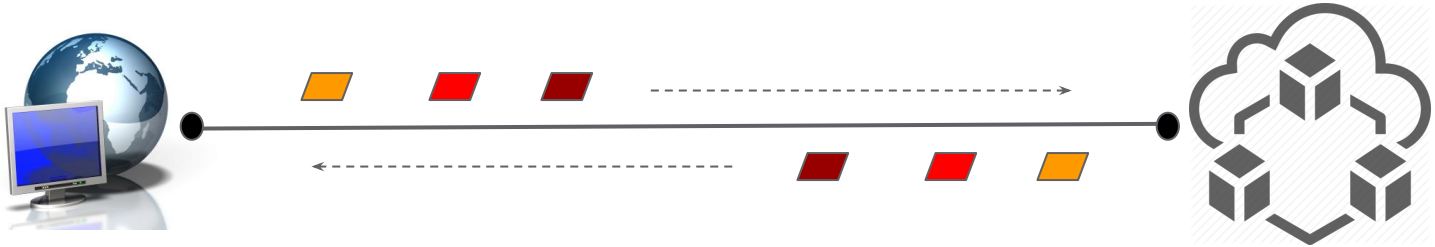
Real-time streaming APIs: From data center to Internet clients

Wenbo Zhu

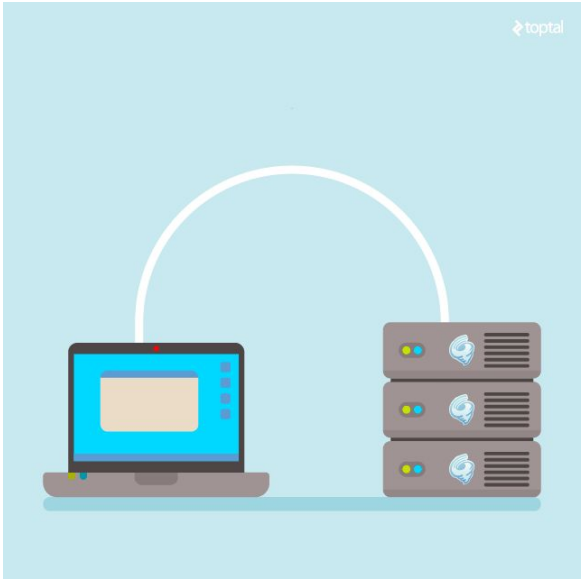
OSCON 2019 (Portland)

Outline

- Recap of streaming semantics
- Safe v.s. unsafe patterns
- The case for (not) having client-driven APIs



Recap of “streaming” semantics

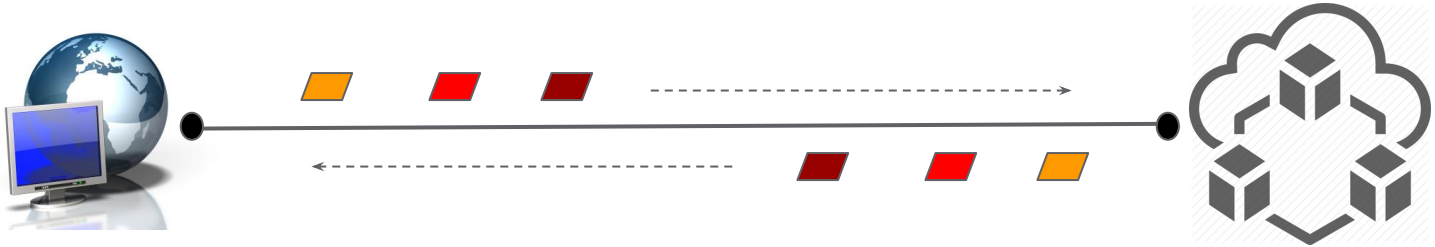


Myth: streaming is a transport feature

On the contrary, it’s a local, runtime concept.

It has everything to do with the (runtime) API.

