NETWORK WORKING GROUP

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 02 September 2026

C.P. Ross
Independent

02 March 2026

Mercurius Window System (MWS)
draft-ross-mercurius-00

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 document is an individual submission to the IETF. Distribution of this document is unlimited.

The latest version of this draft can be found at: https://mercurius.tebibyte.org/draft-ross-mercurius-00.txt

Author’s Address

Christopher Ross
Independent
Email: chris@tebibyte.org
Project Website: https://mercurius.tebibyte.org

Abstract

The Mercurius Window System (MWS) is a networknative, serverside rendering system that enables graphical sessions to be accessed remotely with explicit semantics for windows, input, and session state. MWS allows a user to interact with a workstation from untrusted or resourceconstrained client devices without exposing application data, GPU workloads, or compositor state to those devices. The protocol defines a zerotrust client model, a structured session and window architecture, and a transport profile based on SCTP multistreaming. This document specifies the MWS architecture, message formats, transport requirements, and security model.

Executive Summary (NonNormative)

The Mercurius Window System (MWS) is a secure, highperformance window system for environments where applications run on one machine but users work from another. Whether the host is a personal workstation, a shared server, a large deployment, or a private cloud, MWS provides responsive access from any location without assuming that “local” networks or client devices are trustworthy.

MWS is designed for modern mobility patterns: consultants, remote workers, and digital nomads who move between client sites, hotels, airports, and home offices. These users often rely on lightweight laptops or tablets that may be lost, stolen, or

compromised. MWS ensures that possession of a client device is never sufficient to access the workstation. Authentication requires explicit user presence, and no sensitive data, credentials, or GPU workloads reside on the client.

MWS preserves the network transparency that made X11 valuable, whilst replacing its implicit trust and CPUbound rendering with a modern, zerotrust, GPUaccelerated architecture. The result is a window system that feels local even when the GPU is in another room, another building, or another country, supporting workflows ranging from office productivity to highrefreshrate interactive applications on modern networks.

Table of Contents

1. Introduction

The Mercurius Window System (MWS), named for Mercurius, the Roman messenger god of swift communication, is a secure window system for both local and remote use. A user may work directly at the console of a workstation as on a conventional Unix-like desktop, with full access to its GPU, input devices, and local display. The same session may also be accessed from lightweight, mobile, or untrusted client devices elsewhere, without replicating the workstation’ s software environment or exposing its data or GPU resources. Compute, storage, and rendering remain on the workstation; clients act solely as authenticated display and input endpoints.

This model supports both traditional workstation usage and modern mobility patterns. A user may begin work at a powerful machine in the office and later continue the same session from a laptop, thin client, or secondary desktop in another location, without maintaining multiple environments or synchronising state. Remote access is an extension of the local workstation rather than a separate mode of operation.

MWS is not a local display protocol, nor a pixelstreaming system. It defines a structured, messageoriented protocol for presence, input, and rendering state on a workstation. Rendering is serverresident and GPUaccelerated, and the protocol transmits structured commands rather than framebuffers. The transport is agnostic but optimised for SCTP’ s multistream, messageoriented semantics.

This architecture continues the lineage of early Unix window systems such as X11, which supported networktransparent interaction with applications running on central servers, while applying modern zerotrust security, authenticated multistream transport, and GPU isolation. Earlier systems such as NeWS explored serverside rendering but lacked the transport and security mechanisms required for contemporary workloads. Wayland, by contrast, is intentionally scoped to trusted local compositing and does not address remote GPUs, untrusted clients, or relocatable sessions. MWS occupies a distinct design space: secure, zerotrust remote presence for serverresident graphical environments, without compromising firstclass local console use.

1.1. Scope and Applicability

This document specifies the Mercurius Wire Protocol (MWP), the transportlevel protocol used by MWS to establish, authenticate, and maintain a user’ s graphical presence on a workstation. MWP defines message framing, capability negotiation, session attachment, and input/output semantics.

MWS is intended for environments where:

- applications execute on a central workstation or server
- users may work locally at the console or remotely from other devices
- clients may be untrusted, mobile, or ephemeral
- users may relocate sessions across devices
- GPUaccelerated workloads must remain serverresident
- network transparency is a firstclass requirement
- loss or theft of a device must not compromise workstation security.

MWS does not replace local display protocols such as Wayland, nor does it extend them. It provides a complementary mechanism for secure local and remote presence in multiuser and distributed environments where local display protocols do not apply.

1.2. Design Rationale

Early Unix window systems, including X11, were explicitly designed for network transparency: applications executed on powerful central servers while users interacted from remote terminals. This model proved valuable in multiuser and distributed environments, but X11’ s permissive trust model, unrestricted client capabilities, and CPUbound rendering are incompatible with modern zerotrust requirements and GPUaccelerated workloads.

Wayland addresses these issues by assuming a single trusted local compositor, a local GPU, and a singleuser environment. This model provides excellent performance on personal workstations but does not support remote GPUs, relocatable sessions, multiuser deployments, or untrusted clients. These use cases lie outside Wayland’ s design goals.

MWS intentionally revives and modernises the networktransparent workstation model. It retains the architectural advantages of centralised execution and remote interaction while adopting a zerotrust security model based on mutual TLS [RFC8446], authenticated SCTP [RFC9260] streams, and perclient GPU isolation. Rendering is serverresident and GPUaccelerated; clients transmit structured rendering commands rather than framebuffers. All compositor policy, input routing, and window management occur on the server, ensuring multiuser correctness and preventing privilege escalation.

Crucially, possession of a client device is never sufficient to access the workstation. Client certificates must be protected by the platform, session resume requires fresh authentication, and no sensitive data or credentials are stored on the client. This ensures that a lost or stolen laptop cannot be used to compromise the workstation.

The result is a window system that provides deterministic semantics, strong isolation, and relocatable sessions, enabling users to inhabit remote workstations with the performance and responsiveness of a local environment, whilst preserving firstclass local console operation.

1.3. Cloud and Distributed Computing Context

Many organisations operate private cloud or workstationcluster environments where users access centralised compute and GPU resources from thin clients or mobile devices. Public cloud deployments exhibit similar characteristics: applications execute on remote servers while clients roam across untrusted networks.

MWS aligns with this model by centralising execution and distributing only the user interface. This avoids the inefficiencies of distributed compute systems whilst preserving the benefits of remote access, session mobility, and strong isolation between users. Because clients are untrusted, MWS ensures that compromise or loss of a client device does not grant access to the workstation.

1.4. HighPerformance Rendering and Gaming

MWS is primarily intended for workstation and privatecloud deployments in which terminals connect over wellprovisioned LANs and VPNs, typically with DANE [RFC6698] securing mutually authenticated transport. In these environments, modern GPUs provide hardware accelerated AV1 encoding with extremely low latency, enabling highresolution and highrefreshrate streaming for everyday workstation workloads.

Nevertheless, the same architecture could accommodate high performance remote rendering workloads, including interactive 3D applications and games. Support for such workloads is a stretch goal rather than a primary target, but these use cases inform the design of the transport, security, and rendering model to ensure that MWS remains viable for demanding graphical applications.

1.5. Terminal Requirements and Wireless Considerations

MWS clients are treated as untrusted endpoints. Practical deployments assume a minimum level of capability. A typical terminal is expected to provide a modern CPU, a hardwareaccelerated GPU capable of AV1 decoding, and at least gigabitclass network connectivity. Higher resolutions or refresh rates benefit from greater bandwidth, but MWS remains usable at reduced quality on lowercapacity links.

Emerging wireless standards such as WiFi 8 (IEEE 802.11bn) are expected to deliver multigigabit throughput and submillisecond airinterface latency, enabling MWS terminals to operate over wireless links without compromising interactive performance.

1.6. Session Mobility and Detachable Operation

Because sessions in MWS are serverresident and independent of client connections, the system naturally supports detachable operation. A user may disconnect from one terminal and later resume the same session from another device, with all windows, GPU state, and compositor context preserved.

This model is conceptually similar to terminal multiplexers such as screen or tmux, but applied to a full GPUaccelerated graphical environment. Session mobility is a core design goal of MWS and informs its authentication, transport, and rendering architecture.

2. Conventions Used in This Document

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in RFC2119 and RFC8174 when, and only when, they appear in all capitals.

Terminology relating to Vulkan follows the definitions and naming conventions of the Vulkan 1.4 specification [VK14].

Terminology relating to SCTP follows RFC9260.

Unless otherwise stated:

“workstation” refers to the system on which applications execute, and where all rendering, compositing, and session management occur.

“server” refers to the MWS server daemon (mwsd) running on the workstation. In this document, “workstation” and “server” refer to the same system at different levels of abstraction.

“client device” refers to the physical device used by a user to access the workstation. Examples include laptops, tablets, phones, thin clients, and embedded devices.

“client” refers to the MWS client daemon running on the client device. The client is the protocol endpoint that communicates with the server.

“terminal” refers to the logical role a client assumes once connected: a seatproviding endpoint that delivers input and receives rendered output.

“session” refers to a persistent graphical environment maintained on the workstation independently of client connections.

“seat” refers to a set of input devices and output mappings associated with a session and bound to a terminal.

“surface” refers to a drawable region managed by the compositor and rendered by the workstation’s GPU.

“command stream” refers to the structured, Mercurius protocol messages exchanged on SCTP stream 0 (control stream).

“video surface” refers to a highmotion region encoded using a hardwareaccelerated codec such as AV1.

The names Alice, Bob, Eve, and Mallory are used in their standard roles from security literature. Alice and Bob denote honest users, Eve denotes a passive eavesdropper, and Mallory denotes an active attacker. These names are used solely for threatmodel examples and do not correspond to real users or implementation artefacts.

All multibyte integers are transmitted in network byte order unless explicitly specified otherwise.

3. System Architecture

This section provides a highlevel overview of the Mercurius Window System (MWS). It describes the conceptual model, major components, and architectural principles that inform the detailed design in Section 4 and the protocol specification in Section 5.

MWS is designed around a workstationcentric model in which all rendering, compositing, session management, and windowmanagement policy reside on a central server. Client devices act solely as authenticated display and input endpoints. This model preserves the semantics of a local workstation while enabling secure remote presence across modern networks.

MWS assumes client devices with at least gigabitclass connectivity, including modern WiFi networks that routinely exceed 1Gb/s. The protocol is optimised for 10GbE LANs, where uncompressed or lightly compressed surfaces and highmotion content can be delivered with minimal latency. Devices with substantially lower bandwidth may operate at reduced quality but are not a primary design target.

