MXL:

Media eXchange Layer

How shared-memory ring buffers are replacing network packets for software-defined broadcast media exchange

By: Michael Demb

April 2026

Based on the open-source SDK at github.com/dmf-mxl/mxl
and the fabrics proxy at github.com/jonasohland/mxl-fabrics-proxy

What is MXL?

MXL (Media eXchange Layer) is an open-source SDK designed for high-performance, container-friendly shared media exchange across on-premise or cloud environments. It serves as the data-plane component of the EBU's Dynamic Media Facility (DMF) reference architecture, which replaces dedicated hardware appliances with orchestrated software microservices.

The core idea is straightforward: instead of packetizing video into network packets (as ST 2110 does), applications running as containers on shared infrastructure write media frames into shared memory, and other applications read from that same memory. No network stack, no kernel transit, no copies.

The SDK is implemented in C++ with a C API and Rust bindings. Version 1.0 was released in February 2026, hosted under the Linux Foundation with governance from the EBU and NABA.

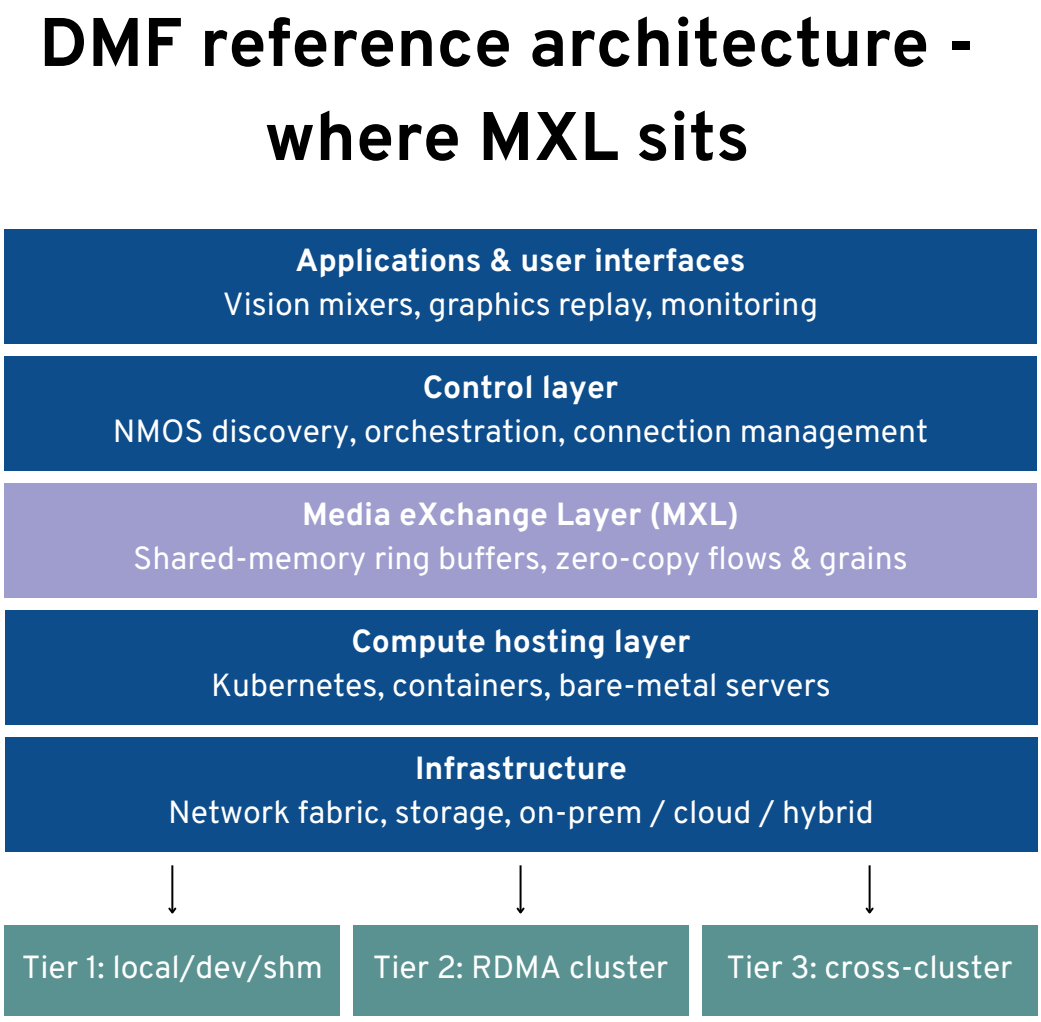


Figure 1: DMP reference architecture - where MXL sits in the stack

Key characteristics

- Implements zero-copy media sharing via POSIX shared memory (tmpfs / /dev/shm)
- Supports V210 video, Float32 audio, V210+Alpha, and SMPTE 291 ancillary data
- Asynchronous by design: enables faster-than-real-time workflows
- Thin library with minimal external dependencies, not a framework
- Deployable on local or cloud compute, including Docker and Kubernetes
- Uses UNIX file permissions for security at domain and flow level

The Problem MXL Solves

The broadcast industry is transitioning from SDI hardware to IP-based software workflows. SMPTE ST 2110 was the first major standard for IP video transport, but it was designed for network transport between hardware appliances. When everything moves into software containers on the same server, 2110 introduces unnecessary overhead.

Under ST 2110, each software application must packetize video into RTP/UDP, hand it to the OS kernel, send it over the NIC (even if the destination is on the same machine), receive it, reassemble, and buffer. MXL eliminates all of that when applications share a host: they just read and write the same memory region.