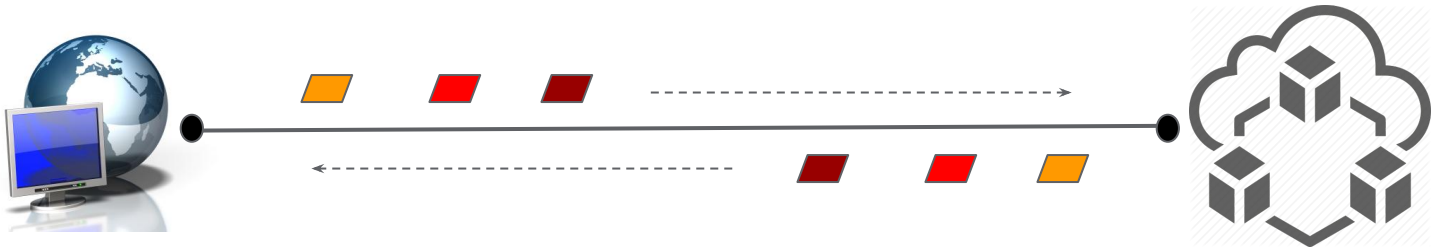
Most transports do one thing properly, i.e. streaming bytes.



Myth: it's streaming v.s. request-response (RPC)

Streaming APIs are an optimization over otherwise "atomic" RPCs.

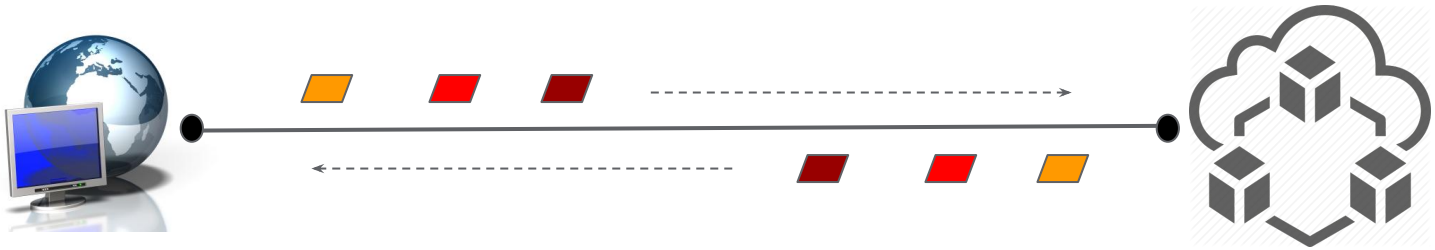
Even a "socket" style API can be constrained with RPC semantics: e.g. handshake, client-first half-close.



Semantics: client-server & causality

Handshake: the explicit roles of a client and server

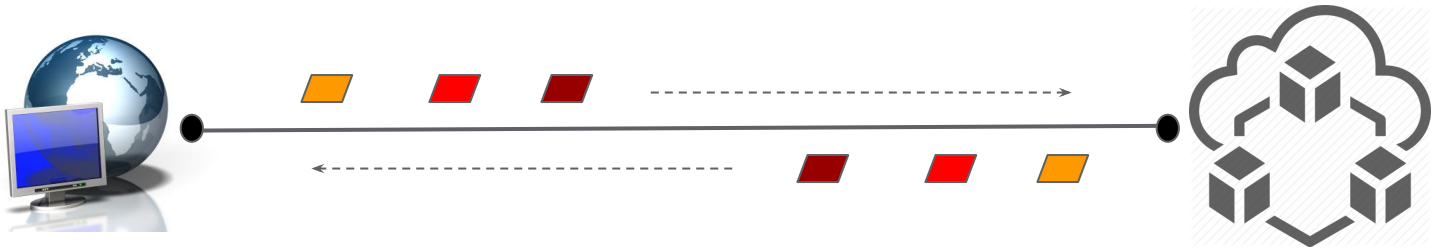
Request-response: input => output dependency



Semantics: runtime message delivery

Messages: byte chunks or atomic “objects”