3.1. Architectural Principles

The design of MWS is guided by the following principles:

Applications execute on a central workstation or server.
Local and remote interaction share identical session semantics.
Client devices may be untrusted, mobile, or ephemeral.

Users may relocate sessions across devices without restarting applications.
GPUaccelerated workloads remain serverresident.
Network transparency is a firstclass requirement.
Loss or theft of a client device must not compromise workstation security.

These principles reflect the goal of treating the workstation as a longlived environment with continuity of storage, configuration, and identity.

Alternative transports such as QUIC were considered. However, SCTP's native multistreaming, messageoriented delivery, and support for partial reliability align directly with the requirements of MWS. QUIC's multiplexed bytestream model, together with the absence of partially reliable streams, would require additional framing and scheduling logic to emulate SCTP semantics. For these reasons, SCTP is the primary transport for MWS.

The protocol's guarantees depend on transport properties that SCTP provides natively, including independent ordered streams, preservation of message boundaries, optional partial reliability, avoidance of crossstream headofline blocking, and stable associations. These properties are required to ensure deterministic compositor behaviour, responsive input under load, support for highmotion video surfaces, relocatable sessions, and multiseat concurrency.

TCP does not provide these properties without substantial additional protocol machinery. A TCPbased transport would therefore be unable to meet the latency, isolation, and concurrency requirements of MWS as defined in this document, and is out of scope for this specification.

3.2. Major Components

MWS consists of the following major components:

The compositor, which manages windows, focus, input routing, and presentation.

The renderer, which executes GPUaccelerated drawing and performs surface composition on the workstation.

The session manager, which maintains user sessions independently of client connections and supports detachable operation.

The transport layer, which provides a secure, multistream, messageoriented channel between workstation and client.

The client device, which decodes surfaces, presents them to the user, and forwards input events to the workstation.

These components interact to provide a deterministic, structured, zerotrust window system suitable for both local console use and remote presence.

3.3. WorkstationCentric Model

The workstation is the authoritative environment. It owns all GPU resources, input devices, and compositor state. Applications run exclusively on the workstation, and all rendering is performed on the workstation's GPU.

A user may interact with the workstation in two ways:

Local console mode, using the workstation's own keyboard, pointer, and display.

Remote presence mode, using an authenticated client device elsewhere on the network.

Local and remote interaction share the same compositor, window tree, and session state. Remote presence is an extension of the local workstation, not a separate mode of operation. Sessions persist across transient disconnections, but longterm persistence requires explicit detachment.

3.4. Rendering and Surface Model

MWS supports two classes of graphical output:

structured rendering commands, representing deterministic drawing operations for lowmotion or vectororiented surfaces

video surfaces, representing highmotion content encoded using hardwareaccelerated codecs such as AV1

The compositor selects the appropriate representation based on surface characteristics and available bandwidth. This allows MWS to operate efficiently on both 1GbE and 10GbE networks, as well as on modern WiFi links.

3.5. Session Model

Sessions are serverresident and persist independently of client connections, but longterm persistence requires explicit detachment. MWS tolerates transient network interruptions; if a client reconnects within the configured grace period, the session continues without interruption. However, if a client disappears without detaching, the session is preserved only for the duration of this grace period. Once the period expires, the session is closed, and applications terminate in the same manner as a workstation session without an active seat.

A user may explicitly detach a session to preserve it beyond the reconnection grace period and may later resume it from any authorised device. A single transport association may provide multiple seats for a session when permitted by policy. The precise rules governing how many transport associations may attach to a session, and under what conditions, are defined in Section 4.7.

This model enables mobility across devices while preserving the semantics of a traditional workstation and avoiding longlived orphaned sessions.

3.6. ZeroTrust Client Model

All client devices are treated as untrusted endpoints, even on local LANs. Trust is established exclusively through cryptographic identity and explicit authorisation rather than network location. A client device is not an identity and is not authorised to access a session by virtue of its presence on the network; only the user is authorised.

Historically, X11 enabled a powerful and flexible model in which users could inhabit remote workstations as naturally as local ones. Even on dedicated X Terminals, users authenticated as themselves and could not access another user's data. The flaw was not the login model but the assumption that any client on the network was inherently trustworthy. Wayland addressed this by

eliminating remote clients entirely. Mercurius instead removes the assumption of trust: client devices may be anywhere, but trust is derived solely from what the user can cryptographically prove.

Device identity is established during the DTLS/mTLS handshake. When DANE is deployed, the client’s certificate is validated against DNSSECprotected TLSA records, ensuring that only devices explicitly provisioned by the domain administrator may initiate a connection. Device authentication alone is insufficient to access a session.

User authentication is performed at the application layer using the mechanismagnostic model defined in Section5.2.1. The server advertises supported mechanisms (e.g., “DANE”, “PAM”, “PASSWORD”, “FIDO2”), and the client selects one. This allows deployments to integrate passwordbased, hardwaretoken, federated, or certificate based user authentication without modifying the protocol.

A client device is assumed to be mobile and at risk of loss or theft. To limit the impact of device compromise, a client stores no confidential data, longterm session state, or reusable credentials. Authentication requires both possession of the device’s private key (validated via mTLS and optionally DANE) and successful user authentication via one of the advertised mechanisms. Mallory stealing Bob’s laptop gives him no more access to the workstation than if he had purchased a brandnew laptop; the device alone is insufficient to access or resume a session.

If a client disconnects unexpectedly, the session persists only for the duration of the reconnection grace period unless the user has explicitly detached. After this period, the session is closed and applications terminate.

In contrast to traditional window systems such as X11, client devices in MWS are not part of the trusted computing base; they are merely display and input conduits for a workstationresident session.

3.7. Network Considerations

MWS is designed to operate over untrusted IP networks, including public networks and variablequality wireless links. The protocol does not assume that terminals are located on the same LAN as the workstation, nor that any network segment provides meaningful security. A terminal on a local LAN and a terminal on a remote network are treated identically by the workstation.

MWS is designed for modern networks:

- 10GbE provides optimal performance and headroom for multiple highresolution seats on a single workstation.
- WiFi 6/6E/7 provides multigigabit throughput with variable jitter and is fully supported for singleseat clients.
- 1GbE provides a usable baseline for typical desktop workloads and a small number of seats on a workstation.
- Subgigabit links are outside the primary design envelope and are not expected to provide an acceptable experience for highresolution, highrefresh workloads.

The transport layer adapts to available bandwidth through dynamic

surface encoding, selective use of video surfaces, adaptive refresh rates, and prioritised input and control streams.

The detailed architecture is specified in Section4, and the wire protocol is defined in Section5.

3.8. Transport Requirements

MWS requires a transport that provides structured, messageoriented delivery with support for multiple independently ordered channels. The transport MUST preserve message boundaries, MUST support concurrent streams with independent ordering, and SHOULD provide mechanisms for partial reliability to avoid retransmission of stale highvolume data such as video surfaces.

The transport MUST avoid crossstream headofline blocking. Input events, control messages, rendering commands, and video surfaces are logically independent flows, and the correctness of compositor behaviour depends on their timely and ordered delivery within their respective channels. A transport that enforces global ordering across all data would introduce latency coupling between these flows and would not meet the responsiveness requirements of MWS.

The transport MUST support stable associations that survive transient network changes, including client mobility across networks. Session attachment, reconnection semantics, and multiseat operation rely on the ability to maintain a consistent transportlevel association identity.

SCTP satisfies these requirements through its native multistreaming model, messageoriented delivery, optional partial reliability, and support for multihoming. These properties align directly with the architectural principles defined in Section 3.1 and are required for deterministic compositor behaviour, responsive input under load, support for highmotion video surfaces, relocatable sessions, and multiseat concurrency.

TCP does not provide these properties without substantial additional protocol machinery. TCP offers only a single inorder byte stream, lacks message boundaries, enforces global headofline blocking, and provides no support for partial reliability or multistreaming. A TCPbased transport would therefore be unable to meet the latency, isolation, and concurrency requirements of MWS as defined in this document, and is out of scope for this specification.

4. Detailed Architecture

The Mercurius Window System (MWS) is structured around a central server (mwsd) that owns all GPU resources, input devices, and compositor state, and a set of untrusted client devices that connect over a secure, messageoriented transport. Once connected, a client device acts as a terminal providing a seat. This section describes the architectural model of sessions, seats, windows, rendering, and compositor behaviour. The wire protocol and message formats are defined in Section 5.

4.1. Sessions

A session represents the complete graphical environment associated with a single authenticated user, including windows, workspaces, GPU resources, and compositor state. Sessions are serverresident and MAY persist independently of client

connections when explicitly detached.

User identity is established during the secure transport handshake. The client certificate subject is mapped to a local user account via PAM. All windows, seats, and compositor state created over that association belong to the resulting session.

4.2. Seats

A seat represents a set of input devices and an output binding for a session. A session MAY have multiple seats simultaneously. Each seat corresponds to a particular terminal, whether that terminal is the local console or a remote client device acting in the terminal role.

Input events are tagged with a seat_id, and the compositor routes them according to seatspecific focus and pointer state. Output mappings (e.g., which windows appear on which displays) may also be seatspecific.

4.3. Windows

Windows are servermanaged objects representing toplevel application surfaces. Each window belongs to exactly one session and is associated with one or more rendering surfaces (structured swapchains or video surfaces) depending on compositor policy.

Window identifiers are scoped to a session. A terminal MUST NOT reference or interact with windows belonging to any other session. The server MUST enforce this isolation and MUST reject or ignore any protocol message that attempts to target a window outside the authenticated session.

4.4. Compositor Model

The compositor maintains the global window tree, stacking order, focus, workspaces, and output mappings for each session. It is responsible for:

applying windowmanagement policy
routing input events based on seat and focus
managing swapchains and presentation timing
selecting between structured rendering and video fallback
revoking or reconfiguring windows according to policy

The compositor SHOULD expose a uservisible mechanism to forcibly terminate an unresponsive window. This mechanism is implementation defined (e.g., a “kill window” gesture similar to CtrlAltEsc in KDE).

The compositor MAY revoke swapchains, reconfigure windows, or migrate them between outputs according to local policy, resource constraints, or security requirements. When a swapchain is revoked, the server notifies the terminal and MAY substitute a placeholder or video surface.

4.5. Rendering Model

Rendering in MWS is serverside. Applications submit rendering commands to the server, which validates and executes them on the GPU. Client devices do not access GPU resources directly.