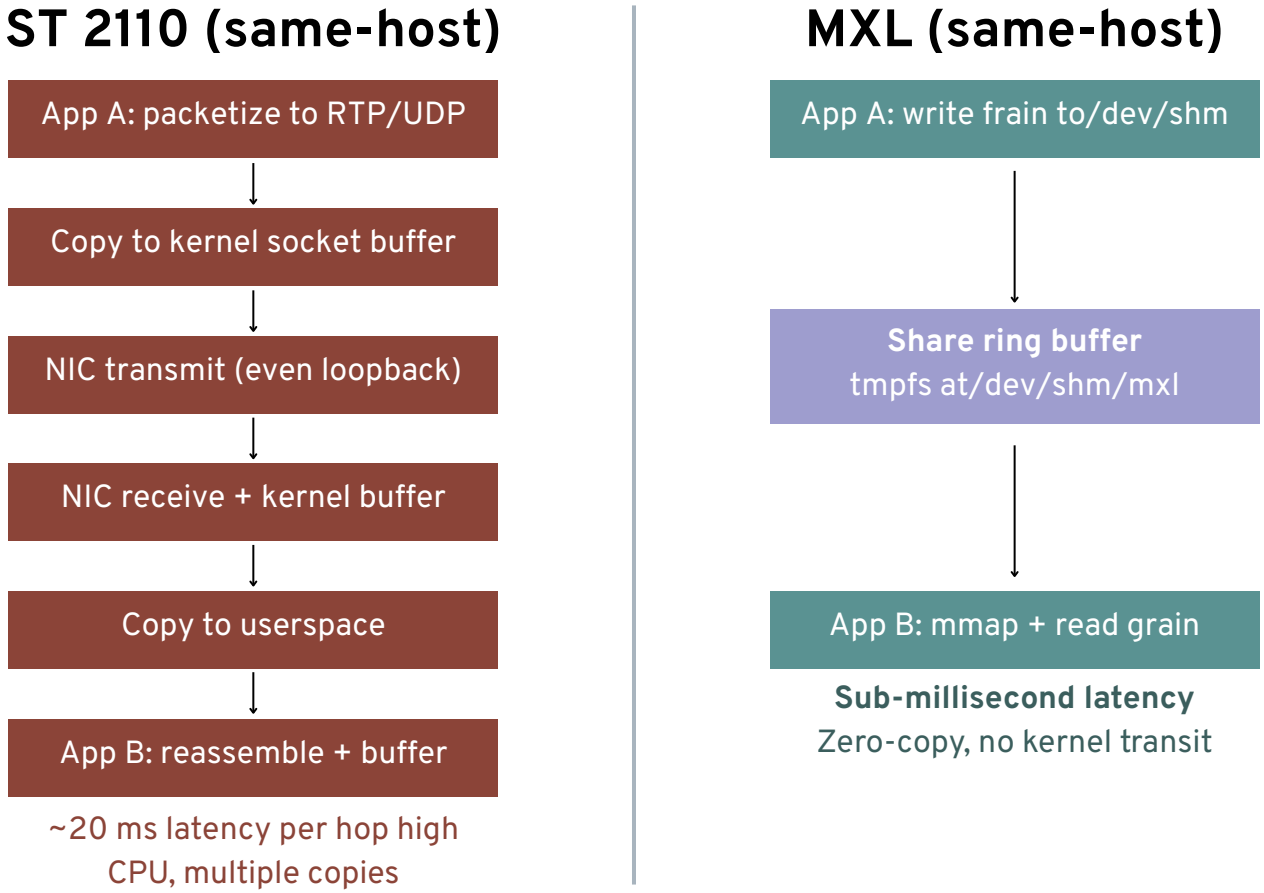


Figure 2: Data path comparison: ST 2110 vs MXL on the same host

ST 2110 data path (same-host)

Step	Operation
1	App A packetizes video into RTP/UDP packets
2	Data copied from userspace to kernel socket buffer
3	Kernel hands data to NIC for transmission (even loopback)
4	NIC receive path: packets arrive at kernel buffer
5	Data copied from kernel back to userspace
6	App B reassembles packets and buffers the frame

Result: ~20 ms latency per hop, high CPU overhead, multiple memory copies.

MXL data path (same-host)

Step 1: Application A writes grain directly into mmap'd shared memory (/dev/shm).

Step 2: Application B reads grain from the same mmap'd region (zero copy).

Result: Sub-millisecond latency, zero copies, no kernel transit, minimal CPU overhead.

MXL vs ST 2110: Complementary, Not Competing

MXL does not replace SMPTE ST 2110. The division of roles places ST 2110 at the network periphery for ingest and output, while MXL handles internal media exchange within compute clusters. They coexist: an ST 2110 receiver ingests video and writes it into an MXL ring buffer, then downstream applications read from that buffer with near-zero latency.

Characteristic	SMPTE ST 2110	MXL
Transport layer	RTP/UDP over IP network	POSIX shared memory (mmap)
Latency (per hop)	~20 ms per device	< 1 ms per transfer
Data copies	Multiple (user→ kernel→ NIC→ kernel →user)	Zero (same mmap region)
Time sync	PTP (SMPTE 2059) required	NTP sufficient (single host)
Infrastructure	PTP-aware switches, multicast VLANs	Standard Linux + tmpfs
Scope	Network transport (inter-device)	Memory exchange (intra-cluster)
Multi-host	Native (IP network)	Via Fabrics API (RDMA/RoCEv2)
Media format	RTP-packetized essence	Native uncompressed in memory
Interop model	Sender/receiver (active push)	Reader/writer (passive access)

Core Data Model: Domains, Flows, and Grains

MXL borrows terminology from the AMWA NMOS IS-04 specification. The key concepts, as implemented in the SDK source code:

MXL domain

A directory on a tmpfs-backed filesystem (typically `/dev/shm/mxl`) that serves as the namespace for all media flows on that host:

```
mxlInstance instance = mxlCreateInstance("/dev/shm/mxl", NULL);
```

Multiple domains can coexist on the same host, providing isolation between different production workflows.