Streaming: incremental delivery of transport bytes as messages

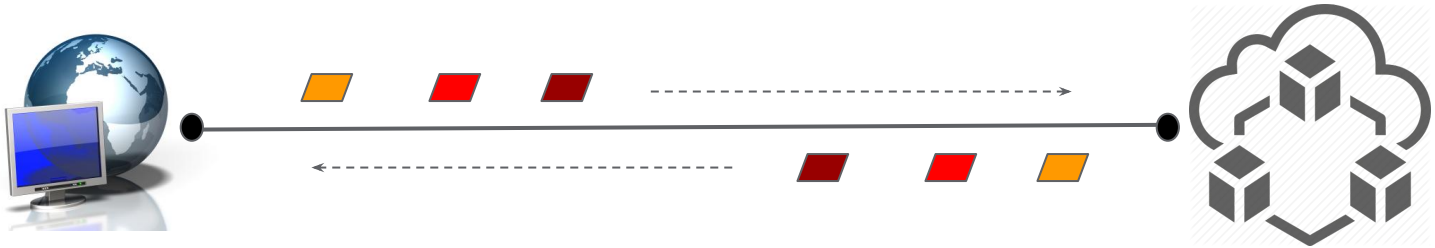


Semantics: EOF

Half-close: transport provided EOF message

To comply with the causality semantics, client first

HTTP: https://tools.ietf.org/html/draft-zhu-http-fullduplex

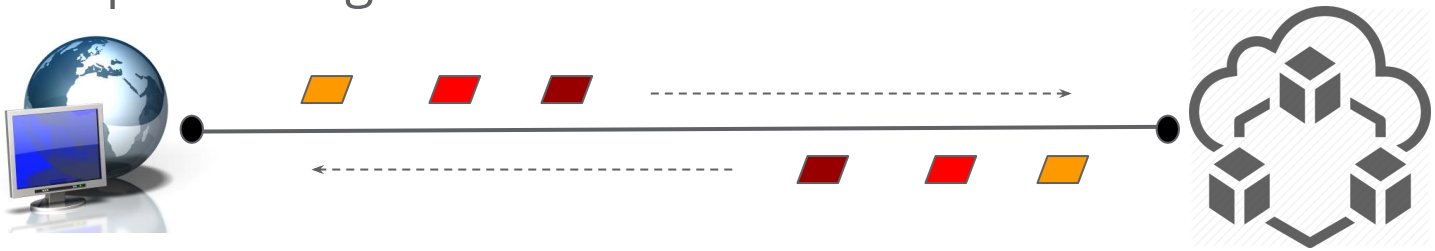


Streaming patterns: safe ones

As good as atomic RPCs

- streamed “download” : e.g. live media
- streamed “upload” : e.g. speech recognition
- simplex bidi - streamed “upload” followed by streamed “download” : e.g. voice translation

Abort (cancellation) may happen in the middle of request or response processing.



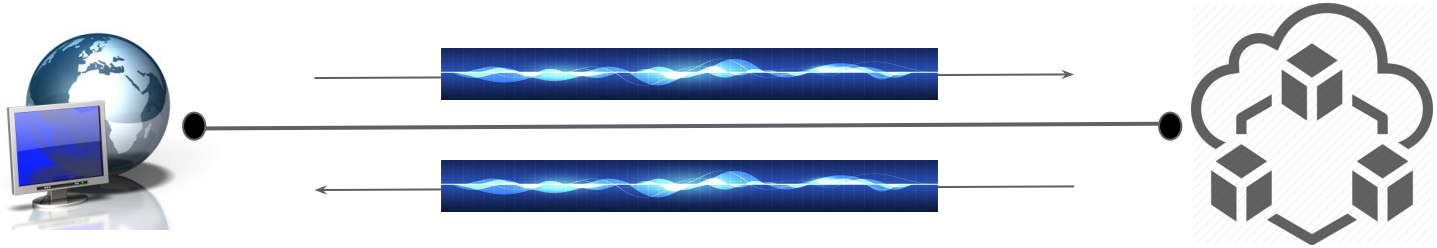
Streaming patterns: safe but complicated ones

Full-duplex (simultaneous) upload and download

Necessary runtime constraints

1. explicit handshake RPC
2. client-first half-close (or abort)

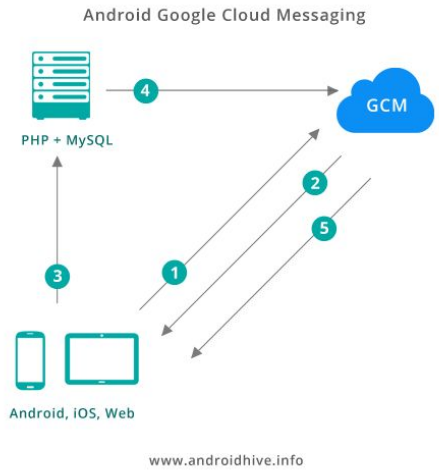
Causality is the key: e.g. video transcoding, voice assistant



Streaming patterns: unsafe ones

Long-lived simplex streaming, spontaneous server-sent messages

Typically the server-push use case, e.g. to avoid p2p communication

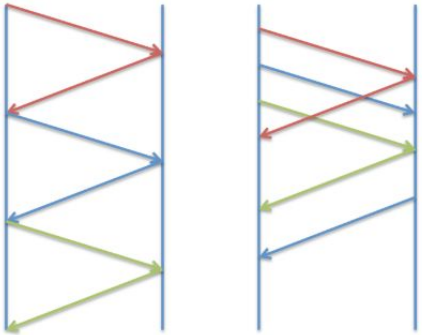


Streaming patterns: unsafe ones

Long-lived full-duplex streaming, with no request-response causality

An optimization over parallel RPCs for stateful applications

- efficiency, similar to batching
- move the complexity of ordering, retrying etc to a lower-level stack