The compositor selects the appropriate representation for each surface:

structured rendering for lowmotion or interactive content

video surfaces for highmotion or bandwidthsensitive content

Because rendering is serverresident, a stalled or misbehaving terminal cannot block the compositor. The server MAY revoke a window's rendering resources, substitute a placeholder surface, or terminate the client if rendering deadlines are repeatedly missed.

4.6. Stream Allocation

MWS uses SCTP multistreaming to separate control traffic from rendering traffic and to prevent headofline blocking between independent windows. Stream allocation is defined as follows:

Stream0 is reserved for control messages and MUST NOT carry rendering data.

Each window is associated with exactly one rendering stream. All structured rendering commands and videosurface updates for that window are sent on its assigned stream.

Multimonitor configurations do not affect stream allocation. A window that spans multiple outputs continues to use a single rendering stream.

The compositor MAY allocate additional streams for specialised rendering contexts (e.g., offscreen surfaces or auxiliary swapchains), but these MUST be explicitly negotiated during window creation and are scoped to the window that requested them.

Streams are not reused across windows unless the compositor has explicitly revoked the prior window and returned the stream to the allocator.

This model ensures that rendering for one window cannot block or delay rendering for another, while avoiding unnecessary proliferation of streams in multimonitor environments.

4.6.1. Session Identity and Message Routing

Each SCTP association corresponds to exactly one client session. The server MUST treat the SCTP association identifier (assoc_id) as the authoritative session identity. No clientsupplied field may select, reference, or influence the session to which a message is delivered.

A session is created only after successful completion of the handshake defined in Section5. Until the handshake completes, the server MUST ignore all messages received on streams other than 0.

For any message received on a nonzero stream, the server MUST:

identify the session associated with the SCTP association

verify that the session is active

dispatch the message to the subsystem corresponding to the stream

reject or ignore the message if it is malformed or references resources outside the session

Messages referencing windows, seats, or other resources not owned by the session MUST be rejected with MWS_ERROR(type=600, fatal=0).

Messages referencing a session other than the one implied by the SCTP association MUST be ignored.

If a message is received for an association that has no active handshake or no active session, or whose session has been closed, the server MUST silently discard the message.

4.7. Session Lifecycle

A session is a longlived serverside construct that persists independently of any particular network connection. A session becomes ACTIVE when a client completes the handshake defined in Section5 and remains ACTIVE until it is explicitly terminated or reclaimed by policy.

A new client device connection MUST create a new session in the ACTIVE state unless the user explicitly requests to resume an existing session. The server MUST NOT automatically reattach a client device to a prior session solely on the basis of matching user identity.

A client MAY explicitly detach from an ACTIVE session. Detach transitions the session from ACTIVE to DETACHED. In the DETACHED state, the session continues to run without any attached transport association; its windows, compositor state, and GPU resources remain serverresident. DETACHED sessions MAY be resumed by any authenticated client device belonging to the same user, subject to server policy.

Loss of the SCTP association (e.g., network failure, timeout, client crash) while a session is ACTIVE does not immediately terminate the session. Instead, the server MUST transition the session to a GRACE state and start a reconnection grace timer. In the GRACE state, the session remains active but has no attached client. If a client reconnects and successfully resumes the session before the grace timer expires, the server MUST transition the session back to ACTIVE and the session continues without loss of state.

If the reconnection grace period expires without a successful resume, the server MUST treat the session as ABANDONED unless the user has explicitly detached it. ABANDONED sessions MUST be terminated and all associated resources reclaimed. Implementations MUST provide a configurable reconnection grace interval and SHOULD allow values sufficient to tolerate brief network outages on typical WiFi and WAN links. Servers SHOULD return a specific error status when a resume request targets an expired session.

Longterm persistence is an explicit, optin behaviour: a session continues to exist beyond the reconnection grace period only if the user has explicitly detached it or otherwise marked it for later resumption. Implementations MUST NOT retain ABANDONED or stale sessions indefinitely. The server SHOULD reclaim resources associated with inactive sessions according to local policy (e.g., idle timeout, logout event, or administrative limits).

Only one transport association (client daemon instance) MUST be attached to a given session at a time in the base protocol. That association MAY provide one or more seats for the session, subject to server policy. An implementation MAY provide a mechanism that allows additional associations to attach to the same session (for example, for technical support), but such behaviour is outside the scope of this specification and MUST NOT alter the semantics defined for the singleassociation model

above.

4.8. Session and Seat Model

Sessions MAY persist independently of client connections. When a client device disconnects, the associated session and its windows MAY remain active in either the GRACE or DETACHED state. The compositor MAY blank or lock the session’ s outputs according to local policy while no seat is attached.

When a user resumes a session (from GRACE or DETACHED), the server:

1. Binds a new seat_id to the resumed session.
2. Sends the current window list, geometry, and focus state.
3. Associates the new seat’ s outputs with the session.

MWS supports both independent sessions and multiseat attachment within a single session. A user may maintain multiple concurrent sessions (e.g., one on the local console and another accessed remotely), or may attach multiple seats to the same session via a single transport association, subject to the singleassociation rule in Section4.7.

MWS also supports explicit session detachment. A user may detach a running session, leaving its windows, compositor state, and GPU resources active on the server without any attached seats. The user may then initiate a new session on the same client device (e.g., to perform unrelated work) and later resume the detached session exactly where it was left. This behaviour is directly analogous to detaching and reattaching a GNU Screen or tmux session, but applied to a full graphical desktop environment spanning one or more seats.

4.9. Local Transport Profile (NonNormative)

Although MWS treats all client devices as untrusted endpoints and applies the same protocol semantics regardless of network location, implementations MAY apply transportlayer optimisations when the client device and server reside on the same physical host. These optimisations MUST NOT alter protocol semantics, message ordering, authentication requirements, or session isolation, and MUST remain transparent to the terminal.

Permitted implementationlevel optimisations include:

loopbackspecific SCTP acceleration
reduced cryptographic overhead
sharedmemory fast paths
GPUdirect resource sharing where supported

These optimisations MUST NOT:

grant additional privileges to local client devices
bypass certificate validation or PAM authentication
modify the behaviour of control, input, or rendering streams
introduce protocol features unavailable to remote client devices

MWS remains a transportagnostic, networktransparent window system. Local optimisations exist solely to ensure that client devices running on the same host as the server achieve performance comparable to traditional localonly systems without compromising the zerotrust security model.

4.10. Security Model

All client devices are treated as untrusted. Trust is established exclusively through cryptographic identity and explicit authorisation rather than network location. The server enforces strict isolation between users, sessions, seats, and windows. In particular:

- each session is bound to a single SCTP association, and the association identifier serves as the authoritative session identity
- window identifiers are scoped to a session and cannot be referenced by other sessions
- input events are scoped to a seat and session, and cannot target windows outside that session
- clients cannot observe, enumerate, or reference resources belonging to other sessions
- all clientoriginated messages are validated before being processed

Transport security is provided by DTLS with mutual authentication. When DANE is deployed, the client's certificate is validated against DNSSECprotected TLSA records, ensuring that only devices explicitly provisioned by the domain administrator may initiate a connection. Device authentication alone does not grant access to a user session.

MWS does not mandate a specific encryption mechanism beyond requiring confidentiality, integrity, and mutual authentication of endpoints. Deployments SHOULD use a transport that provides forward secrecy, such as DTLS 1.3, WireGuard, or IPsec with PFSenabled cipher suites. The choice of transportlayer security does not affect protocol semantics, and MWS remains agnostic to whether encryption is provided by DTLS or by an external secure tunnel.

User authentication is performed at the application layer using the mechanismagnostic model defined in Section5.2.1. The server advertises supported mechanisms (e.g., "DANE", "PAM", "PASSWORD", "FIDO2"), and the client selects one. This separation of device and user identity ensures that compromise of a device does not grant access to a user's session without the corresponding user credential.

The server validates all clientoriginated messages, including window creation, input events, and rendering commands. A client may not reference windows, sessions, or resources outside its authenticated session. Attempts to do so are rejected with MWS_ERROR(type=600, fatal=0). Malformed or semantically invalid messages are ignored, and the session continues unless the error is marked fatal.

The client is not part of the trusted computing base. It stores no confidential data, longterm session state, or reusable credentials. If a client disconnects unexpectedly, the session persists only for the duration of the reconnection grace period unless the user has explicitly detached. After this period, the session is closed and applications terminate.

Loss of the transport association for any reason (network failure, timeout, endpoint crash) is treated as a fatal transport error. The session MAY persist according to the rules in Section 4.7 and MAY be resumed from another client device subject to policy.

5. Protocol Specification

5.1. Message Framing

All MWS messages consist of a fixed-size header followed by an optional payload. Messages are carried within a single SCTP user message and MUST NOT be fragmented across multiple SCTP user messages.

The header format is:

```
struct MwsHeader {
   uint32_t magic;      // MWS_MAGIC_VALUE
   uint16_t type;       // MWS_* opcode
   uint16_t reserved;   // MUST be zero
   uint32_t length;     // payload length in bytes
};
```

The payload immediately follows the header. Implementations MUST validate the magic value, type, and length before processing the payload. Messages with invalid headers MUST be rejected with MWS_ERROR(type=600).

5.2. Control Messages (Stream 0)

Control messages manage authentication, session establishment, window lifecycle, and compositor state. All control messages MUST be sent on SCTP Stream0.

The control channel is strictly ordered and defines the protocol state machine for session creation, resumption, and teardown. Rendering, input, and video fallback streams operate independently and are not blocked by controlplane latency.

5.2.1. Initial Handshake (001099)

The initial handshake establishes protocol version, user identity, and session parameters. Mutual TLS (mTLS), optionally validated using DANE (Section 9.1), authenticates the client device at the transport layer. The applicationlayer handshake authenticates the user and establishes a session.

User authentication is mechanismagnostic. The server advertises one or more supported authentication mechanisms, and the client selects one. This allows deployments to integrate PAM, WebAuthn, FIDO2, Kerberos, OAuth2, or future mechanisms without modifying the protocol.

The handshake proceeds as follows on SCTP Stream 0:

1. MWS_QUERY (type=001) — Client → Server
   Initiates protocol negotiation and requests session parameters.

2. MWS_AUTH_CHALLENGE (type=002) — Server → Client
   Advertises the available authentication mechanisms. The payload contains a list of mechanism identifiers (e.g., "PAM", "PASSWORD", "FIDO2", "WEBAUTHN"). The server MAY advertise multiple mechanisms.