Flow

A unidirectional stream of media, identified by a UUID. Each flow is a directory under the domain:

```
${mxlDomain}/
  ${flowId}.mxl-flow/
    data          ← Flow header (2048 bytes, mmap'd)
    flow_def.json     ← NMOS IS-04 flow resource
    access         ← Touch file for read tracking
    grains/
      0, 1, 2, ... N-1 ← Grain header + payload (mmap'd)
```

Grain

An individual unit of media data within a flow. For video, one grain equals one frame. For ancillary data (SMPTE 291), grains arrive at irregular intervals. Audio uses a different continuous ring buffer model.

Two ring buffer types

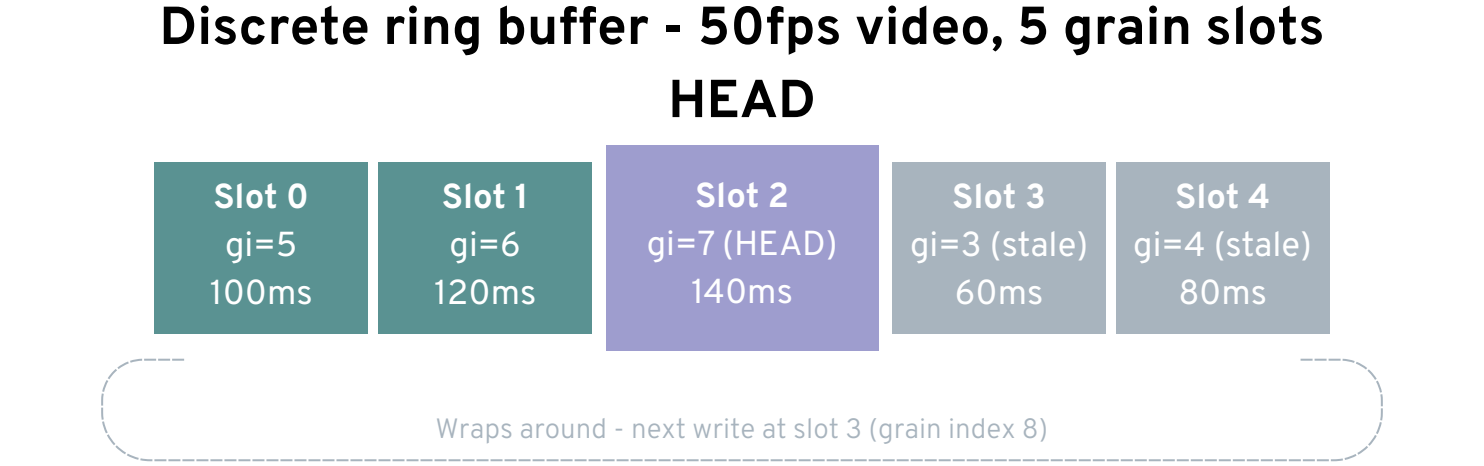
Discrete ring buffers: Used for video (video/v210, video/v210a) and ancillary data. Each grain occupies a fixed slot. The `mxlDiscreteFlowConfigInfo` struct specifies grainCount and sliceSizes per plane.

Continuous ring buffers: Used for audio (audio/float32). A single channels blob in shared memory laid out as channelCount independent circular buffers of bufferLength samples each.

Timing Model

MXL's timing model is built around the SMPTE 2059-1 epoch (TAI time). Every grain index maps directly to a nanosecond timestamp:

```
GrainDurationNs = (GrainRateDenominator * 1,000,000,000) /
GrainRateNumerator
GrainIndex    = Timestamp / GrainDurationNs
RingBufferSlot = GrainIndex % grainCount
```



Timing: grain index to timestamp

GraindDurationNs = (denom x 1e9) / numer GrainIndex = Timestamp / GrainDurationNs	RingSlot = GrainIndex % grainCount e.g. gi=7 → 7 % 5 = slot 2

Figure 3: Discrete ring buffer mechanics - 50fps video, 5 grain slots

Example indexing (50fps video, 5 grain ring buffer)

Grain Index	TAI Timestamp (ns)	Ring Slot
0	0	0
1	20,000,000	1
2	40,000,000	2
3	60,000,000	3
4	80,000,000	4
5	100,000,000	0
6	120,000,000	1
7	140,000,000	2

Grain indices are globally meaningful, no negotiation of sequence numbers is needed. MXL does not require PTP; it only uses the SMPTE 2059 epoch as a reference. NTP is sufficient for single-host deployments.