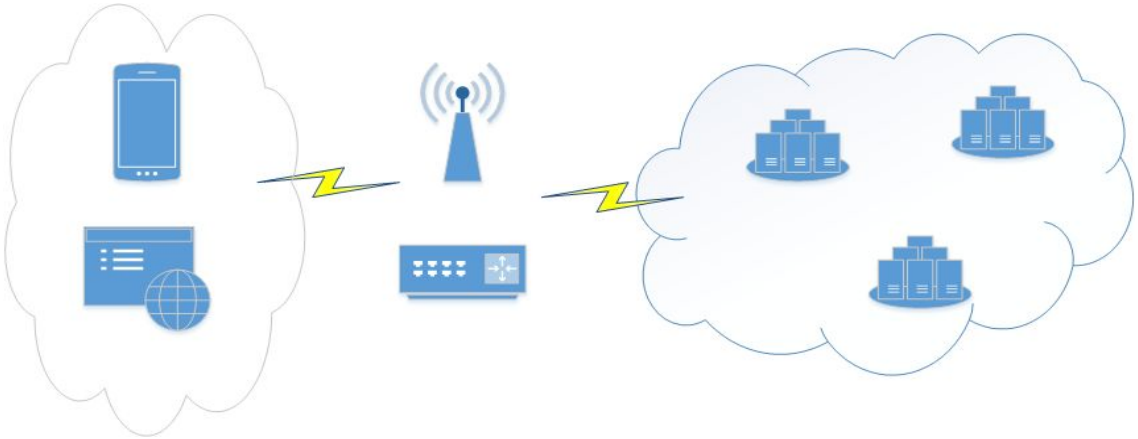


System properties

I. Proxy (Internet) friendly

Simplex HTTP, except for early errors (abort)

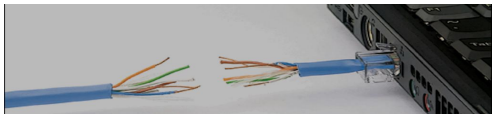
- legacy infra, firewalls, MITM ...

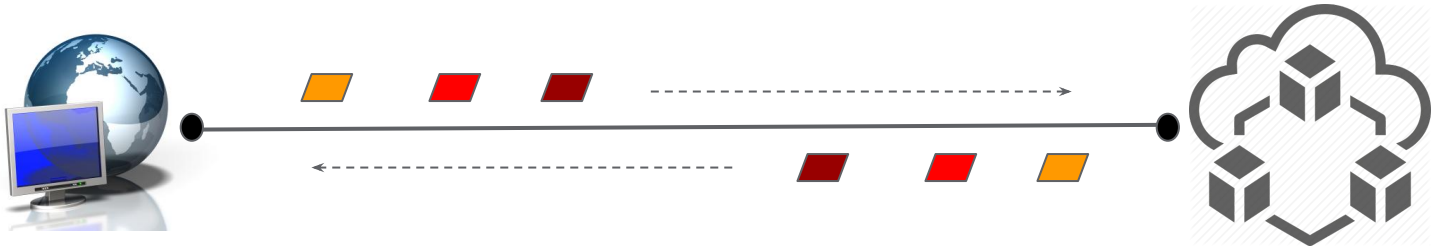


System properties

II. Fault-tolerance

Causality => implicit Ack





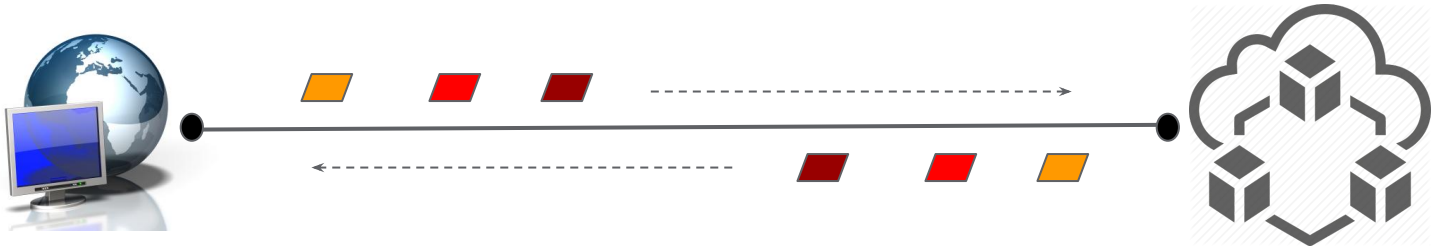
System properties

III. Scalability

Load balancing, DoS, security, redirect (failover) ...

