Z. Wang
Beijing Infosec Technologies Co., LTD.
M. Sahni
K. Sheth
Palo Alto Networks

18 May 2026

Service Affinity Solution based on Transport Layer Security (TLS)
draft-wang-tls-service-affinity-02

Abstract

This draft proposes a service affinity solution between client and server based on Transport Layer Security (TLS). An extension to Transport Layer Security (TLS) 1.3 to enable session migration. This mechanism is designed for network architectures, particularly for multi-homed servers that possess multiple network interfaces and IP addresses.

This document also introduces a Reliable Framing Layer that operates above the TLS record layer to provide message framing, sequence numbering, acknowledgment tracking, and automatic retransmission. The Framing Layer ensures zero application data loss during TLS session migration by buffering unacknowledged data frames and retransmitting them to the new server endpoint after migration completes.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 November 2026.

Table of Contents

1. Introduction

The rapid growth of internet services and the increasing complexity of network architectures have created a demand for more flexible and resilient connections between clients and servers. Modern service deployments often utilize multi-homed servers. These are systems that are equipped with multiple network interfaces and IP addresses. This architecture enhances availability, balances loads, and optimizes routing based on dynamic network conditions.

In such environments, clients often need to migrate their connections from one server IP address to another. Service continuity must be maintained during traffic migration. This necessity can arise from changes in network topology, server maintenance requirements, or the need to balance computational resources across different service nodes.

In traditional solutions for maintaining service affinity or facilitating migration, each device needs to maintain a customer-based connection status table. This table will not change dynamically with the change of network status and computing resources. Moreover, the network devices should keep large amounts of status table to keep the service affinity for every customer flow. As the number of sessions increases, this table will grow in size, and an excessive number of sessions will impose pressure on the device.

Besides, in the load balance scenario, a load balancer is usually put in front of all the physical servers so that all the packets sent and received by the physical servers should pass through the load balancer. This deployment may lead to the load balancer become the bottleneck when the traffic increases.

HTTP redirection enables automatic page jumps by having the browser automatically send a new request based on the specific response status code and the value of the Location field returned by the server. It mainly involve the communication between client and server. Both client and server do not perceive changes in network status and cannot achieve comprehensive optimization based on network status and computing resource status.

DNS redirection can redirect customer requests from one domain name to another by modifying DNS resolution records, or change the resolution result of a domain name to point to a different server IP address. However, due to the caching time of DNS records, it takes some time for the modification to take effect, which may result in customers still accessing servers that have been taken offline, thereby affecting customer experience.

We propose a solution for the service affinity between client and server by extending TLS 1.3. This proposal is designed for environments where operational simplicity and migration speed are paramount. It intentionally omits the path validation steps to minimize the latency of the migration process. Furthermore, it simplifies the trigger mechanism by using a new TLS alert, which is a direct and unambiguous signal.

While the TLS session migration mechanism described in Sections 4-5 ensures cryptographic session continuity, it does not by itself guarantee that in-flight application data is preserved during the reconnection. To address this, Section 6 introduces a Reliable Framing Layer that provides message framing, sequence tracking, and automatic retransmission to ensure zero application data loss across migrations.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] .

3. Motivation and design rationale

In distributed cloud and edge computing architectures, traditional session identification based on static IP addresses can no longer meet the demands of dynamic networks. This proposal chooses to implement service affinity at the TLS layer, rather than through redirection at the application layer (e.g., HTTP) or the network layer, based on the following three core dimensions:

First, TLS 1.3 [RFC8446] provides a secure channel for negotiating connection parameters without exposing sensitive data to network intermediaries. By conveying migration instructions and cryptographic material within the handshake, the solution avoids the visibility and interference issues associated with in-band application-layer signaling (e.g., HTTP redirects) or out-of-band network-layer mechanisms.

Second, the TLS session resumption framework offers a natural abstraction for session continuity. The proposed extension leverages the existing NewSessionTicket message to bind migration authorization to the session state. This approach ensures that migration tokens are cryptographically bound to the original session keys, preventing unauthorized redirection or session hijacking.

Third, integrating migration into the handshake enables 0-RTT resumption at the new endpoint. When a client migrates, it presents the ticket containing the migration extension, allowing the new server instance to validate the token and resume the session without performing a full cryptographic handshake. This minimizes the latency impact of migration, which is critical for real-time applications.