Flow synchronization groups

For multi-flow synchronization, the SDK provides `mxlFlowSynchronizationGroup`. Readers can be grouped and a single `waitForDataAt(timestamp, timeoutNs)` call blocks until all flows have data at the requested timestamp.

The Reader/Writer Model

MXL uses a reader/writer model rather than a sender/receiver model. The writer places data in shared memory and readers independently access it. There is no active sending, no connection establishment, and no data copies.

Writer side (producing media)

```
// 1. Create flow writer from NMOS flow definition
mxlCreateFlowWriter(instance, flowDef, NULL, &writer, NULL, &created);

// 2. Open grain slot (returns pointer into mmap'd shared memory)
mxlFlowWriterOpenGrain(writer, index, &grainInfo, &buffer);

// 3. Fill buffer with video/audio data (in place, no copy)
memcpy(buffer, sourceFrame, grainInfo.grainSize);

// 4. Signal readers via futex
mxlFlowWriterCommitGrain(writer, &grainInfo);
```

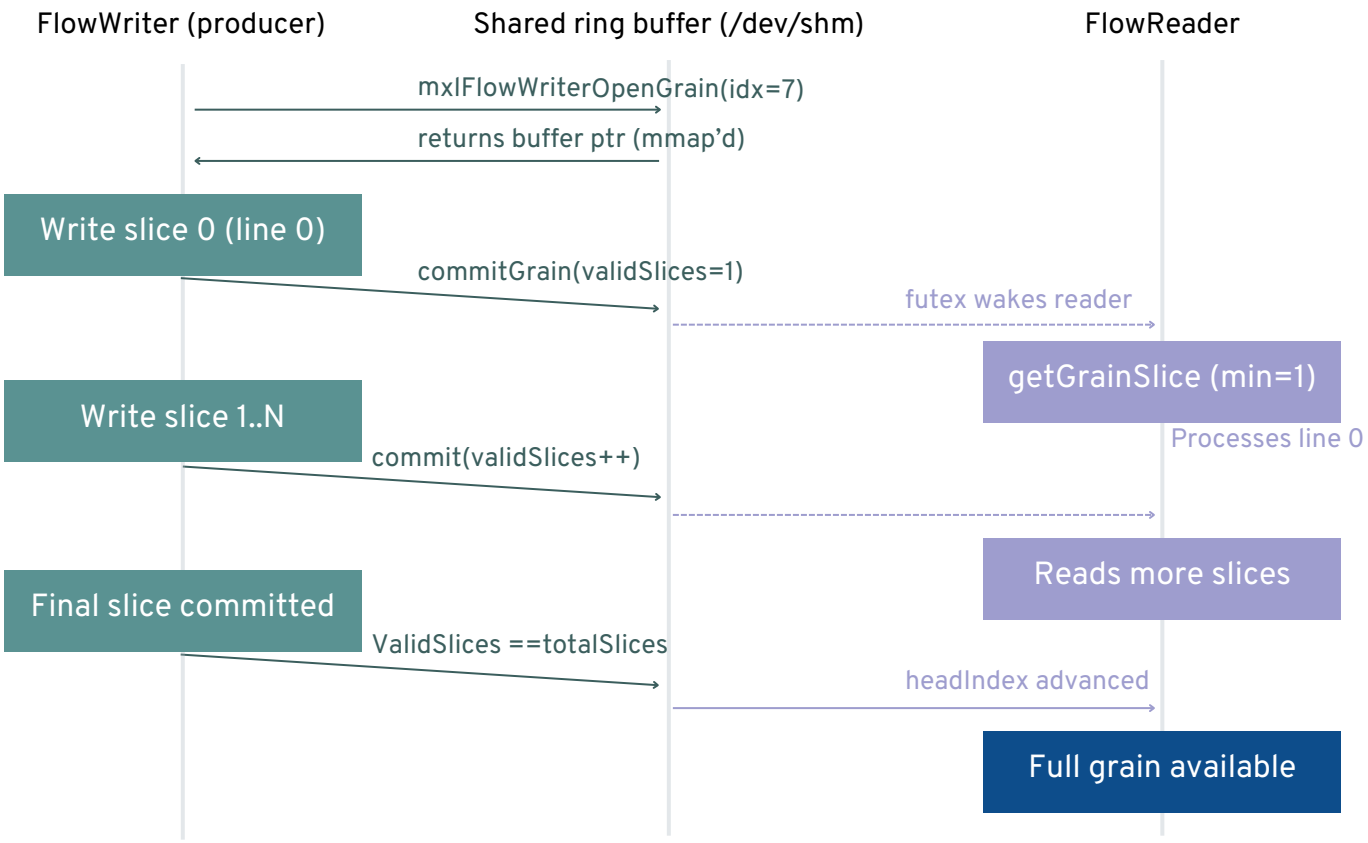
Reader side (consuming media)

```
// 1. Open existing flow for reading
mxlCreateFlowReader(instance, flowId, NULL, &reader);

// 2. Block until grain is available (zero-copy pointer returned)
mxlFlowReaderGetGrain(reader, index, timeoutNs, &grainInfo, &payload);

// 3. Process data in place, then advance to next index
```