   The payload of MWS_AUTH_CHALLENGE has the following format:

   ```
   uint8_t mechanism_count;

   repeated mechanism_count times:
       uint8_t name_len;
       char    name[name_len];
   ```

Mechanism names are UTF8 strings and are not NULterminated. name_len specifies the length in bytes of each name and MUST be greater than zero. The payload MUST be exactly long enough to contain all advertised entries; extra bytes are a protocol error.

mechanism_count MAY be zero. In this case the server is indicating that no authentication mechanisms are available, and the client MUST treat the challenge as a fatal configuration error and abort the handshake.

Mechanism names are casesensitive identifiers drawn from the "MWS Authentication Mechanisms" registry (Section 9.2).

3. MWS_AUTH_RESPONSE (type=003) — Client → Server
   Selects an authentication mechanism and provides credentials or mechanismspecific data. The payload contains:

   selected mechanism identifier
   mechanismspecific credential payload

   The server MUST reject responses that specify a mechanism not offered in MWS_AUTH_CHALLENGE.

   The generic payload format of MWS_AUTH_RESPONSE is:

```
uint8_t  mech_name_len;
char     mechanism[mech_name_len];
uint16_t credential_len;
uint8_t  credential[credential_len];
```

   mechanism is a UTF8 identifier that MUST exactly match one of the mechanism names advertised in the preceding MWS_AUTH_CHALLENGE. mech_name_len specifies the length in bytes of the mechanism identifier and MUST be greater than zero.

   credential is an opaque mechanismspecific blob whose internal structure is defined by the selected mechanism. Implementations MUST NOT inspect or reinterpret credential except as specified by the mechanism definition.

   If mechanism does not correspond to an advertised entry, or if credential_len does not match the actual payload length, the server MUST treat this as a protocol error and respond with MWS_ERROR(type=600, fatal=1).

4. MWS_SURFACE_CAPS (type=004) — Server → Client
   Returns surface and WSI capability information, including supported Vulkan extensions, swapchain formats, and presentation modes.

5. MWS_SESSION_INFO (type=005) — Server → Client
   Returns session parameters, including:

   session identifier
   initial compositor state
   seat and input configuration
   resume token (if applicable)

After successful completion of this sequence, the client is fully authenticated and may create windows or resume an existing session on Streams 115.

All multibyte integer fields in MWS messages are encoded in network byte order (bigendian). Implementations MUST convert

values to and from host byte order when constructing or parsing messages. Structures shown in this document illustrate field layout only and do not imply host endianness.

5.2.1.1 Session Identifier Semantics

The session identifier returned in MWS_SESSION_INFO is assigned solely by the server. Clients MUST treat this value as opaque and MUST NOT attempt to select, predict, or construct session identifiers. All client-originated messages that include a session_id field are advisory; the server MUST validate the session_id against the session associated with the SCTP association on which the message was received.

A client MUST NOT assume that a session identifier remains valid across reconnects unless the server has explicitly offered the session for resumption. Session identifiers from terminated or reclaimed sessions MUST NOT be reused by the server.

Resume tokens included in MWS_SESSION_INFO are hints that allow clients to correlate local state with resumable sessions. Resume tokens are not client-authoritative and MUST be validated by the server during MWS_SESSION_RESUME_REQUEST processing.

5.2.2. Session Management (100199)

Session resume allows a client to reattach to an existing session previously detached by the user.

MWS_SESSION_RESUME_OFFER (type=100) — Server → Client
Indicates that a resumable session exists for the authenticated user.

MWS_SESSION_RESUME_REQUEST (type=101) — Client → Server
Requests resumption of the indicated session.

MWS_SESSION_RESUME_COMPLETE (type=102) — Server → Client
Confirms that the session has been resumed and provides updated compositor state.

5.2.2.1 Resume Semantics

A session becomes resumable when the user has explicitly detached it or when the SCTP association has been lost and the session has entered the reconnection grace period defined in Section4.7. The server MUST NOT offer resumption for sessions that have been terminated or reclaimed by policy.

After successful user authentication, the server MUST send MWS_SESSION_RESUME_OFFER (type=100) if and only if one or more resumable sessions exist for the authenticated user. The offer includes a list of resumable session identifiers and MAY include metadata such as creation time, last activity time, or compositor state summary. If no resumable sessions exist, the server MUST NOT send a resume offer.

To resume a session, the client sends MWS_SESSION_RESUME_REQUEST (type=101) specifying the session identifier. The server MUST validate that the requested session:

belongs to the authenticated user
is currently resumable
is not attached to another client device, unless local policy permits forced detachment

If validation succeeds, the server MUST attach the client to the

session, cancel any active reconnection grace timer, and send MWS_SESSION_RESUME_COMPLETE (type=102) containing the updated compositor state.

MWS_SESSION_RESUME_COMPLETE MUST include a complete reconstruction of session state sufficient for the client to synchronise its local representation, including the window list, geometry, stacking order, focus state, and seat/output mappings.

If validation fails, the server MUST reject the request with MWS_ERROR(type=600, fatal=0) and MUST NOT reveal the existence or attributes of sessions belonging to other users.

Resume tokens provided in MWS_SESSION_INFO are advisory hints that allow clients to identify resumable sessions across reconnects. Resume tokens are not client-authoritative; the server MUST validate all resume requests against its internal session table.

Only one client device MAY be attached to a session at a time. If a second device attempts to resume an active session, the server MUST either reject the request or forcibly detach the existing client, according to local policy.

5.2.3. Window Lifecycle (200299)

Window creation, destruction, mapping, and configuration are managed through the following messages:

- MWS_CREATE_WINDOW (type=200) — Client → Server
  Requests creation of a new toplevel window. The window is created within the session associated with the SCTP association on which the request was received.
- MWS_WINDOW_CREATED (type=201) — Server → Client
  Confirms window creation and returns window_id and initial geometry.
- MWS_DESTROY_WINDOW (type=202) — Client → Server
  Requests destruction of a window owned by the session.
- MWS_WINDOW_DESTROYED (type=203) — Server → Client
  Confirms destruction of a window.
- MWS_MAP_WINDOW (type=204) — Client → Server
  Requests that a window become visible.
- MWS_UNMAP_WINDOW (type=205) — Client → Server
  Requests that a window become hidden.
- MWS_CONFIGURE_WINDOW (type=206) — Server → Client
  Notifies the client of geometry or state changes.
- MWS_FOCUS_WINDOW (type=207) — Server → Client
  Notifies the client that a window has gained or lost focus.
- MWS_SWAPCHAIN_REVOKED (type=208) — Server → Client
  Indicates that a window’ s swapchain has been revoked due to policy, timeout, or resource constraints.

5.2.3.1. Window Identifier Scope

Window identifiers are scoped to the session that created them. A client MUST NOT reference, manipulate, or query windows belonging to any other session. The server MUST validate that all window-related messages refer to windows owned by the session

associated with the SCTP association on which the message was received.

If a client attempts to reference a window outside its session, the server MUST reject the message with MWS_ERROR(type=600, fatal=0). The server MUST NOT reveal the existence, geometry, focus state, or any other attributes of windows belonging to other sessions.

Window identifiers are never reused across sessions, and the server MUST ensure that identifiers from one session cannot collide with or be interpreted as identifiers from another. Implementations MAY use per-session identifier namespaces, randomised identifiers, or any other mechanism that guarantees isolation.

These rules ensure that windows are private to the session that owns them and that clients cannot observe or interfere with the graphical state of other users.

5.3. Rendering Commands (300399) — Stream 1

Rendering commands are delivered on SCTP Stream1. Commands are validated and executed by the server.

MWS_VK_SUBMIT (type=300) — Client → Server
Submits a VkCommandBuffer for execution.

MWS_VK_SYNC (type=301) — Client → Server
Requests synchronisation of GPU state.

MWS_VK_DESTROY (type=302) — Client → Server
Requests destruction of Vulkan resources associated with a window or pipeline.

5.4. Input Events (400499) — Stream 2

Input events are delivered on SCTP Stream2. All input events MUST be tagged with a seat_id and window_id.

MWS_INPUT_EVENT (type=400) — Client → Server
Delivers an XI2compatible input event.

MWS_INPUT_ACK (type=401) — Server → Client
Acknowledges receipt of an input event.

5.4.1. Pointer Motion Events

Pointer motion is reported using both absolute and relative coordinates. All motion events, including mouse_move and mouse_drag, MUST include the following fields:

x, y: absolute pointer position in surfacelocal pixel units (uint32)

delta_x, delta_y: relative motion since the previous pointer event, in signed pixel units (int16)

Terminals MUST send both absolute and relative values. The compositor uses absolute coordinates for hittesting and focus routing, and MAY use relative deltas for highprecision motion, gesture recognition, or subpixel accumulation. A delta of zero indicates no relative motion.

mouse_move
Generated when the pointer moves with no buttons held.

mouse_drag
Generated when the pointer moves while one or more buttons are held. Encoded identically to mouse_move, with the addition of a bitmask of currentlyheld buttons.

5.4.2 Input Scoping and Session Isolation

Input events are scoped to the session and seat from which they originate. A client MUST NOT send input events targeting windows belonging to any other session. The server MUST validate that the seat_id and window_id in every MWS_INPUT_EVENT refer to resources owned by the session associated with the SCTP association on which the message was received.

If a client attempts to deliver input to a window outside its session, the server MUST reject the message with MWS_ERROR(type=600, fatal=0). The server MUST NOT reveal the existence, geometry, focus state, or any other attributes of windows belonging to other sessions.

Each session owns exactly one logical seat unless additional seats have been explicitly negotiated. seat_id values are therefore scoped to the session and MUST NOT collide with or reference seats belonging to other sessions.

These rules ensure that input events cannot be redirected, spoofed, or injected across session boundaries, and that clients cannot observe or influence the input state of other users.