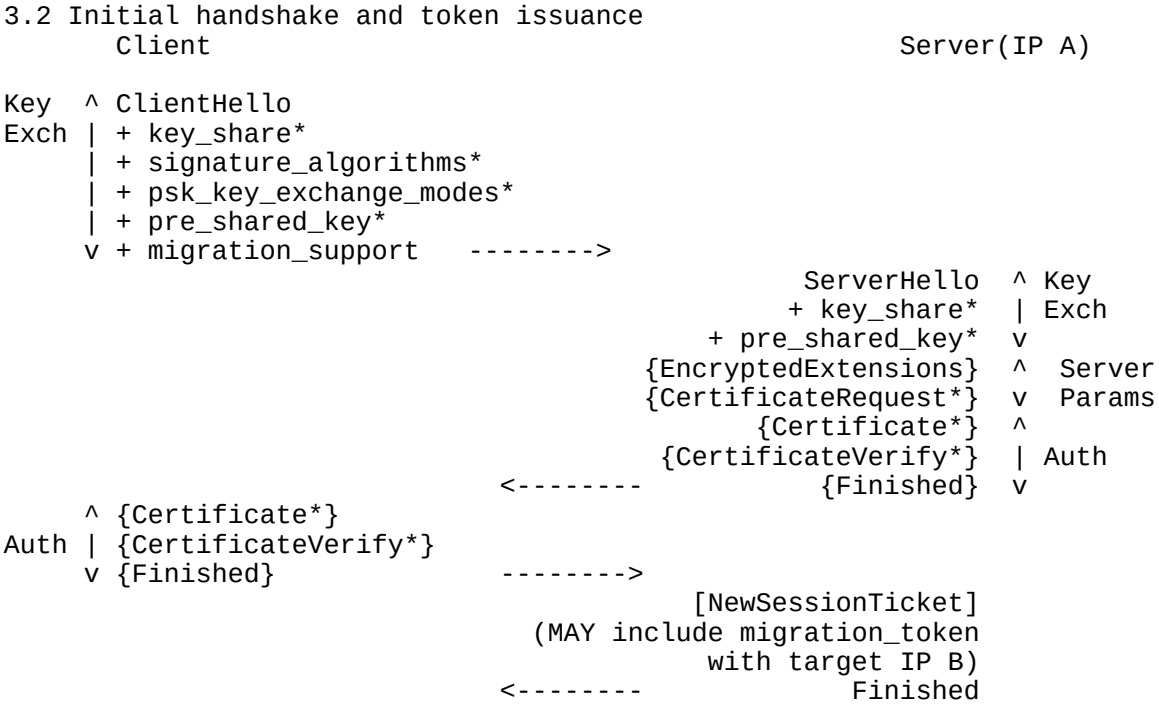
Critically, this design does not require changes to the application data flow. It is transparent to both the application and the network path, making it compatible with any protocol running over TLS.

4. Procedures of the proposed solution

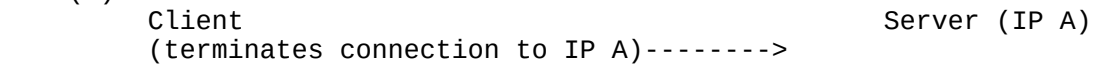
4.1. Message flow of the overall procedure

The message flow of the procedures of service affinity mechanism based on TLS are shown in Figure 1.

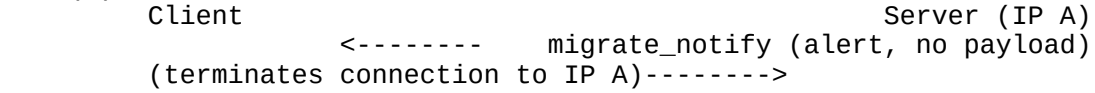
3.2 Initial handshake and token issuance



3.3 (a) Client initiated:



3.3 (b) Server initiated:



3.4 Reconnection and resumption

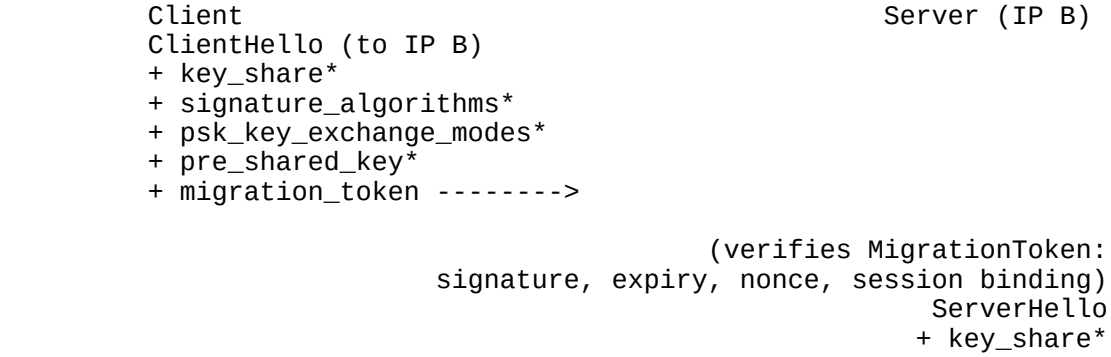


Figure 1: service affinity mechanism based on TLS

4.2. Phase 1: initial handshake and token issuance

1. A client supporting this mechanism includes the ‘migration_support‘ extension in its initial ‘ClientHello‘ message to the server at IP A. This extension is empty and serves only to signal capability.

2. The server at IP A completes a standard TLS 1.3 handshake.

3. After the handshake is complete, the server sends a ‘NewSessionTicket‘ message to enable standard Pre-Shared Key-based (PSK-based) session resumption. Within this message, the server MAY include the new ‘migration_token‘ extension. This extension contains the ‘MigrationToken‘, an authorization credential that includes the pre-determined destination (IP B) for a future migration.

4.3. Phase 2: migration trigger

a) If the session migration is triggered by the client, the client can directly switch the session to the new server according to business requirements.

b) If the session migration is triggered by the server, it performs as follow:

1. At a later point, the server at IP A initiates the migration.

2. The server sends a new TLS alert, ‘migrate_notify‘, over the encrypted and authenticated connection. This alert has no payload and serves as a simple, direct instruction for the client to initiate the migration process.

4.4. Phase 3: reconnection and resumption

1. The client inspect its stored ‘MigrationToken‘. If a valid token exists, it extracts the target IP address and port, terminates its connection to IP A, and initiates a new TLS connection to IP B.

2. The client sends a ‘ClientHello‘ message to IP B. This message MUST include:

* The standard ‘pre_shared_key‘ extension, containing the session ticket identity received from IP A.

* The ‘migration_token‘ extension, containing the ‘MigrationToken‘ it received from IP A.