Partial grain I/O (slice-by-slice)

Grains can be written and consumed incrementally. For video, individual scan lines (slices) can be committed progressively. The `validSlices` field tracks progress. A reader calling `mxlFlowReaderGetGrainSlice()`can start processing the top of a frame before the bottom has arrived.

Discrete grain I/O - partial (slice-by-slice) write



No copies at any stage - writer and reader both use mmap'd pointers into /dev/shm

Figure 4: Discreter grain I/O - partial (slice-by-slice) write sequence

Synchronization mechanism

Synchronization uses Linux futexes rather than POSIX mutexes. Readers only mmap in read-only mode (PROT_READ), so POSIX mutexes (which need write access) cannot be used. Advisory file locks (flock) detect stale flows from crashed processes via `mxlGarbageCollectFlows()`.

Supported Media Formats

Format ID	Type	Description
video/v210	Video	10-bit 4:2:2 uncompressed (broadcast-grade SDI quality)
video/v210a	Video + Alpha	V210 fill + alpha key in single grain (graphics/keying)
audio/float32	Audio	IEEE 754 32-bit float, range [-1.0, +1.0], continuous ring buffer
video/smpte291	Ancillary	SMPTE 291 packets (captions, timecodes, AFD) per RFC 8331

Continuous audio buffer layout

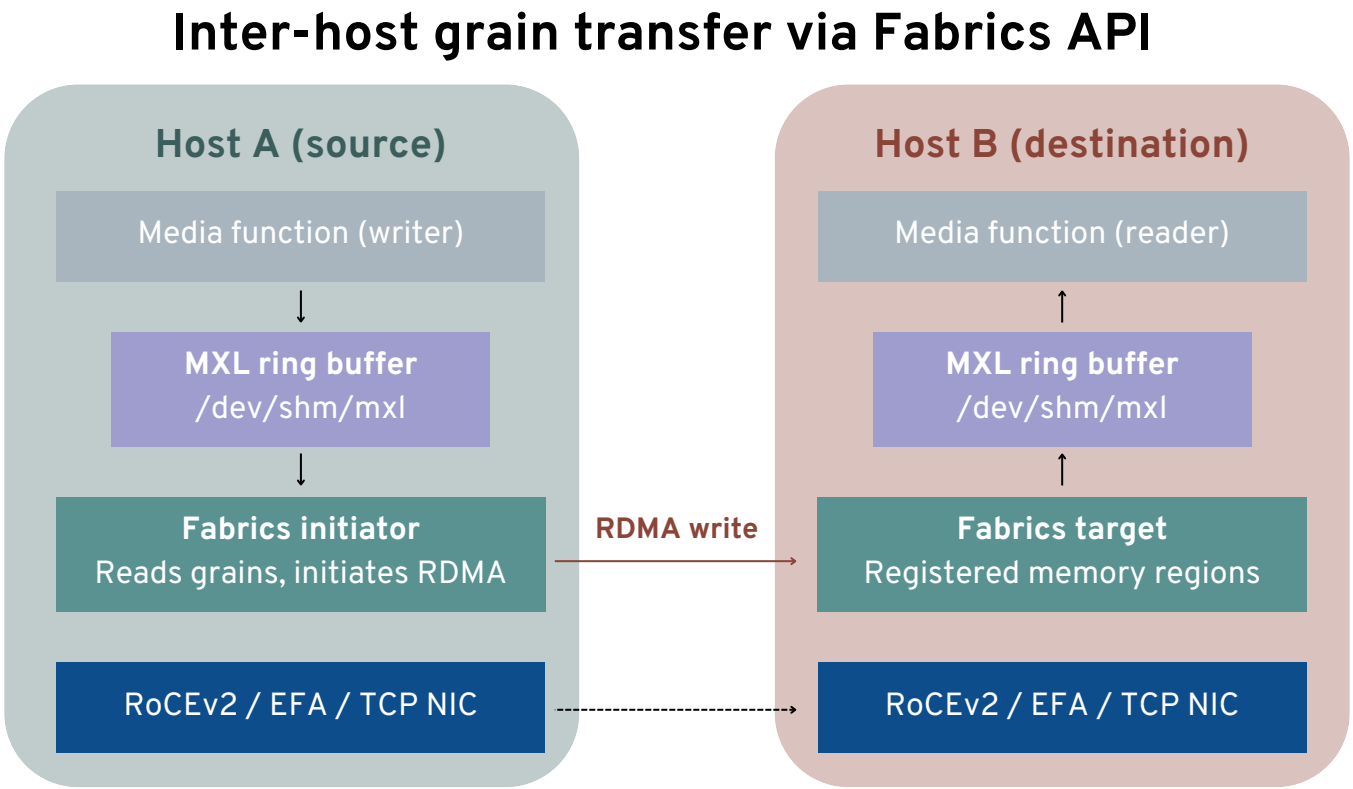
Audio flows use a single contiguous shared memory blob organized as de-interleaved channel ring buffers:

```
channel 0 buffer (bufferLength x sizeof(float) bytes)
channel 1 buffer (bufferLength x sizeof(float) bytes)
...
channel N-1 buffer
```

Readers access sample windows via `mxlFlowReaderGetSamples()`, which returns an `mxlWrappedMultiBufferSlice` handling the ring buffer wraparound as two contiguous fragments. Readers can access up to bufferLength/2 samples at once.

The Fabrics API - Extending Beyond a Single Host

MXL's v1.0 Flow API handles same-host shared memory. The Fabrics API extends MXL across hosts using RDMA (Remote Direct Memory Access) via the Linux libfabric abstraction layer.



Grain indices, flow IDs, and ring buffer geometry arre presserved across hosts

Figure 5: Inter-host grain transfer via the Fabrics API

Initiator/target model

Initiator: The side that has the data. Reads grains from a local flow reader and pushes them to remote targets via RDMA remote writes.

Target: The side that receives data. Exposes registered memory regions for the initiator to write into.

Supported fabric providers

Provider	Description
TCP	Linux TCP sockets (development/testing)
VERBS	libibverbs + librdmacm (InfiniBand / RoCEv2)
EFA	AWS Elastic Fabric Adapter
SHM	Shared memory (intra-host moves between memory regions)

Grain indices, flow IDs, and ring buffer geometry are preserved across hosts during replication.