5.5. Video Fallback (500599) — Stream 3

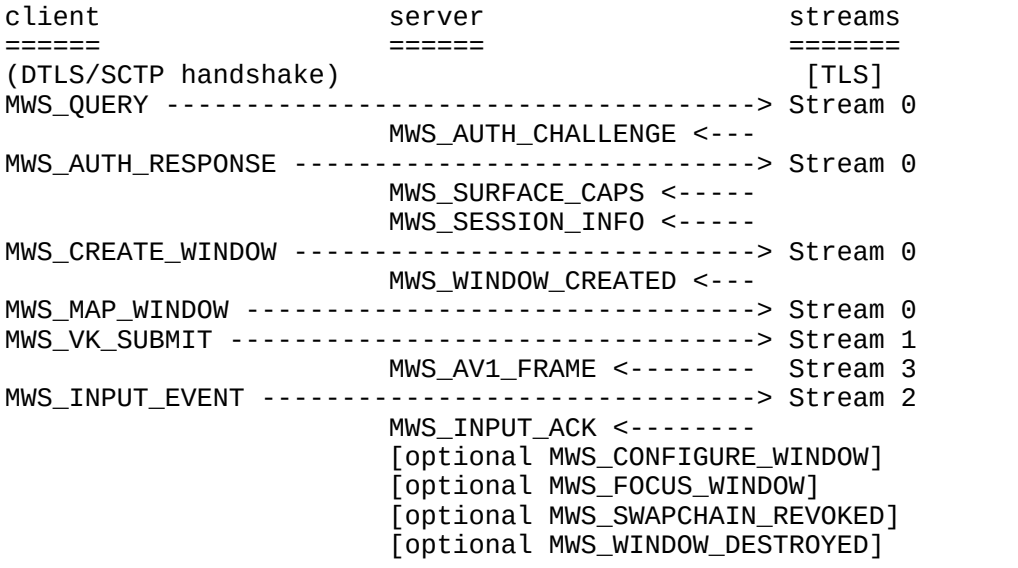
Video fallback is used when serverside rendering produces a pixel stream rather than a Vulkan command stream. Stream3 uses PRSCTP to allow frame drops under congestion.

MWS_AV1_FRAME (type=500) — Server → Client
Delivers an AV1encoded video frame.

MWS_PLACEHOLDER_FRAME (type=501) — Server → Client
Delivers a placeholder frame when rendering is unavailable.

5.6. Protocol State Machine

5.6.1. Initial Connection



5.6.2. Session Resume

```
client                          server
======                          ======
(DTLS/SCTP handshake)
MWS_QUERY ------------------->
          <------------------- MWS_SESSION_RESUME_OFFER
MWS_SESSION_RESUME_REQUEST -->
          <------------------- MWS_SESSION_RESUME_COMPLETE
```

5.7. WSI Extension (Surface Creation)

MWS defines a Vulkan WSI extension for creating surfaces associated with MWS windows. All WSI requests are validated by the server.

```
typedef struct VkMWSSurfaceCreateInfoMWS {
   VkStructureType  sType;
   const void*      pNext;
   VkDevice         device;
   uint32_t         session_id;
   uint32_t         sctp_stream_id;
   uint32_t         window_id;
   VkExtent2D       initial_extent;
} VkMWSSurfaceCreateInfoMWS;

VkResult mwsCreateMWSSurfaceMWS(
   VkInstance                          instance,
   const VkMWSSurfaceCreateInfoMWS*    pCreateInfo,
   const VkAllocationCallbacks*        pAllocator,
   VkSurfaceKHR*                       pSurface
);
```

Surface creation proceeds as follows:

1. The client calls vkCreateInstance(), receiving MWS_SURFACE_CAPS.
2. The client calls mwsCreateMWSSurfaceMWS() with session and window parameters.
3. The server validates the request and creates a VkSurfaceKHR bound to the specified window_id.
4. The client calls vkCreateSwapchainKHR() on the returned surface.

5.7.1. Surface Binding and Session Validation

The fields session_id, sctp_stream_id, and window_id in VkMWSSurfaceCreateInfoMWS are not client-authoritative. They are treated as requests that the server MUST validate against the session associated with the SCTP association on which the WSI request was received.

The server MUST enforce the following rules:

session_id MUST match the authenticated session associated with the SCTP association. A client MUST NOT request creation of a surface for any other session.

window_id MUST refer to a window owned by the same session. The server MUST reject any request that attempts to bind a surface to a window belonging to another session.

sctp_stream_id MUST match the rendering stream allocated to the specified window. The server MUST reject requests that specify an incorrect or unauthorised stream.

If any of these validations fail, the server MUST reject the request with MWS_ERROR(type=600, fatal=0). The server MUST NOT

reveal the existence, geometry, or state of windows belonging to other sessions.

These rules ensure that surface creation cannot be used to infer or access the graphical resources of other users, and that Vulkan surfaces remain correctly bound to the window and rendering stream allocated by the compositor.

5.7.2. Required Enums

```
#define VK_STRUCTURE_TYPE_MWS_SURFACE_CREATE_INFO_MWS 1000053000
```

5.8. Error Handling (600699)

Errors are reported using the MWS_ERROR message. Errors are asynchronous and MAY be sent by either endpoint on any stream. Control-plane errors SHOULD be sent on Stream 0. Errors relating to rendering, input, or video fallback MAY be sent on the stream on which the offending message was received.

Receipt of an error does not terminate the SCTP association unless the error is marked fatal.

MWS_ERROR (type=600)
Reports a protocol, semantic, or transport-related error.

Fields:

```
uint32_t error_code;
uint32_t offending_type;   // MWS_* opcode that caused error
uint32_t window_id;        // 0 if not applicable
uint32_t fatal;            // 0 = recoverable, 1 = fatal
char     description[];    // UTF-8 diagnostic string
```

Error classes:

Protocol errors:

- unknown or unsupported opcode
- malformed header or payload
- invalid magic value
- message sent on the wrong SCTP stream
- invalid length field
- framing violations

Protocol errors are fatal unless explicitly stated otherwise or the endpoint can reliably discard the offending message without desynchronising protocol state.

Semantic errors:

- referencing a window outside the authenticated session
- referencing a destroyed or revoked resource
- message not valid in the current protocol state
- invalid seat_id or session_id

Semantic errors are recoverable unless otherwise stated.

Policy errors:

- compositor policy violation
- swapchain revoked due to timeout or resource pressure

Policy errors are recoverable unless the compositor explicitly marks them fatal.

Timeout errors:

- expected reply not received within implementation-defined limits
- client or server unresponsive

Timeout errors MAY be fatal depending on implementation policy.

Transport errors:
SCTP association loss
excessive retransmissions
PR-SCTP frame discard (non-fatal)
Transport errors reflect underlying SCTP conditions. Association loss is always fatal.

Endpoint failures:
server crash or restart
client crash or termination
Endpoint failures are fatal conditions and do not generate MWS_ERROR messages.

Recoverable errors (fatal=0) indicate that the offending message has been ignored and the session MAY continue. Fatal errors (fatal=1) indicate that the SCTP association MUST be closed immediately after transmitting the error, unless the error prevents the message from being parsed.

Loss of the SCTP association for any reason (network failure, timeout, endpoint crash) is treated as a fatal error. The session MAY persist according to the rules in Section 4.7.

All error_code and window_id fields follow the network-byte-order rules defined in Section 5.1.

5.8.1. Session and Resource Validation Errors

The server MUST validate that all window_id, seat_id, and session_id fields in client-originated messages refer to resources owned by the authenticated session associated with the SCTP association on which the message was received.

The following conditions constitute session and resource validation errors (a subclass of semantic errors) and MUST be reported using MWS_ERROR(type=600, fatal=0):

referencing a window belonging to another session
referencing a seat belonging to another session
attempting to bind a Vulkan surface to a window outside the authenticated session
specifying an sctp_stream_id that does not match the rendering stream allocated to the window
attempting to resume or manipulate a session not associated with the current SCTP association
providing a session_id that does not match the authenticated session

When reporting such errors, the server MUST NOT reveal the existence, geometry, focus state, or any other attributes of resources belonging to other sessions. The window_id field in the error message MUST be set to zero if revealing the true identifier would disclose cross-session state.

These rules ensure that clients cannot infer the presence of other users, windows, or seats, and that all resource identifiers remain strictly scoped to the authenticated session.

6. Reference Implementation

A reference implementation of the Mercurius Window System (MWS) is provided to validate the protocol, exercise compositor and rendering paths, and demonstrate interoperability between server and client components. The implementation is intentionally minimal: it confirms architectural correctness rather than

providing a productionready desktop environment.

The reference implementation reflects a common deployment model in which a single powerful workstation is accessed from multiple untrusted, mobile, or ephemeral terminal devices across a highbandwidth local network. MWS is designed for this pattern: the workstation is the authoritative environment, and terminal devices act solely as access points. The reference implementation validates this model and provides a foundation for further development.

6.1. Server Components (metamwsserver)

The metamwsserver package installs all components required to run an MWS server on a workstation. It is expected to be installed on the machine that owns GPU resources, sessions, and compositor state.

mwsd
: The primary MWS server daemon. Implements SCTP transport, DTLS mutual authentication, DANE validation of server and client certificates, PAM integration, Vulkan rendering, compositor policy, and validation layers. Owns all session, seat, and window state and enforces the security and isolation rules described in Sections 35.

mlogo
: A minimal Phase 1 test client that runs on the server and connects to the local mwsd instance. Used to validate transport, handshake, session establishment, and basic rendering on the workstation itself. Displays a static Mercurius logo in a window managed by the compositor and exits. This client is analogous to the historical xlogo utility used to verify X11 transport and rendering.

6.2. Client Components (metamwsclient)

The metamwsclient package installs all components required to run MWS terminals on untrusted or mobile devices. It is expected to be installed on laptops, thin clients, or other endpoints that attach to an MWS workstation over the network.

mwsc
: The primary client executable. Implements the clientside transport stack, handshake logic, window management, and event routing. Uses the shared protocol definitions described in Section5.

libmws.a
: Static client library providing SCTP/DTLS transport bindings, DANEvalidated mTLS authentication, AV1 decode (via dav1d) for Stream 3 fallback, and Vulkan loader integration. Intended for use by test clients and early adopters. The library is built as a single static archive to simplify deployment and avoid external runtime dependencies on terminal devices.

mws_protocol.h
: Public protocol header defining all message types, opcodes, structure layouts, and constants corresponding to the formats described in Section 5. This header is generated directly from the protocol specification and opcode registry (Appendix A) and is intended to remain stable across minor revisions.

6.3. Demonstration Clients

In addition to mlogo, the reference implementation includes (or

anticipates) progressively more complex demonstration clients. These clients are not part of the core protocol but are essential for validating compositor behaviour, swapchain management, input latency, and longrunning rendering workloads. Installation of these clients MAY be provided by either metamwsserver or metamwsclient depending on deployment preferences.