Long-lived transactions => stateful architecture

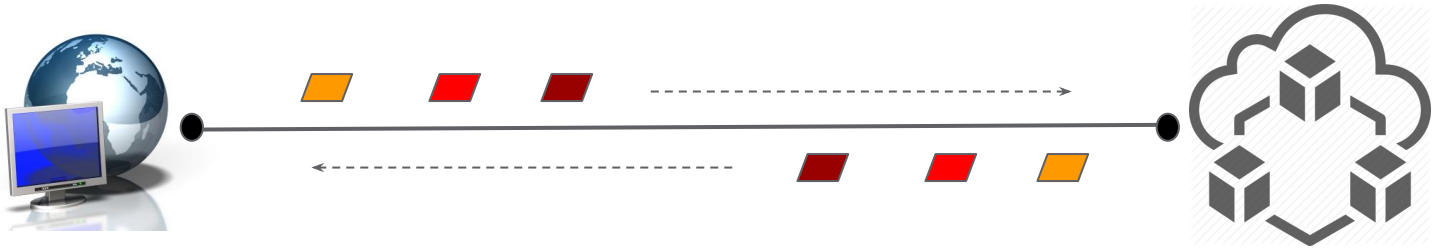
- in-band flow-control is a must
- draining has to involve applications due to in-flight messages



So, what's the deal?

Embrace those safe patterns, e.g. as supported by frameworks.

Stateful APIs are often necessary for performance reasons, i.e. unsafe patterns may still be adopted.

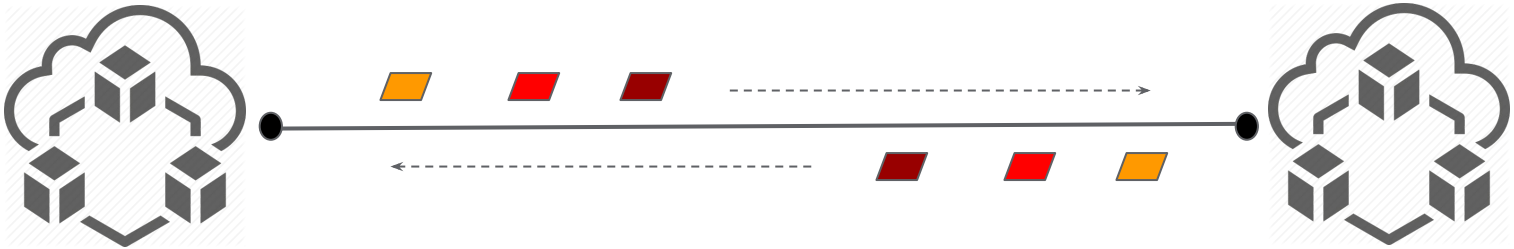


Client-driven: data center clients

Direct client-server connection

Unsafe patterns are “safe” to adopt

- failures are rarer, e.g. caused by peer failures
- failure-detection is more effective: less packet loss, short RTT



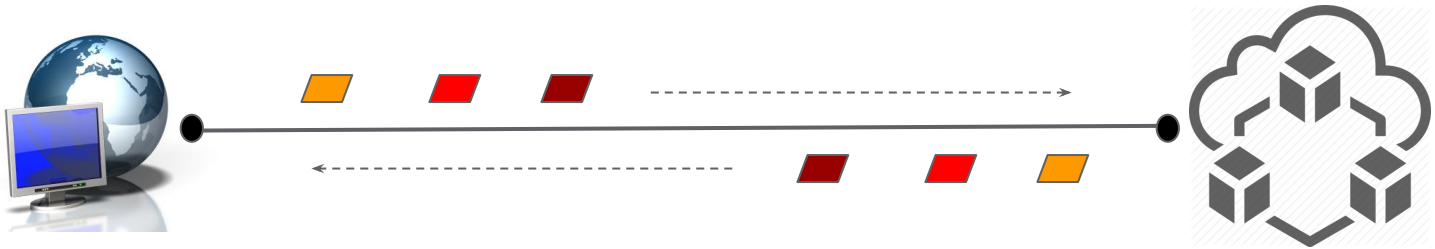
Client-driven: Internet clients

Avoid unsafe patterns.