The mxl-fabrics-proxy

The `jonasohland/mxl-fabrics-proxy` repository is a standalone sidecar process that bridges MXL flows between hosts using the Fabrics API. Written by Jonas Ohland (Riedel Communications), it uses C++ for the data plane and Go for the control plane.

C++ data plane (src/)

- main.cpp: Reads JSON config; runs as either Initiator or Target
- initiator.cpp: Opens DiscreteFlowReader, creates fabrics initiator, tight transfer loop
- target.cpp: Creates DiscreteFlowWriter, sets up fabrics target, commits received grains
- fabrics.cpp: C++ wrapper around raw C Fabrics API
- metrics.cpp: Exposes transfer metrics via Unix domain socket

Go control plane (go/)

- go/cmd/: Orchestration sidecar managing C++ proxy worker lifecycle
- go/pkg/worker/: Spawns and supervises C++ child processes
- go/pkg/initiator/: Manages subscriptions and domain discovery
- go/pkg/target/: Target management logic
- go/pkg/metrics/: Reads metrics from C++ process via Unix socket
- go/pkg/server/: HTTP API for orchestration

The Go sidecar discovers flows, spawns C++ workers in the appropriate role, and monitors health. The C++ workers handle the hot path with optional real-time scheduling.

Security Model

MXL leverages standard UNIX file permissions at both the domain and individual flow level:

- No shared IPC or process namespace required between containers
- Readers only map memory as PROT_READ. A compromised reader cannot corrupt flow data
- Advisory file locks (flock) detect stale flows; mxlGarbageCollectFlows() cleans orphans
- The mmap model works across containers mounting the same tmpfs volume

Flow Data Memory Layout

The flow metadata file (data) is a fixed 2048-byte structure stored in shared memory:

Offset	Size	Description
0x0000	0x04	version (currently 1)
0x0004	0x04	size (must stay 2048)
0x0008	0x80	mxlCommonFlowConfigInfo (ID, format, rate, batch hints, payload info)
0x0088	0x40	Discrete or Continuous FlowConfigInfo union
0x00C8	0x40	mxlFlowRuntimeInfo (headIndex, lastWriteTime, lastReadTime)
0x0108	0x6F8	Reserved padding (cache-line aligned)

Grain info structure

Field	Type	Purpose
version	unit32	Structure version (currently 2)
index	unit64	Epoch grain index for this ring buffer entry
flags	unit32	MXL_GRAIN_FLAG_INVALID if grain is invalid
grainSize	unit32	Total payload size in bytes
totalSlices	unit16	Number of slices in a complete grain
validSlices	unit16	How many slices are currently committed

Summary

MXL represents a fundamental rethink of how media moves inside software-defined broadcast infrastructure. Rather than treating every connection like a network stream, it treats media exchange as a shared-memory problem. This is the correct abstraction when devices are containers on the same compute cluster.

The key technical innovations:

- Ring buffers indexed by PTP-epoch nanoseconds provide a universal time-based address space
- Zero-copy reader/writer model with futex-based signaling eliminates all unnecessary data movement
- Slice-by-slice partial grain commits enable ultra-low-latency line-by-line processing
- The Fabrics API with RDMA extends the shared-memory model across hosts
- UNIX file permissions provide security without IPC namespace coupling
- Open-source, Linux Foundation governance avoids vendor lock-in

For teams building the next wave of broadcast facilities, MXL offers what network packets cannot: a direct, efficient path between co-located applications that treats media exchange as a memory problem rather than a networking problem.

References

- MXL SDK Repository: https://github.com/dmf-mxl/mxl
- MXL Fabrics Proxy: https://github.com/jonasohland/mxl-fabrics-proxy
- EBU MXL Initiative: https://tech.ebu.ch/dmf/mxl
- Linux Foundation MXL Project Announcement (April 2025)
- EBU Dynamic Media Facility Reference Architecture White Paper
- SMPTE ST 2110 Suite of Standards
- AMWA NMOS IS-04 Discovery and Registration Specification

TAG VIDEO SYSTEMS

www.tagvs.com