3. The server at IP B uses the PSK identity to retrieve the session state. It then MUST validate the ‘MigrationToken‘, confirming its signature, expiration, and nonce, and verifying that the token is cryptographically bound to the session.

4. If all checks pass, the server accepts the PSK and completes the abbreviated handshake.

5. Detailed formats

This section defines the structure of the new protocol elements, following the presentation language of [RFC8446].

5.1. migration_support extension

This extension is sent in the ‘ClientHello‘ to indicate support for this protocol. The ‘extension_data‘ field of this extension is zero-length.

```
struct { } MigrationSupport;
```

5.2. migration_token extension

This extension is sent in the ‘NewSessionTicket‘ message and contains the ‘MigrationToken‘ structure. It is also sent by the client in the ‘ClientHello‘ during a migration attempt.

```
MigrationToken migration_token;
```

The ‘MigrationToken‘ is a credential that authorizes the migration of a specific session to a pre-determined destination.

```
enum { ipv4(0), ipv6(1) } IPAddressType;

struct {
    IPAddressType type;
    select (IPAddress.type) {
        case ipv4: uint8 ipv4_address[4];
        case ipv6: uint8 ipv6_address[16];
    };
    uint16 port;
} IPAddress;

struct {
    IPAddress target_address;
    opaque session_id<32..255>;
    uint64 expiry_timestamp;
    opaque nonce<16..255>;
    opaque signature<32..255>;
} MigrationToken;
```

Where:

* target_address: An ‘IPAddress‘ structure specifying the destination IP address (v4 or v6) and port for the client to reconnect to.

* session_id: A unique identifier for the TLS session, derived from the session’s ‘resumption_master_secret‘ using an HKDF-Expand function.

* expiry_timestamp: A 64-bit unsigned integer representing the Unix timestamp after which this token becomes invalid.

* nonce: A unique, single-use value generated by the server to prevent replay attacks.

* signature: An HMAC tag providing integrity and authenticity. The signature is computed over a concatenation of the ‘target_address‘, ‘session_id‘, ‘expiry_timestamp‘, and ‘nonce‘ fields. The key for the HMAC MUST be derived from the ‘resumption_master_secret‘.

5.3. migrate_notify alert

This proposal introduces a new alert type to trigger the migration.

```
enum {
    ...,
    migrate_notify(TBD3),
    ...
} AlertDescription;
```

The ‘migrate_notify‘ alert is a notification-level alert. Upon receiving this alert, the client SHOULD initiate the migration process as described in Section 3.3. It does not indicate a protocol error.

6. Reliable Framing Layer

6.1. Overview and Motivation

The TLS session migration mechanism described in Sections 4-5 provides cryptographic session continuity, enabling a client to seamlessly resume a TLS session on a new server endpoint. However, during the migration process, there is a period where the client has sent application data to the original server (Server A) that has not yet been acknowledged by the application layer. If the client disconnects from Server A before receiving acknowledgments for all sent data, that data may be lost.

This section defines a Reliable Framing Layer that operates above the TLS record layer (as shown in Figure 2) to provide:

* Message Framing: Delineation of application-level messages within the continuous TLS byte stream, independent of TLS record boundaries.

* Sequence Numbering: Unique, monotonically increasing sequence numbers for each sent frame, enabling ordered delivery and gap detection.

* Acknowledgment Tracking: Explicit ACK frames that allow the receiver to confirm receipt of specific data frames, giving the sender visibility into which data has been processed.

* Automatic Retransmission: Upon migration to a new server endpoint, the client automatically retransmits all data frames that were not acknowledged by the previous server, ensuring zero application data loss.

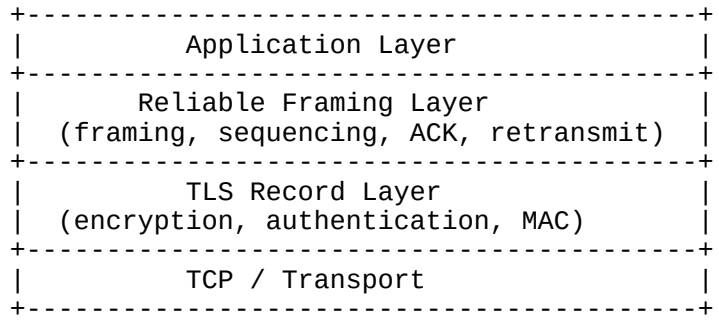


Figure 2: Protocol Stack with Reliable Framing Layer

The Framing Layer is designed to be:

* Transparent to TLS: It operates on top of the established TLS connection and does not modify TLS handshake behavior.

* Transport-agnostic: While specified here in the context of TLS session migration, the framing protocol can operate over any reliable byte-stream transport.

* Minimal overhead: The frame header is compact (11 bytes), and the protocol does not require additional round trips beyond those needed for TLS session migration.

* Application-controlled ACK policy: The receiver decides when to send ACKs, enabling strategies such as immediate ACK, delayed ACK, or selective ACK based on application requirements.