6.3.1. Screensaver Demonstration

A port or adaptation of a classic Unix screensaver (e.g. from the XScreenSaver collection) is provided as a midcomplexity test client. This client exercises continuous animation, swapchain reuse, timing stability, and fallback video paths. It validates that MWS can sustain longrunning rendering without resource leaks or timing drift and that video fallback on Stream 3 behaves correctly under varying bandwidth conditions.

6.3.2. Indie Game Demonstration

A small Vulkan or OpenGLbased indie game is used as a highcomplexity test client. This client exercises realworld interactive rendering workloads, including continuous animation, highfrequency input events, window focus changes, and typical GPU command patterns found in games. It validates that MWS can host practical latencysensitive graphical applications while maintaining low input latency, stable frame pacing, and correct swapchain behaviour. This demonstrates that MWS supports realistic workloads beyond synthetic benchmarks.

6.4. Dependencies

The reference implementation targets a contemporary Linux environment. The following libraries are required by the provided server and client components:

libsctp1
: SCTP stack (Linux 6.8 or later recommended).

vulkantools
: Vulkan loader and validation layers.

libdav1d
: AV1 decoder used for Stream 3 fallback.

libpam0g
: PAM authentication backend.

libtevent
: Asynchronous event loop library.

6.5. Bootstrap Example

This example illustrates a minimal twohost deployment in which a workstation (xavier) runs the MWS server and application processes, and a remote terminal (flash) provides the users display and input. All connections use the default SCTP transport.

For this example there are two hosts:

 xavier: a workstation acting as the server
 flash: a thin client

1. Start the MWS server on xavier:

```
xavier$ mwsd --port 49152
```

This starts the MWS server daemon mwsd on xavier, listening for DTLS/SCTP associations on port 49152 and exposing the compositor and session manager to remote terminals.

2. Start the terminal on flash and attach to xavier:

```
flash$ mwsc --server xavier --port 49152
```

This starts the MWS terminal client on flash. It connects to xavier:49152 over DTLS/SCTP, performs mutual certificate authentication (optionally validated via DANE), completes user authentication, and establishes a session and seat bound to the display and input devices on flash.

3. Run the test application on xavier:

```
xavier$ mlogo
```

The mlogo client connects to the local mwsd instance on xavier, joins the existing session associated with the seat on flash, creates a window, and maps it. The Mercurius logo appears on the display attached to flash while the application process executes on xavier.

This bootstrap validates the endtoend path between a workstation and a remote terminal: DTLS/SCTP transport establishment, DANEvalidated certificate authentication, user authentication, session and seat creation, window creation and mapping, swapchain initialisation, and basic GPU rendering.

7. Implementation Requirements and Validation

This section defines normative requirements for any conformant MWS implementation. These requirements ensure correct behaviour under load, predictable session semantics, and robust isolation between clients. The reference implementation demonstrates these properties but does not attempt to optimise for all hardware configurations.

7.1. Test Matrix

An implementation of MWS MUST demonstrate correct behaviour across four major dimensions:

Session semantics — creation, resume, detachment, identifier stability, and state continuity.

Window lifecycle — creation, mapping, resizing, destruction, and identifier scoping.

Rendering correctness — surface creation, command ordering, GPU isolation, and frame delivery.

Transport behaviour — SCTP stream allocation, ordering guarantees, error handling, and reconnection.

The following matrix defines the minimum set of tests required to validate interoperability between an MWS client and server. These tests are not exhaustive; they represent the baseline necessary to confirm that the architectural components described in this document behave as specified.

7.1.1. Core Validation Tests

Test Case	Description	Success Criteria
Bootstrap + Render	Establish a session and render a minimal surface using the reference client.	Initial frame displayed in a reasonable time on reference hardware; session terminates or detaches cleanly.
Window Lifecycle	Create, map, unmap, and destroy a window while observing compositor events.	Correct CREATE→MAP→UNMAP→DESTROY sequence; no orphaned resources.
Session Persistence	Start a session, detach or allow the client to disconnect, then resume using the same session identifier.	Session resumes with compositor state reconstructed as defined in Sections 4.7 and 5.2.2; client can redraw without protocol violations.
GPU Isolation	Run multiple clients concurrently, each creating independent surfaces.	No crosssession resource leakage; surfaces and windows remain isolated.
Transport Stream Allocation	Exercise streams 03 with mixed control, rendering, input, and video fallback traffic.	No reordering within a stream; correct routing based on session and window IDs.
Input Event Semantics	Deliver pointer and keyboard events to multiple windows across seats.	Events delivered only to the focused window; correct seat and session scoping.
Error Handling	Trigger invalid IDs, malformed messages, and protocol violations.	Server returns appropriate error codes (600699) as defined in Section 5.8; session integrity maintained.

7.1.2. Reference Implementation Commands (NonNormative)

The reference implementation provides convenience tools for exercising the above tests. The following examples illustrate the intended usage pattern; conforming implementations MAY use any equivalent mechanism.

For this example there are two hosts:

xavier: a workstation acting as the server
flash: a thin client

Bootstrap + Render:

```
xavier$ mwsd --port 49152
flash$  mwsc --server xavier --port 49152
xavier$ mlogo
```

Expected result: a logo window appears on the display attached to flash; mlogo exits with status 0.

Window Lifecycle:

```
xavier$ mlogo --once
```

Expected result: the server log shows a CREATE→MAP→DESTROY sequence for the test window; no orphaned compositor or GPU resources remain.

Session Persistence:

```
flash$  mwsc --server xavier --port 49152
xavier$ mlogo
[disconnect terminal or detach session, then resume]
```

Expected result: the session resumes with windows and compositor state reconstructed as defined in Sections 4.7 and 5.2.2; the client can reestablish rendering without violating protocol or WSI rules.

GPU Isolation:

```
xavier$ mlogo &
xavier$ mlogo &
```

Expected result: independent windows are created in separate sessions without visible interference; destroying one client does not affect the others.

These examples are illustrative only. They do not form part of the normative protocol and do not constrain implementationspecific tooling.

7.2. GPU Isolation Requirements

Implementations MUST ensure that GPU workloads from one session cannot compromise the integrity or confidentiality of another session’ s resources.

The server SHOULD avoid allowing GPU workloads from one session to starve or block those of another. Implementations MAY use separate Vulkan queues, queue subsets, or fair scheduling mechanisms to achieve this.

The server MUST validate VkCommandBuffer submissions sufficiently to prevent malformed or outofbounds accesses that would violate isolation guarantees. Invalid or malformed command buffers MUST be rejected without execution.

The server MUST enforce persession limits on GPU resource usage, including device memory, descriptor sets, and command buffer size. When limits are exceeded, the server MAY throttle, reject further submissions, or terminate the session.

The server SHOULD implement watchdog mechanisms to detect and recover from GPU hangs attributable to a particular session. Recovery MAY include resetting client queues, revoking swapchains, or terminating the offending session while preserving other sessions.

7.3. Bandwidth and Transport Isolation Requirements

Implementations MUST ensure that control and input remain responsive under load and that one client cannot monopolise transport resources to the detriment of others.

Stream 0 (control) and Stream 2 (input) MUST use reliable, ordered delivery and MUST be prioritised over bulk data on other streams.

Stream 1 (rendering commands) MUST use reliable, ordered delivery. The server MAY impose rate limits on VK_SUBMIT or equivalent rendering messages to prevent excessive queueing.

Stream 3 (AV1 fallback video) MAY use partially reliable delivery (PRSCTP). The server MAY drop frames under congestion to maintain interactivity.

The server SHOULD implement persession or perclient bandwidth limits to prevent link saturation. Limits MAY be enforced at the SCTP layer, via traffic shaping, or using equivalent mechanisms.

The server MUST be able to unilaterally terminate a misbehaving client without impacting other sessions. Termination SHOULD be signalled with MWS_ERROR(type=600, fatal=1) followed by closure of the SCTP association, as defined in Section 5.8. Termination MUST release all GPU, transport, and compositor resources associated with that client, and MUST NOT affect other active sessions.

8. Performance Considerations

MWS is designed for environments where the workstation and client devices are connected by modern highbandwidth, lowlatency networks. Contemporary 10 GbE and fibrebacked LANs routinely deliver roundtrip latencies on the order of 200500 s; in such environments the network contributes only a small fraction of the endtoend inputtodisplay path. A typical RTT in this range implies oneway transport latency well below 0.5 ms, meaning that interactive responsiveness is dominated by GPU encode and decode rather than by the network.

Modern GPUs provide hardwareaccelerated AV1 encoding with submillisecond latency for common workstation resolutions. Because MWS performs all rendering and compositing on the server, and because clients do not execute application code or GPU workloads, the clientside overhead is minimal. This eliminates the multiple GPU roundtrips, buffer copies, and synchronisation barriers found in traditional remotedesktop systems, where the client must composite, scale, or colourconvert frames before display. By reducing the client to a lightweight presentation endpoint with minimal CPU and GPU involvement, MWS ensures that interactive latency is dominated by serverside encode time rather than by clientside processing. As a result, a remote graphical session over a modern LAN is effectively indistinguishable from local use for typical workstation workloads.

The following nonnormative figures illustrate achievable encodeandtransport performance on a 10 GbE LAN using an RTX 5070class GPU. These values represent serverside encode latency plus network transit time under ideal conditions; they do not include clientside display latency.

Resolution	Bitrate	Encode+Transport Latency	CPU
4K60 HDR	200 Mbps	<0.5 ms	<1%
4K120 Gaming	500 Mbps	<1.0 ms	<2%
8K60 HDR	1.2 Gbps	<1.5 ms	<3%

These figures demonstrate that MWS can support highresolution, highrefreshrate graphical workloads over a modern LAN without compromising interactive performance. For typical desktop and workstation applications, the resulting experience is comparable to local use.

9. Security Considerations

MWS is designed according to zerotrust principles: no client device, network segment, or intermediary is implicitly trusted. All trust is derived from cryptographic identity and explicit authorisation rather than network location. The protocol assumes that client devices may be compromised, mobile, or operating on hostile networks, and that attackers may observe, inject, or replay traffic unless prevented by cryptographic protections.

Transport security is provided by a secure datagram transport with mutual authentication (for example, DTLS over SCTP). When DANE is deployed, the client and server certificates are validated against DNSSECprotected TLSA records, ensuring that only devices explicitly provisioned by the domain administrator may initiate a connection. Deployments without DNSSEC or without control over their DNS zone SHOULD use traditional PKI validation instead.

