Infinite Parallel Universes: State at the Edge

Peter Bourgon · Fastly
Infinite Parallel Universes: State at the Edge

Peter Bourgon · Fastly
Infinite Parallel Universes

Context
Architecture
Protocol
Complications
Conclusions
Fastly Expands Serverless Capabilities With the Launch of Compute@Edge

SAN FRANCISCO, Calif. (Nov. 6, 2019) – Fastly, Inc. (NYSE: FSLY), provider of an edge cloud platform, today announces the beta launch of Compute@Edge, a language-agnostic compute environment. The major milestone marks Fastly’s edge computing capabilities and the company’s innovation in performance.

Fastly’s Compute@Edge is designed to empower developers to develop advanced edge applications with greater security, more robust logic, and lower latency performance.

Developers are being empowered to create new and improved digital experiences quickly and with low operational overhead. Instead of spending time on operational overhead, developers are now able to focus their own technology choices around the cloud platforms, services, and programming languages needed. Rather than spend time on operational overhead, they can continue to reinvent the way end users live, work, and play on the web.

Fastly Compute@Edge gives developers the freedom to push complex logic closer to end users, and reduces the time to innovate by allowing developers to focus on strategy that drives their companies forward.

Fastly Compute@Edge Leverages Speed for Global Scale and Security

At 35.4 microseconds, Fastly’s Compute@Edge environment offers a 10x faster startup time than any other solution on the market. Benefits of Compute@Edge, powered by Fastly’s WebAssembly compiler and runtime, include:

If Fastly provides compute at the edge, what about state?
Fastly Compute@Edge Level

At 35.4 microseconds, Fastly’s time than any other solution on
What are we doing here
- Not a general purpose database
- Operate in the request lifecycle
- Local (POP) reads and writes
- Eventually consistent
- State conflicts are normal
  \[ \Rightarrow \text{Sort of a writeable cache} \]
**Data model**

- Lean in to the physical constraints
- A single, global truth? — *No, a fiction!*
- Multiple, simultaneous truths — *Reality!*
- Converge toward a stable global state
State primitive

- CRDT
- Type-specific methods + **Merge**
- Associative, commutative, idempotent
- Tolerates out-of-order, duplicate merges
  
  \[ \Rightarrow \text{Reduces higher-order complexity} \]
Integer addition

- Associative? \((a + b) + c = a + (b + c)\) ✔
- Commutative? \(a + b = b + a\) ✔
- Idempotent? \(a + a = a\) ✗

⇒ Not a CRDT
Set union

- Associative? \((\{a\} \cup \{b\}) \cup \{c\} = \{a\} \cup (\{b\} \cup \{c\})\) — Yes
- Commutative? \(\{a\} \cup \{b\} = \{b\} \cup \{a\}\) — Yes
- Idempotent? \(\{a\} \cup \{a\} = \{a\}\) — Yes

\[\Rightarrow A \text{ CRDT}\]
{1}
Join semilattice
Infinite Parallel Universes

- Context
- Architecture
- Protocol
- Complications
- Conclusions
1. Single write primary

Familiar. Consistent. Slow. Doesn't satisfy local read/write criteria.
2. Gossip network

Lots of connections. Where do objects live? How do objects propagate? etc.
3. Hub-and-spoke

Fewer connections. Objects live in root. Replica like LRU cache.
Context
Architecture
Protocol
Complications
Conclusions
Protocol (i)
- How do objects get from A to B?
- Track objects that receive any request
- Batch them into epochs
- Regularly emit batches
Protocol (ii)
- How do objects get from B to C?
- Bad answer: *push* from B to C
- Key insight: sites know which objects have been requested
- Better answer: *pull* relevant objects to C from B — *sync*
Site
1. Write K1=V1
2. Read K2: *miss*
3. Write K3=V3

State
{K1:V1}
K3:V3

Interest
{K1}
K2
K3}
Sync step 1
After some time, make a SyncRequest with all interesting keys and values.
Sync step 2
Send the SyncRequest to the upstream.

SyncRequest
{K1:V1
K2:--
K3:V3}
Sync step 3
Upstream merges incoming data to its own data.

\[
\text{State} \cup \text{SyncRequest} = \text{New state}
\]

\[
\{K1:V5, K2:V6, K4:V7\} \cup \{K1:V1, K2:--, K3:V3\} = \{K1:V5 \cup V1, K2:V6, K3:V3, K4:V7\}
\]

Infinite Parallel Universes · Protocol
@peterbourgon · Fastly
Sync step 4
Upstream returns a SyncResponse to the site, with latest value for each requested key.
Sync step 5

Site merges the SyncResponse to its own state, and resets the interest set.

\[
\text{State} \cup \text{SyncResponse} = \text{New state}
\]

\[
\{K1:V1, K2:V6, K3:V3\} \cup \{K1:V8, K2:V6, K3:V3\} = \{K1:V1 \cup V8 = V8, K2:V6, K3:V3 \cup V3 = V3\}
\]
Properties of sync (i)
- Synchronous
- The only way data moves
- Bandwidth minimized
- Data sets minimized
- Authoritative upstream
Properties of sync (ii)
- Schedule is flexible
- Missed syncs impact liveness, not correctness
- Eventually consistent
- Quiet by default

Site

State
{K1:V8, K2:V6, K3:V3}

Upstream

State
{K1:V8, K2:V6, K3:V3, K4:V7}
Infinite Parallel Universes · Protocol

@peterbourgon · Fastly
Infinite Parallel Universes · Protocol
Incoming user requests

- Requests are hashed and proxied to N replicas
- Response is the union of all returned CRDTs
- Compare union with individual responses
- Discrepant replicas easily identified, fixed
Outgoing sync requests
- Each instance has different set of objects
- But CRDTs tolerate over-merging
- Instances may sync totally independently
- Upstream state converges to stability
Incoming sync requests
- User request = 1 key \cdot Op
- Sync = N keys \cdot Merge
- Otherwise identical
- Choose any upstream replica
Fractal design

- N-ary tree
- Can be hierarchical
- May insulate against regional connectivity issues
- Tradeoff between liveness and capacity
Smart primitives, simple systems
- One operation for moving data
- No coördination between
- Faults need no handling besides retry
  \[ \Rightarrow \text{All from CRDT properties} \]
Infinite Parallel Universes

Context
Architecture
Protocol
Complications
Conclusions
Complication: Read miss
- First read of an object will likely miss
- Object may "appear" later
- **Approach:** blocking sync request to upstream
- Better UX, worse latency
Complication: API design
- CRDTs can be nonintuitive to program with
- Applications \textit{really like} a single global truth
- General-purpose CRDT-based state layer not obvious
- \textbf{Approach:} narrowly-scoped APIs $\times$ time
Complication: Resources
- Reliability via duplication of effort — cost to bandwidth
- CRDTs larger than normal data types — cost to storage
- **Approach**: careful attention to cost-per-byte
Complication: Quantum entanglement

- Quantum entanglement renders all of this irrelevant
- Pro: easy, instantaneous communication invariant to distance
- Con: I'm probably out of a job
- Approach: 🤷
Context
Architecture
Protocol
Complications
Conclusions
State at the edge — IMO

- There are *inescapable* constraints at large physical scale
- Old abstractions (single global truth) break down
- New abstractions (multiple parallel truths) necessary
- Reliable systems require more robust primitives (CRDTs)
Infinite Parallel Universes: State at the Edge

Peter Bourgon · Fastly