6.2. new section

Every frame transmitted over the Framing Layer has the following structure:

```
enum {
    FRAME_MAGIC(0x46524D),  // "FRM" in ASCII
} FrameMagic;

enum {
    FRAME_FLAG_DATA(0x00),
    FRAME_FLAG_ACK(0x01),
    FRAME_FLAG_FIN(0x02),
    FRAME_FLAG_RETRANSMIT(0x04),
} FrameFlag;

struct {
    uint16 magic;            // Frame magic number (0x4652)
    uint8  flags;            // Bitwise OR of FrameFlag values
    uint32 sequence_number;  // Sequence number of this frame
    uint32 length;           // Length of payload in bytes
    opaque payload[length];  // Frame payload (0 for ACK/FIN)
} Frame;
```

The total header size is 11 bytes:

Offset	Size	Field	Description
0	2	magic	Frame magic number 0x4652 ("FR")
2	1	flags	Bitwise OR of frame type flags
3	4	sequence_number	Sequence number of this frame
7	4	length	Payload length in bytes (0 for ACK/FIN)
11	variable	payload	Frame payload data

Constants:

```
const uint16 FRAME_MAGIC       = 0x4652;
const uint8  FRAME_FLAG_DATA   = 0x00;
const uint8  FRAME_FLAG_ACK    = 0x01;
const uint8  FRAME_FLAG_FIN    = 0x02;
const uint8  FRAME_FLAG_RETRANSMIT = 0x04;
const uint16 FRAME_HEADER_SIZE = 11;
const uint32 FRAME_MAX_PAYLOAD = 4096;
const uint32 FRAME_MAX_QUEUE   = 1024;
```

6.3. Frame Types and Flags

6.3.1. DATA Frame (0x00)

DATA frame carries application data from the sender to the receiver. Its format is shown in Figure 3.

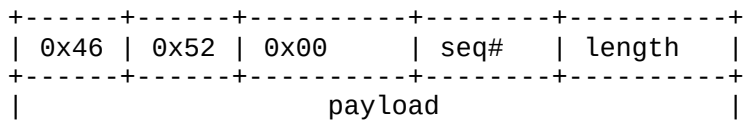


Figure 3: The format of DATA frame

Where:

* sequence_number: The sender’s monotonically increasing sequence number for this data frame.

* length: The length of the application data payload.

* payload: The application data bytes, up to 4096 bytes.

The receiver SHOULD send an ACK frame for each received DATA frame, according to its ACK policy.

6.3.2. ACK Frame (0x01)

ACK frame is the acknowledges receipt of a specific DATA frame. Its format is shown in Figure 4.

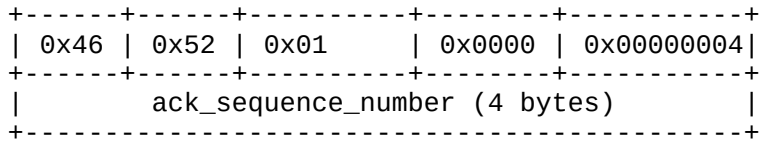


Figure 4: The format of ACK frame

Where:

* sequence_number field in header: MUST be set to 0 (unused for ACK).

* length: MUST be 4.

* payload: A 4-byte big-endian unsigned integer containing the sequence number of the DATA frame being acknowledged.

Upon receiving an ACK frame, the sender marks the corresponding entry in its send queue as acknowledged.

6.3.3. FIN Frame (0x02)

FIN frame signals the graceful end of the framing session. No payload. Its format is shown in Figure 5.

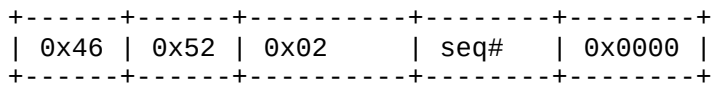


Figure 5: The format of FIN frame

Where:

* sequence_number: The sender’s next expected send sequence number.

* length: MUST be 0.

The receiver of a FIN frame SHOULD respond with its own FIN frame and then close the connection.

6.3.4. RETRANSMIT Flag (0x04)

RETRANSMIT flag is combined with FRAME_FLAG_DATA using bitwise OR to indicate that a DATA frame is a retransmission of a previously sent frame.

```
flags = FRAME_FLAG_DATA | FRAME_FLAG_RETRANSMIT  // 0x04
```

The receiver SHOULD process retransmitted frames in the same manner as new frames. If the frame has already been processed (i.e., its sequence number is less than the receiver’s next_recv_seq), the receiver MAY silently discard it but MUST still send an ACK.

6.4. Sequence Numbering and Acknowledgment

6.4.1. Sequence Number Space

Each endpoint maintains two sequence counters:

* next_send_seq: The sequence number to assign to the next DATA frame sent. Initialized to 1 and incremented after each DATA frame is queued for sending.

* next_recv_seq: The sequence number expected for the next DATA frame received. Initialized to 1 and incremented when a new (non-duplicate) DATA frame is successfully received.

Sequence numbers are 32-bit unsigned integers. They MUST be initialized to 1 at the start of each framing session (i.e., when a new TLS connection is established).

6.4.2. Send Queue

The sender maintains a send queue that stores metadata for each sent DATA frame until it is acknowledged:

```
struct {
    uint32 sequence_number;   // Sequence number of the frame
    uint32 length;            // Payload length
    opaque data[length];      // Copy of the payload data
    boolean acked;            // Whether this frame has been ACKed
    uint8  retransmit_count;  // Number of times retransmitted
} SendQueueEntry;
```

The send queue has a maximum capacity of FRAME_MAX_QUEUE (1024) entries. If the queue is full, the sender MUST stop accepting new application data until space becomes available (i.e., until some frames are acknowledged).

6.4.3. Acknowledgment Processing

When the sender receives an ACK frame with ack_sequence_number equal to N, it searches its send queue for the entry with sequence_number == N and marks it as acked = true.

The sender MAY prune acknowledged entries from the send queue to reclaim memory, but MUST retain all unacknowledged entries for potential retransmission.

6.4.4. Duplicate Detection

When the receiver receives a DATA frame with sequence_number < next_recv_seq, it identifies the frame as a duplicate (retransmission). The receiver:

* MUST still send an ACK for the frame.

* SHOULD discard the payload without delivering it to the application.