User authentication is performed at the application layer using the mechanismagnostic model defined in Section 5.2.1. The server advertises supported mechanisms (e.g., "DANE", "PAM", "PASSWORD", "FIDO2") as UTF8 identifiers in MWS_AUTH_CHALLENGE, and the client selects one. This separation of device and user identity ensures that compromise of a device does not grant access to a user’s session without the corresponding user credential.

Each user is given an isolated session and compositor context. Clients cannot observe or interfere with other users’ windows, input events, or rendering state. Window identifiers are scoped to a session, and all clientoriginated messages are validated by the server. Attempts to reference resources outside the authenticated session are rejected with MWS_ERROR(type=600) as described in Section 5.8.

Rendering commands (300399), input events (400499), and video frames (500599) are isolated on separate SCTP streams to limit the impact of congestion or packet loss. Partially reliable delivery MAY be used for video fallback to avoid resource exhaustion and to prevent attackers from inducing excessive retransmissions without affecting control or input flows.

The client is not part of the trusted computing base. It stores no confidential data, longterm session state, or reusable credentials. If a client disconnects unexpectedly, the session persists only for the duration of the reconnection grace period unless the user has explicitly detached. After this period, the session is terminated and all associated resources are destroyed, as defined in Section 4.7.

Loss of the transport association for any reason (network failure, timeout, endpoint crash) is treated as a fatal transport error. The session MAY persist according to the rules in Section 4.7 and MAY be resumed from another client device subject to policy.

9.1. DANE Deployment (NonNormative)

Deployments that operate their own DNS infrastructure MAY use DNSSEC and TLSA records (DANE) to authenticate client and server certificates during the DTLS/SCTP handshake. When DNSSEC validation is available, DANE provides a robust mechanism for binding workstation and device identity to DNS without relying on public certificate authorities.

When the "DANE" authentication mechanism is selected, the client proves possession of a private key whose certificate is published via a DNSSECprotected TLSA record. No credential payload is required in MWS_AUTH_RESPONSE (type=003); all authentication material is derived from the mutualTLS handshake and DNSSEC validation. This allows deployments to authenticate devices without storing reusable secrets on the client.

DANE is OPTIONAL and does not alter the protocol semantics. When enabled, it reduces operational complexity in closed trust domains by eliminating external trust dependencies, simplifying certificate lifecycle management, and mitigating maninthemiddle attacks even in the event of public CA compromise.

Deployments without DNSSEC or without administrative control over their DNS zone SHOULD use traditional PKI validation instead.

10. IANA Considerations

This document requests two IANA actions:

1. Allocation of an SCTP port number for the Mercurius Window System (MWS). The suggested port is 49152.

2. Registration of the applicationlayer protocol identifier (ALPN) "mws".

No other registries are required. In particular, MWS message types, opcodes, and SCTP stream assignments are managed entirely within the protocol and do not require IANA allocation.

11. Acknowledgements

Christopher Ross (chris@tebibyte.org) provided the initial design and reference implementation.

The reference implementation described in Section 6 is available from the Mercurius project website at

<https://mercurius.tebibyte.org>.

The corresponding source code repository may be cloned via <https://mercurius.tebibyte.org/git/mercurius.git>.

SSH access is also available for contributors. These resources are provided for convenience and are nonnormative; the protocol defined in this document does not depend upon any specific implementation.

12. References

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

[RFC4895] Tuexen, M., Stewart, R., and P. Lei, "Authenticated Chunks for Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007.

[RFC9260] Stewart, R., Tuexen, M., and X. Dutreilh, "Stream Control Transmission Protocol", RFC 9260, DOI 10.17487/RFC9260, June 2022.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018.

12.2. Informative References

[VK14] Khronos Group, "Vulkan 1.4 Specification", 2024.

[NIST800-207] National Institute of Standards and Technology, "Zero Trust Architecture", NIST Special Publication 800-207, August 2020.

[RFC9261] Tuexen, M. and R. Stewart, "Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets", RFC 9261, DOI 10.17487/RFC9261, June 2022.

12.2. Informative References

[VK14] Khronos Group, "Vulkan 1.4 Specification", 2024.

[NIST800207] National Institute of Standards and Technology, "Zero Trust Architecture", NIST SP 800207, 2020.

Appendix A. MWS Opcode Registry

This appendix defines the complete registry of MWS opcodes. All opcodes are 16bit unsigned integers. Opcodes are grouped into 100entry ranges according to functional category. Implementations MUST treat unknown opcodes as protocol errors and respond with MWS_ERROR(type=600) as described in Section 5.8.

A.1. Handshake and Authentication (000099)

NOTE: Type 000 is reserved and MUST be treated as a NULL/invalid value. Implementations encountering type=000 MUST respond with MWS_ERROR.

001 MWS_QUERY
002 MWS_AUTH_CHALLENGE
003 MWS_AUTH_RESPONSE
004 MWS_SURFACE_CAPS
005 MWS_SESSION_INFO

050099 Reserved for future handshake extensions

A.2. Session Management (100199)

100 MWS_SESSION_RESUME_OFFER
101 MWS_SESSION_RESUME_REQUEST
102 MWS_SESSION_RESUME_COMPLETE

150199 Reserved for future sessionmanagement extensions

A.3. Window Lifecycle (200299)

200 MWS_CREATE_WINDOW
201 MWS_WINDOW_CREATED
202 MWS_DESTROY_WINDOW
203 MWS_WINDOW_DESTROYED
204 MWS_MAP_WINDOW
205 MWS_UNMAP_WINDOW
206 MWS_CONFIGURE_WINDOW
207 MWS_FOCUS_WINDOW
208 MWS_SWAPCHAIN_REVOKED

250299 Reserved for future windowlifecycle extensions

A.4. Rendering Commands (300399)

300 MWS_VK_SUBMIT
301 MWS_VK_SYNC
302 MWS_VK_DESTROY

350399 Reserved for future rendering extensions
(e.g., bitmap upload, GPUside composition, etc.)

A.5. Input Events (400499)

400 MWS_INPUT_EVENT
401 MWS_INPUT_ACK

450499 Reserved for future input extensions
(e.g., haptics, multiseat extensions)

A.6. Video Fallback (500599)

500 MWS_AV1_FRAME
501 MWS_PLACEHOLDER_FRAME

550599 Reserved for future videofallback extensions

A.7. Error Reporting (600699)

600 MWS_ERROR

650699 Reserved for future errorreporting extensions

A.8. Capability and Introspection (700799)

700799 Reserved for capability discovery, introspection, and future protocolnegotiation mechanisms.

A.9. Reserved for Future Extensions (800899)

800899 Reserved for future protocol extensions.

A.10. Experimental and VendorSpecific (900999)

900999 Experimental, vendorspecific, or implementationdefined opcodes. These MUST NOT be used in interoperable deployments and MUST NOT be relied upon in Internetscale deployments.

Appendix B. Authentication Mechanism Registry

MWS supports a mechanismagnostic authentication model. During the initial handshake, the server advertises one or more authentication mechanisms using MWS_AUTH_CHALLENGE (type=002). The client selects a mechanism and responds with MWS_AUTH_RESPONSE (type=003), providing mechanismspecific credentials or authentication data.

This appendix defines the registry of authentication mechanism identifiers. Mechanism identifiers are UTF8 strings and are compared using casesensitive bytewise comparison. Identifiers MUST NOT exceed 64 bytes in length.

Implementations MUST ignore unknown mechanism identifiers and MUST NOT attempt to interpret their payloads. Servers MUST NOT advertise mechanisms they do not fully support.

B.1. Standard Mechanisms

The following mechanism identifiers are defined by this specification:

"PAM"
The server authenticates the user using the system’s Pluggable Authentication Modules (PAM) stack. The credential payload contains a NULterminated username followed by a NULterminated password.

"PASSWORD"
A simple username/password mechanism. The credential payload contains a NULterminated username followed by a NULterminated password. This mechanism is intended for deployments that do not use PAM but still require passwordbased authentication.

"DANE"
Authentication using DNSSECvalidated TLSA records. The client proves possession of a private key whose certificate is bound to a DNS name via a TLSA record. When this mechanism is selected, the credential payload is empty; all authentication material is derived from the mTLS handshake and DNSSEC validation. This mechanism is intended for closed trust domains that operate their own DNSSEC infrastructure.

B.2. Extensible Mechanisms

The following identifiers are reserved for future specifications or external standards. Their payload formats are not defined by this document.

"FIDO2"
Authentication using a FIDO2 authenticator.

"WEBAUTHN"
Authentication using a WebAuthn ceremony.

"KERBEROS"
Authentication using a Kerberos APREQ exchange.

"OAUTH2"
Authentication using an OAuth 2.0 device or authorizationcode flow.

Servers MAY advertise any subset of these mechanisms. Clients MAY implement any subset.

B.3. Private and Experimental Mechanisms

Mechanism identifiers beginning with the prefix "X" are reserved for private, experimental, or vendorspecific use. These identifiers MUST NOT appear in interoperable deployments or Internetfacing services.

Examples:
"XFINGERPRINT"
"XHARDWARETOKEN"
"XSSOPROTOTYPE"

B.4. Registration Policy

New mechanism identifiers MAY be defined by future MWS extensions or external standards. To avoid collisions, new identifiers SHOULD be registered with IANA if this specification is published on the IETF Standards Track.

Until such time, implementers SHOULD use the "X" prefix for experimental mechanisms and MUST NOT assume global uniqueness.

Appendix C. SCTP Stream Usage Summary

MWS uses multiple SCTP streams to isolate control, rendering, input, and video traffic. This appendix summarises the required stream assignments. All streams use DTLS for confidentiality and integrity.

Stream assignments are fixed and MUST NOT be repurposed for other message classes. Implementations MAY open additional streams for experimental or vendorspecific extensions, provided they do not conflict with the assignments below.

C.1. Stream 0 — Control Plane

Stream 0 carries all ordered controlplane traffic, including:

handshake messages (001099)
sessionmanagement messages (100199)
windowlifecycle messages (200299)
error messages (600699)

Messages on Stream 0 MUST be delivered reliably and in order.

C.2. Stream 1 — Rendering Commands

Stream 1 carries rendering commands (300399), including Vulkan command submission and GPUresource management.

Messages on Stream 1 MUST be delivered reliably and in order.

C.3. Stream 2 — Input Events

Stream 2 carries input events (400499), including keyboard, pointer, and tablet events.

Messages on Stream 2 SHOULD be delivered reliably but MAY be processed out of order by the server if permitted by the input subsystem.