Or you risk turning your APIs into a custom transport.

E.g. when you see any of the following in an API

- ack, keep-alive, ordering, resumption ...



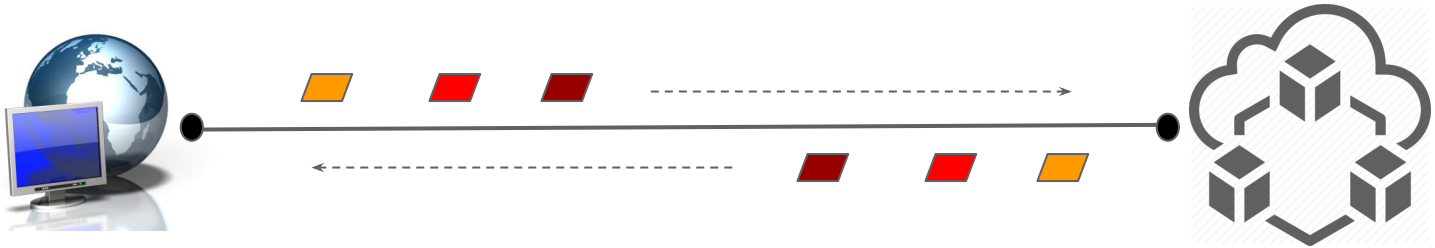
Different APIs for different clients?

Generally yes, to minimize complexity for API consumers

- far more client implementations, esp. for public APIs

Avoid addressing transport-level concerns as part of your API design.

... but, maybe you are building an infrastructure service.



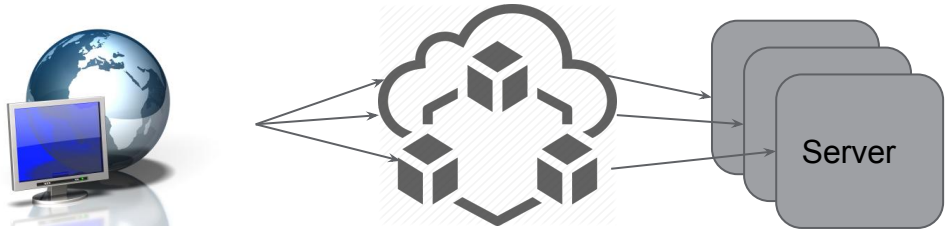
A “better” transport?

Transports (L4/7) are hop-by-hop

- some concerns are e2e by nature, e.g. ack of one-way messages

Transport features are not always visible to applications

- HTTP API in browsers, socket APIs ...



Transports in future

QUIC is coming

- connection migration
- multiplexed streams

Fallback to HTTP & TCP is needed.

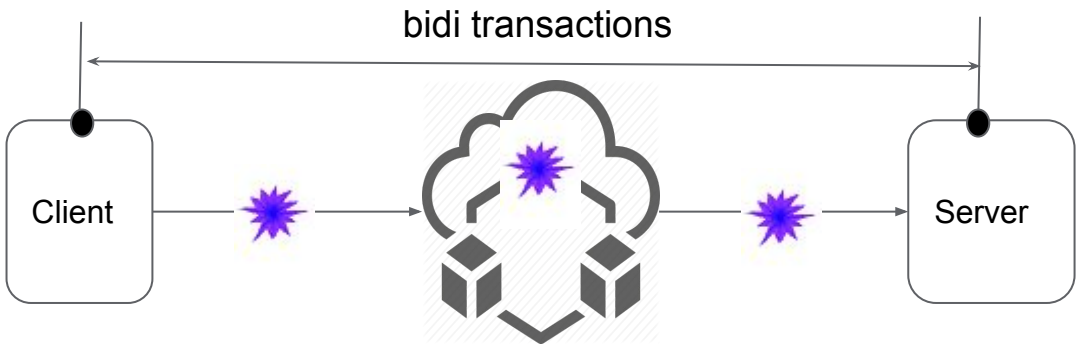
OSSification is a risk.

Framework-based solutions

github/bidiweb: e2e streaming over short-lived streaming GETs and parallel atomic POSTs

Long-lived streaming is only visible to the endpoints.

Overhead compared to transport-mapped transactions



Conclusions

Those safe streaming patterns are, safe, to adopt.

Avoid long-lived, stateful APIs with Internet clients. I.e. the old-school 3-tier architecture still wins.

Lastly, do not turn your streaming API into a custom transport.