6.5. Send Queue and Retransmission

6.5.1. Retransmission Trigger

Retransmission is performed in two scenarios:

* Migration-triggered retransmission: After completing TLS session migration to a new server (Section 4.4), the client MUST retransmit all unacknowledged DATA frames from its send queue to the new server. This is the primary use case addressed by this specification.

* Timer-based retransmission (optional): Implementations MAY support a retransmission timer that triggers retransmission of unacknowledged frames after a configurable timeout. This is useful for handling packet loss within a single connection but is OPTIONAL for the migration use case.

6.5.2. Retransmission Procedure

To retransmit unacknowledged frames, the sender:

1. Iterates through the send queue from the oldest to the newest entry.

2. For each entry where acked == false:

* Constructs a new DATA frame with flags = FRAME_FLAG_DATA | FRAME_FLAG_RETRANSMIT.

* Sets sequence_number to the original frame's sequence number.

* Copies the original payload data.

* Sends the frame over the current TLS connection.

* Increments the entry's retransmit_count.

3. The sender MUST preserve the original sequence numbers during retransmission. The receiver uses sequence numbers to detect and deduplicate retransmitted frames.

```
RetransmitAllUnacked():
    for each entry in send_queue:
        if entry.acked == false:
            frame = Frame {
                magic: 0x4652,
                flags: DATA | RETRANSMIT,
                sequence_number: entry.sequence_number,
                length: entry.length,
                payload: entry.data
            }
            send(frame)
            entry.retransmit_count++
```

6.6. Framing Layer State Machine

6.6.1. Sender State

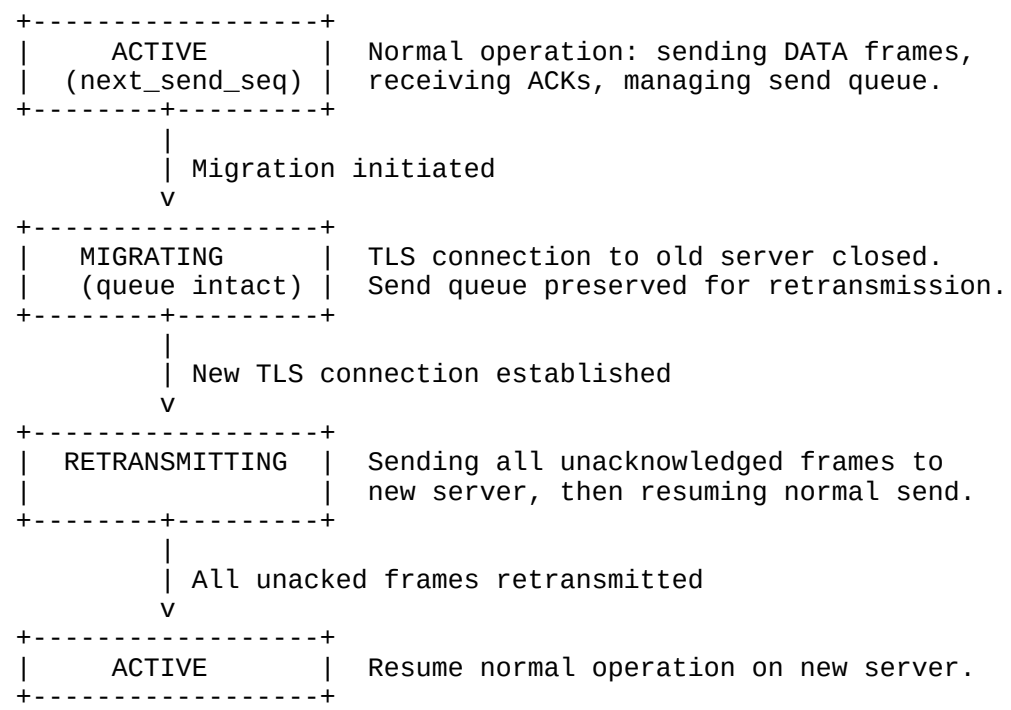


Figure 6: Sender state machine

6.6.2. Receiver State

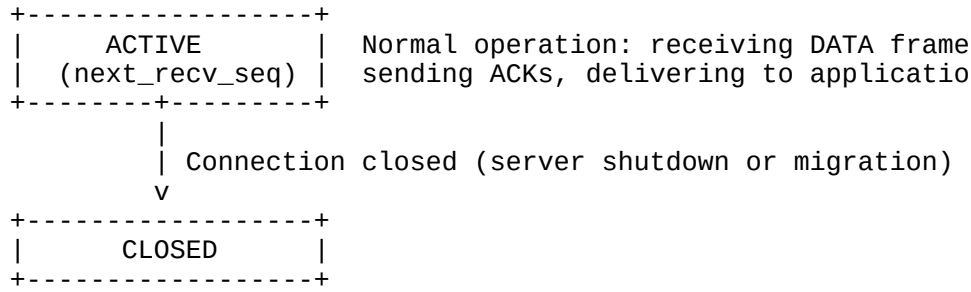


Figure 7: Receiver state machine

On the new server, the receiver initializes next_recv_seq = 1 and processes both retransmitted and new frames normally. The receiver does not need to be aware that some frames are retransmissions from a previous server.

6.6.3. State Reset on New Connection

When a new TLS connection is established (after migration), the sender:

* MUST reset next_recv_seq to 1 (for receiving ACKs and data from the new server).

* MUST preserve the send queue intact (for retransmission).

* MUST preserve next_send_seq (to avoid sequence number collisions with retransmitted frames).

The receiver on the new server:

* MUST initialize next_recv_seq to 1.

