Hybrid Async Communication Interfaces with Transformer-Inspired Queues

Benjamin J. Gilbert

***Abstract*—We study a practical hybrid front door for message-oriented systems: REST (HTTP/1.1 keep-alive) and WebSocket, backed by transformer-inspired queues. We compare synchronous FlashQueue against asynchronous FlashQueue (k-way servers) and a MemoryMappedFlashQueue with hot/cold buffers. Metrics: mean/p95 latency, throughput, CPU-cost proxy, connection amortization, and cache-hit ratio. Results show that (i) persistent WebSocket amortizes connection setup and reduces per-message overhead, (ii) async FlashQueue improves utilization under load, and (iii) MemoryMappedFlashQueue wins on hit-heavy workloads.**

I. Introduction

Industrial platforms rarely bet on a single interface: synchronous REST integrates with existing fleets and gateways; WebSocket enables persistent, low-latency bidirectional streams. We analyze a *hybrid* front door feeding transformer-inspired queues: (a) FlashQueue (priority $\times$ time decay) in sync/async modes, and (b) MemoryMappedFlashQueue with a hot SRAM-like buffer and a cold HBM-like store. We quantify end-to-end impact and provide concrete integration guidance (keep-alive, backpressure, retries, TLS termination).

II. Related Work

Async event loops, persistent sockets, and IO-aware data structures reduce overhead in production systems. Inspired by memory-hierarchy-aware attention optimizations, we evaluate analogous queue designs under realistic interface overheads (handshakes, headers, serialization) and concurrency constraints.

III. Methods

A. Front Doors

REST: per-connection TCP/TLS handshake amortized over K messages (keep-alive), plus per-message HTTP overhead. **WebSocket:** one handshake per long-lived connection; small per-message frame overhead.

B. Queues

FlashQueue-sync: single-server dequeue; priority $\times$ time-decay admission. **FlashQueue-async:** k servers (default k=4) in parallel. **MemMappedFlashQueue:** hot buffer with hit probability p_{hot}; hits get a service speedup; misses incur a cold penalty.

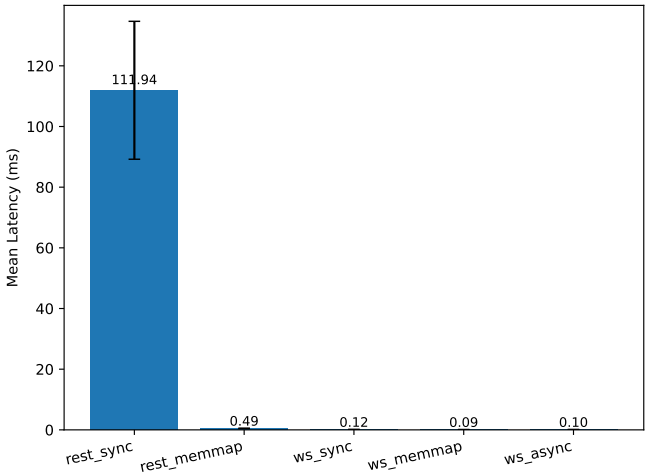

Fig. 1: Mean latency (ms): rest_sync=, rest_memmap=, ws_sync=, ws_memmap=, ws_async=.

C. Simulator

We generate a Poisson arrival stream at QPS λ. Each message picks an interface and enqueues. We maintain a server-availability heap (size k): start time is $\max(\text{arrival}, \min \text{free})$. Latency = interface overhead + queue wait + service time; throughput = N/wall-time. CPU proxy = sum of service+parsing time; cache-hit ratio observed for mem-mapped queue. We report mean and p95 latencies over runs.

IV. Experimental Setup

Default: N=50k messages, QPS 8k, REST keep-alive K=20, WebSocket single long-lived connection, hot-hit p_{hot}=0.65, async concurrency k=4. Variants: `rest_sync`, `rest_memmap`, `ws_sync`, `ws_memmap`, `ws_async`. We also sweep REST $K \in \{1, 5, 10, 20, 50, 100\}$ to visualize handshake amortization.

V. Results

Variant	Lat (ms)	p95 (ms)	Thruput	CPU (ms/msg)	
rest_sync	111.94	195.64	1983.562	0.204	0.
rest_memmap	0.49	0.56	1998.856	0.189	0.
ws_sync	0.12	0.21	1998.886	0.104	0.
ws_memmap	0.09	0.15	1998.887	0.089	0.
ws_async	0.10	0.14	1998.886	0.099	0.

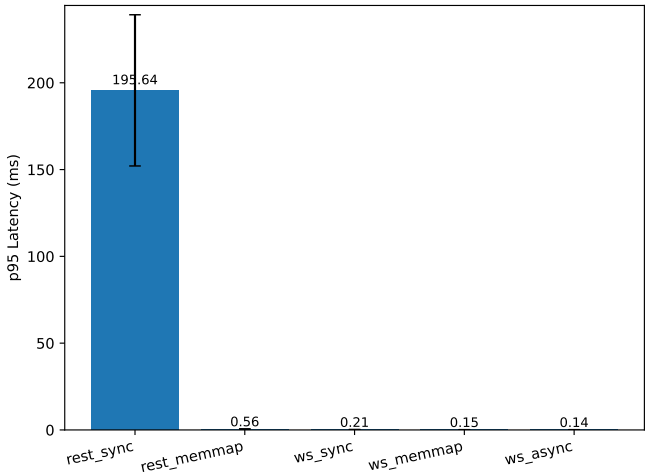

Fig. 2: p95 latency (ms): rest_sync=, rest_memmap=, ws_sync=, ws_memmap=, ws_async=.