C.4. Stream 3 — Video Fallback

Stream 3 carries videofallback traffic (500599), including AV1 frames and placeholder frames.

Stream 3 MUST use PRSCTP (Partial Reliability SCTP) to allow frame discard under congestion. Implementations SHOULD use timed reliability or limited retransmission to avoid headofline blocking.

C.5. Additional Streams

Streams beyond Stream 3 are reserved for future extensions. Such extensions MUST specify:

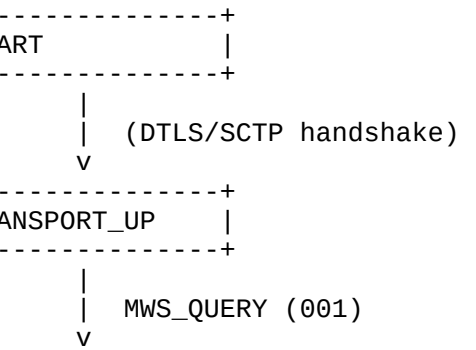
- reliability requirements (reliable, PRSCTP, unordered)
- congestioncontrol expectations
- interaction with the control plane

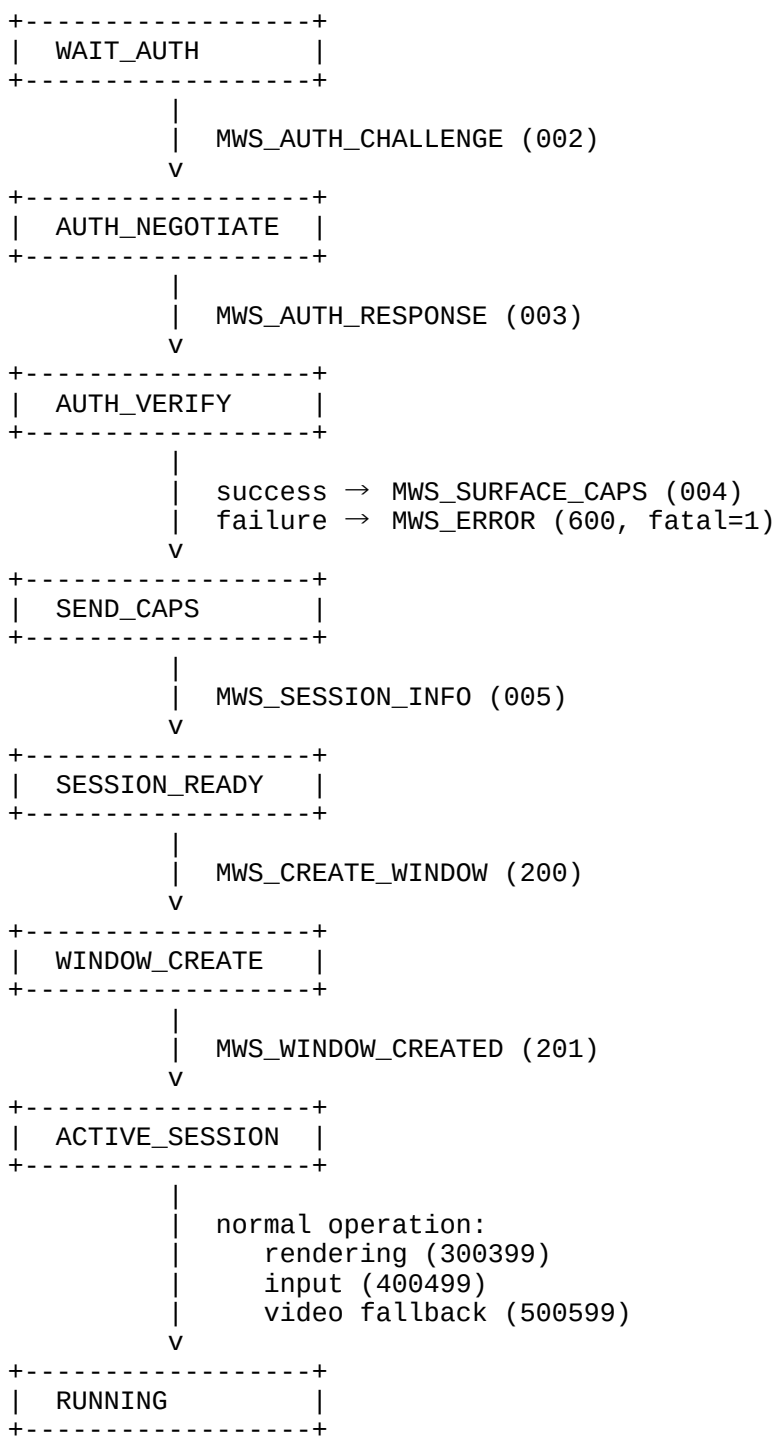
Experimental or vendorspecific extensions SHOULD use streams 16 to avoid collision with future standardised assignments.

Appendix D. Protocol State Machine Diagrams

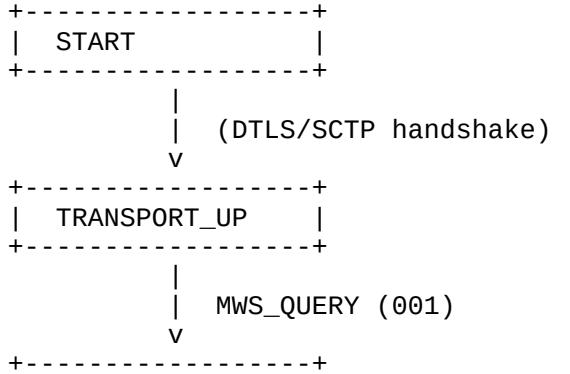
This appendix provides normative statemachine diagrams for the MWS protocol. These diagrams illustrate the ordered interactions between client and server during initial connection, session resumption, and normal operation. All controlplane transitions occur on SCTP Stream 0. Rendering, input, and video traffic occur on their respective streams as defined in Appendix C.

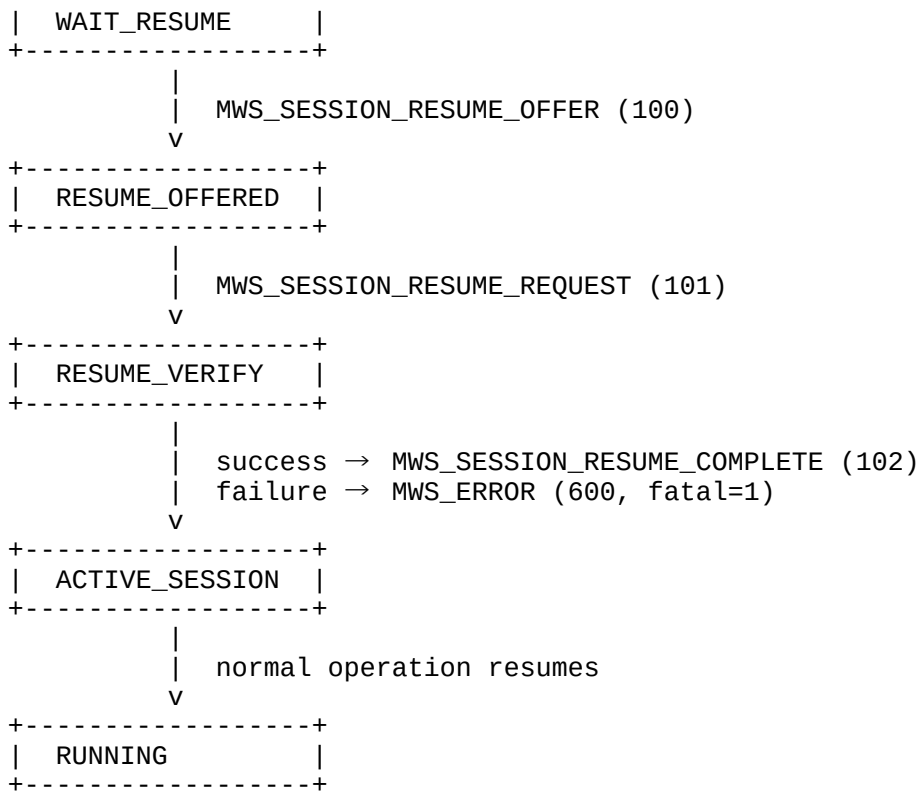
D.1. Initial Connection State Machine





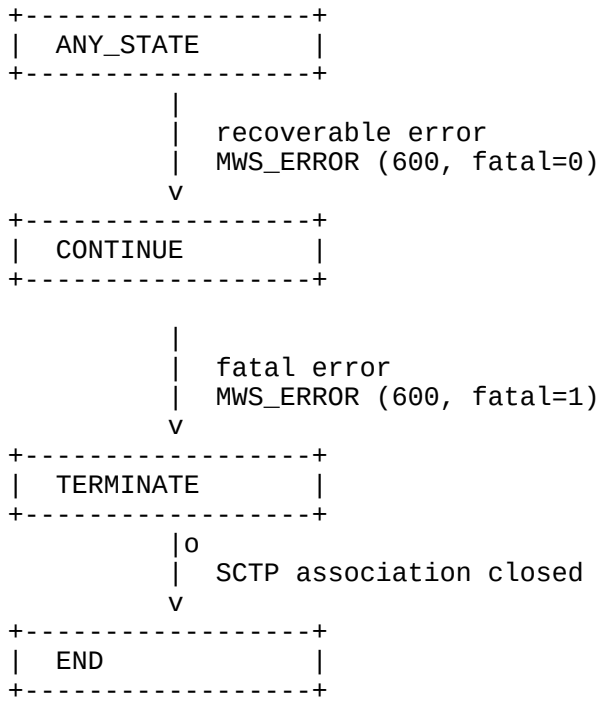
D.2. Session Resume State Machine





D.3. Error Handling State Machine

Errors may occur at any point in the protocol. The following diagram illustrates the errorhandling model:



D.4. Stream Interaction Summary

The following diagram illustrates the concurrency model across SCTP streams:

Stream 0 (control): ordered, reliable

```
        handshake (001099)
        session management (100199)
        window lifecycle (200299)
        errors (600699)

    Stream 1 (rendering): ordered, reliable
        Vulkan commands (300399)

    Stream 2 (input):     reliable; MAY be processed out of order
        input events (400499)

    Stream 3 (video):     PRSCTP, typically unordered
        AV1 frames (500599)
```

These streams operate independently. Loss or delay on one stream MUST NOT block progress on any other stream.

13. Copyright