* MUST initialize an empty send queue.

6.7. Framing Layer Procedures During Migration

6.7.1. Pre-Migration (Phase 1)

During normal data transfer with Server A:

* The client sends DATA frames via the Framing Layer. Each frame is assigned a monotonically increasing sequence number and added to the send queue.

* Server A receives DATA frames and sends ACK frames according to its ACK policy. The application on Server A processes the data and decides whether to acknowledge each frame.

* The client processes incoming ACK frames and marks corresponding send queue entries as acknowledged.

6.7.2. Migration (Phase 2)

When migration is triggered:

* The client records the current state of its send queue, noting which frames remain unacknowledged.

* The client disconnects from Server A. The send queue is preserved in memory.

* The client initiates a new TLS connection to Server B, performing PSK-based session resumption as described in Section 4.4.

6.7.3. Post-Migration (Phase 3)

After the new TLS connection is established with Server B:

* The client resets its receive state (next_recv_seq = 1) for the new connection.

* The client calls RetransmitAllUnacked() to resend all unacknowledged DATA frames to Server B. These frames carry the RETRANSMIT flag.

* Server B receives the retransmitted frames. Since Server B initializes next_recv_seq = 1, it processes them as new frames, sends ACKs, and delivers the data to the application.

* The client resumes normal data transfer, sending new DATA frames with sequence numbers continuing from where it left off.

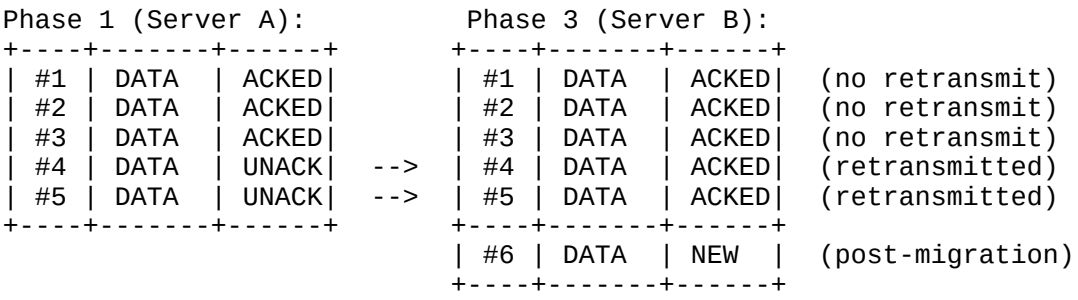
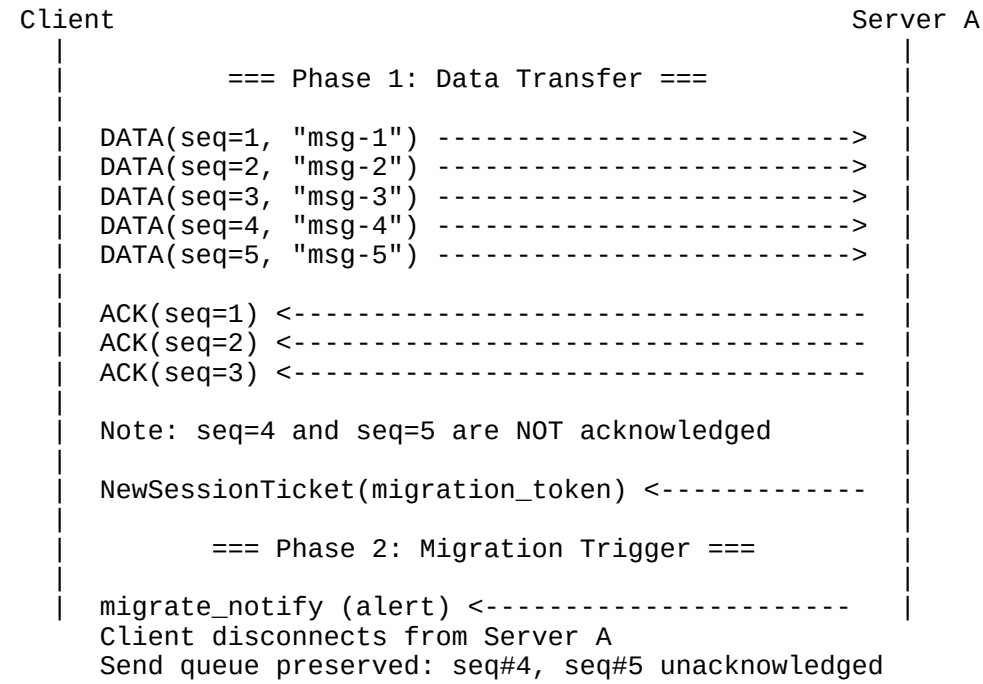


Figure 8: Send Queue State Transition During Migration

6.8. Message Flow with Framing Layer

The following diagram illustrates the complete message flow including the Framing Layer during a migration scenario where Server A acknowledges only the first 3 of 5 data frames:



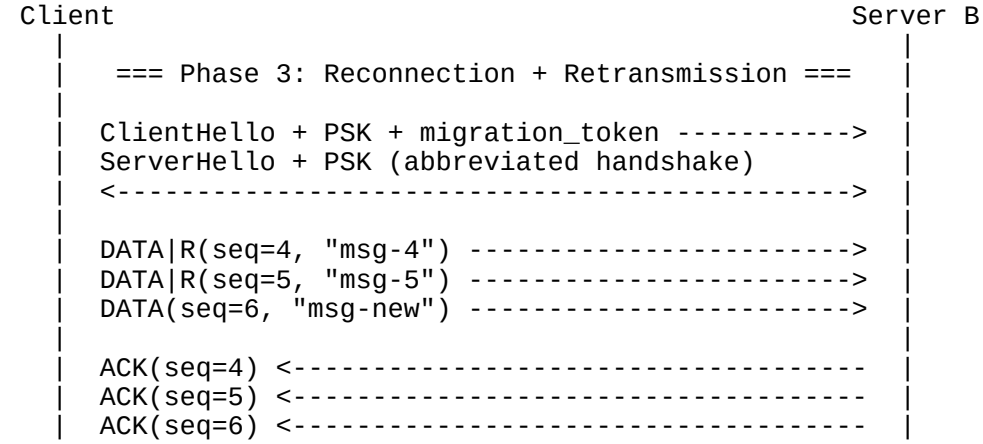


Figure 9: Framing Layer Message Flow During Migration

7. Use case

Computing-Aware Traffic Steering (CATS) provides a compelling use case for TLS-layer session migration. In CATS architectures, traffic is dynamically steered to optimal endpoints based on real-time network conditions, server load, and computational resource availability.The scenario is shown as Figure 10, and the transmission process of packets is shown in Figure 11.

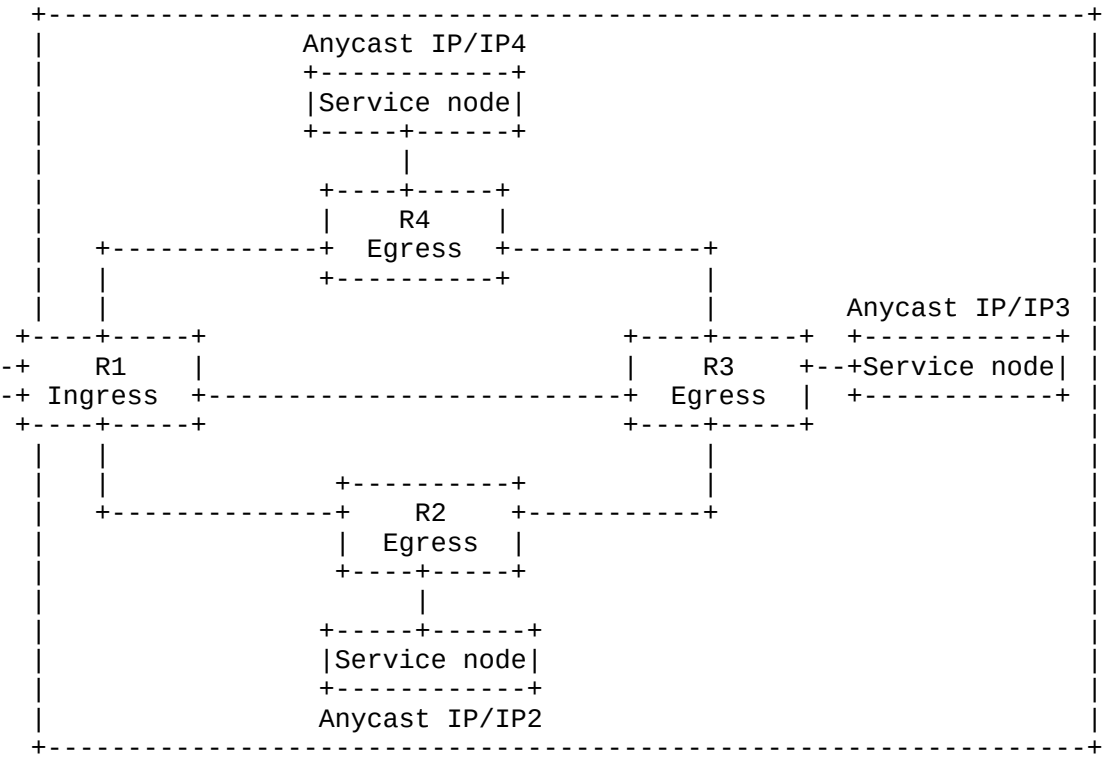


Figure 10: The Computing-Aware Traffic Steering (CATS) scenario

Customer A and customer B want to access the same service. For customer A, the packet will firstly be transmitted to the corresponding anycast IP address. The ingress will determine the optimal service node for customer A based on the access cost, computing resources of each service node, and the scheduled computing resource scheduling algorithm. Similar processing will be performed when customer B accesses the same service.

When customer A accesses to the service, it presents the following steps:

* Step 1: Customer A access to the service. It sends a initial ‘ClientHello‘ message which includes the ‘migration_support‘ extension to R1. The destination address of this packet is set to the anycast IP address of this service (IPs).

* Step 2: R1 schedules the customer A’s service connection request according to the real-time status of the network and computing resources, and determine that the server (IP address = IP4) will provide services to customer A.

* Step 3: the server completes a standard TLS 1.3 handshake.

* Step 4: the server sends a ‘NewSessionTicket‘ message to enable standard PSK-based session resumption. It carry the ‘MigrationToken‘, an authorization credential that refers to IP4.

* Step 5: customer A re-establishes the connection to server through IP4.