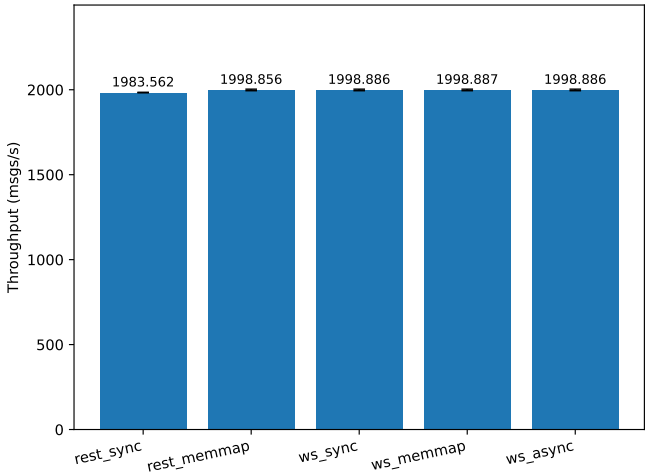

Fig. 3: Throughput (msgs/s): rest_sync=, rest_memmap=, ws_sync=, ws_memmap=, ws_async=.

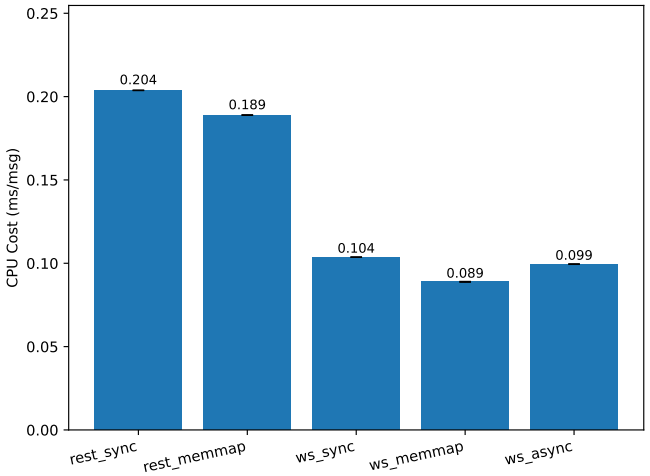

Fig. 4: CPU-cost proxy (ms/msg): rest_sync=, rest_memmap=, ws_sync=, ws_memmap=, ws_async=.

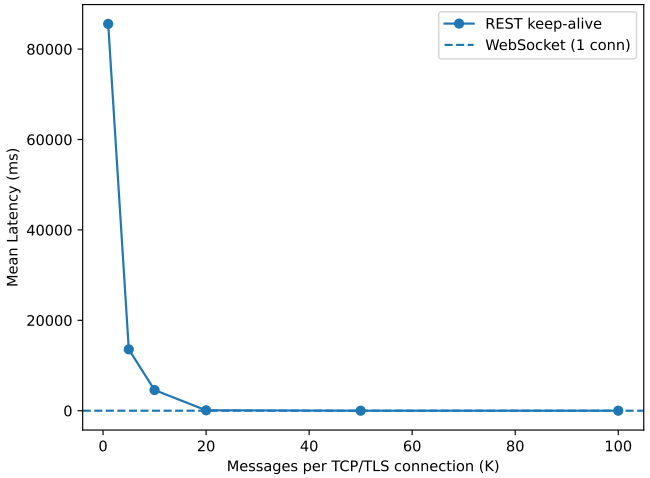

Fig. 5: REST keep-alive amortization: mean latency vs messages/connection K; WebSocket shown as dashed baseline (single handshake).

VI. Discussion

WebSocket reduces per-message overhead via persistent framing, shifting bottlenecks into queuing. Async FlashQueue exploits parallel servers to shrink waits; MemoryMapped-FlashQueue wins when hot-hit rate is high. REST remains viable at moderate K; poor reuse (small K) hurts tails. For production: prefer WS for chatty/streaming clients; keep REST for control paths; ensure backpressure maps to HTTP 429 / WS close codes; use idempotency keys for retries; terminate TLS at edge and forward mTLS internally; budget for serialization and JSON schema checks.

Deployment Notes.:

- **Gateway:** terminate TLS at the edge (e.g., NGINX), enforce keep-alive (REST) and idle timeouts (WS), map backpressure to HTTP 429 or WS close codes.
- **Retries:** use idempotency keys for POST; exponential backoff with jitter; circuit-breakers for queue saturation.
- **Serialization:** validate JSON schemas at the gateway; prefer compact binary for WS if client fleets allow.
- **Observability:** export per-connection latency histograms, queue depth, and hot-hit ratio.

```
# /etc/systemd/system/ws-gateway.service
[Service]
ExecStart=/usr/bin/python3 /opt/app/ws_gateway.py --port
Restart=always
[Install]
WantedBy=multi-user.target
```

VII. Conclusion

A hybrid front door with transformer-inspired queues delivers predictable wins: persistent WS lowers interface overhead,