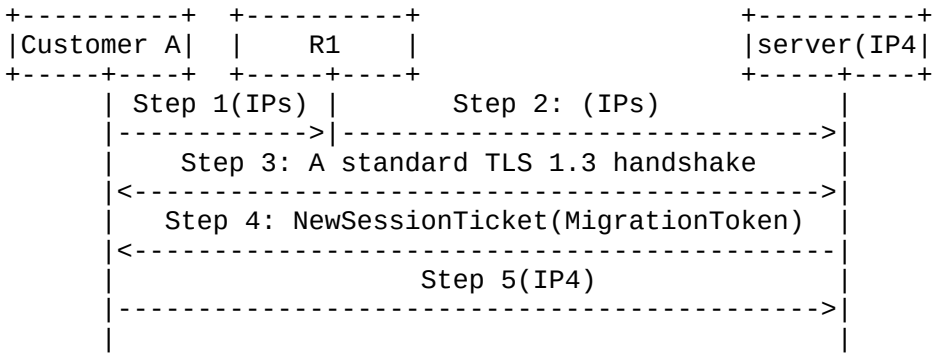


Figure 11: Procedures for the service affinity solution

In the whole process, devices in the network only need to broadcast the information of the computing network <Anycast IP Address, Service node Status>, and perform optimized scheduling of computing network resources according to this information.

Comparing to the existing solutions such as maintaining the customer-based connection status table in network devices, HTTP redirection and DNS redirection, this solution can avoid the waste of resources caused by saving a large amount of customer status data in the network devices, and realize the optimized scheduling of resources based on network conditions and computing resources in the computing-aware traffic steering scenario, so as to realize the reasonable operation of network resources, cloud resources and computing resources.

8. new section

8.1. Threat model

8.2. Informal security goals

This section defines the informal security goals required for the TLS Service Affinity mechanism. These goals aim to ensure data integrity, server authentication, and replay protection during the session migration process.

1) Integrity: Integrity of the MigrationToken MUST be guaranteed. An adversary MUST NOT be able to modify the server identifier (e.g., Server IP B) or the expiration time within the MigrationToken without detection.

2) Server Authentication: Authentication of the Target Server MUST be guaranteed. The client MUST verify that the server providing the MigrationToken is a legitimate member of the server cluster and not a malicious third party.

3) Freshness and Replay Protection: Freshness of the MigrationToken MUST be guaranteed. The client MUST ensure that the stored MigrationToken was issued in a recent session to prevent an adversary from replaying old, expired tokens.

8.3. Framing Layer Security

8.3.1. Confidentiality and Integrity

The Framing Layer operates entirely within the TLS encrypted channel. All frame headers and payloads are protected by TLS record-layer encryption and authentication. An attacker who cannot break the TLS cipher suite cannot read, modify, or inject frames.

8.3.2. Sequence Number Manipulation

Sequence numbers are transmitted in the clear within the TLS record (i.e., they are encrypted but the receiver must be able to read them). Since the entire frame is protected by TLS, an attacker cannot modify sequence numbers without detection. Implementations SHOULD verify that received sequence numbers are within an acceptable window to detect potential protocol-level attacks.

8.3.3. Denial of Service via Send Queue Exhaustion

A malicious receiver could refuse to send ACK frames, causing the sender's send queue to grow until it reaches FRAME_MAX_QUEUE. At that point, the sender MUST stop accepting new application data. To mitigate this, implementations SHOULD:

* Monitor send queue utilization and log warnings when it approaches capacity.

* Provide a configurable maximum queue size appropriate for the deployment environment.

* Consider a timeout after which unacknowledged frames are discarded and the connection is terminated with an error.

8.3.4. Retransmission Flooding

A malicious client could retransmit a large number of frames to a new server after migration. The new server SHOULD apply rate limiting to retransmitted frames and SHOULD validate that retransmitted frame sequence numbers are within the expected range. Since retransmitted frames carry the RETRANSMIT flag, the server can apply differentiated rate limiting policies for new vs. retransmitted frames.

9. IANA Considerations

This document requires IANA to allocate new codepoints from the following TLS registries, as defined in [RFC8446]:

1. From the "TLS ExtensionType Values" registry for 'migration_support' and 'migration_token'. This document suggests the values TBD1 and TBD2.

2. From the "TLS Alert Registry" for the 'migrate_notify' alert. This document suggests the value TBD3.

3. From the "TLS ExtensionType Values" registry for framing_layer. This document suggests the value TBD4. This extension, when present in ClientHello, indicates that the client supports the Reliable Framing Layer defined in Section 6. The extension_data is zero-length:

```
struct { } FramingSupport;
```

10. Normative References

[I-D.ietf-cats-usecases-requirements]
Yao, K., Contreras, L. M., Shi, H., Zhang, S., and Q. An, "Computing-Aware Traffic Steering (CATS) Problem Statement, Use Cases, and Requirements", Work in Progress, Internet-Draft, draft-ietf-cats-usecases-requirements-14, 2 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-cats-usecases-requirements-14>.

[I-D.li-cats-attack-detection]
Zhou, H., Wang, W., and S. Deng, "Computing-aware Traffic Steering for attack detection", Work in Progress, Internet-Draft, draft-li-cats-attack-detection-01, 8 April 2024, <https://datatracker.ietf.org/doc/html/draft-li-cats-attack-detection-01>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

Authors' Addresses

Wei Wang
China Telecom
Beiqijia Town, Changping District
Beijing
Beijing, 102209
China
Email: weiwang94@foxmail.com

Aijun Wang
China Telecom
Beiqijia Town, Changping District
Beijing
Beijing, 102209
China
Email: wangaj3@chinatelecom.cn

Zongbin Wang
Beijing Infosec Technologies Co., LTD.
No. 2 Building, Jinyu Science and Technology Park, Xisanqi, No. 6 Jianfeng Road (Nanyan), Haidian District
Beijing
China
Email: wangzb@infosec.com.cn

Mohit Sahni
Palo Alto Networks
San Francisco
Email: msahni@paloaltonetworks.com

Ketul Sheth
Palo Alto Networks
San Francisco
Email: ksheth@paloaltonetworks.com