Towards a Solution to the Red Wedding Problem

Christopher S. Meiklejohn
Université catholique de Louvain
Instituto Superior Técnico

Heather Miller
Northeastern University
École Polytechnique Fédérale de Lausanne

Zeeshan Lakhani
Comcast Cable

Abstract

Edge computing promises lower latency interactions for clients operating at the edge by shifting computation away from Data Centers to Points of Presence which are more abundant and located geographically closer to end users. However, most commercially available infrastructure for edge computing focuses on applications without shared state. In this paper, we present the *Red Wedding Problem*, a real-world scenario motivating the need for stateful computations at the edge. We sketch the design and implementation of a prototype database for operation at the edge that addresses the issues presented in the *Red Wedding Problem* and present issues around implementing our solution on commercial edge infrastructure due to limitations in these offerings.

1 Edge Computing

Edge computing promises lower latency interactions for clients operating at the edge by shifting computation away from Data Centers (DCs) to Points of Presence (PoPs) which are more abundant and located geographically closer to end users. Not only do PoPs reduce latency, but they also serve to alleviate load on origin servers enabling applications to scale to sizes previously unseen.

Recently, large-scale cloud providers have started providing commercial access to this edge infrastructure through the use of "serverless" architectures, giving application developers the ability to run arbitrary code at these PoPs. To support this, client code is normally executed inside of a transient container where the application developer must resort to using storage that's typically provided only at the DC. Therefore, to take most advantage of the edge, application developers are incentivized to use these architectures for applications where there is no shared state, thereby ruling out one of the most common types of applications developed today: collaborative applications which rely on replicated, shared state.

The first generation of edge computing services provided by Content Delivery Networks (CDNs) let customers supply code that will be run at PoPs during the HTTP request-response cycle. Akamai [1] allows customers to author arbitrary JavaScript code that operates in a sandboxed environment and is permitted to modify responses before the content is returned to the user. Fastly [7] gives users the ability to extend their CDN with similar rewrite functionality by authoring code in the Varnish Configuration Language, which is also executed in a restricted environment. In both cases, customers are not allowed to inspect or modify content at the PoP and can only modify requests to and from the origin and to and from the end user.

Second generation services, such as Amazon's Lambda [2] and its extension to the edge, Lambda@*Edge*, enables the customer to write stateless functions that either run inside one of Amazon's region availability zones or at any number of the hundreds of PoPs it has through its CloudFront caching service. While Lambda provides binary execution in a restricted containerized environment, Lambda@*Edge* is limited to Node.JS applications. Both Google's *Cloud Functions* and Microsoft Azure's Functions at IoT Edge provide similar services to Lambda and Lambda@*Edge*.

This paper defines a real-world scenario for motivating stateful computations at the edge, titled the *Red Wedding Problem*. We sketch the design and implementation of a prototype eventually consistent, replicated peer-to-peer database operating at the edge using Amazon Lambda that is aimed at addressing the issues presented by the *Red Wedding Problem*. In realizing our prototype on real-world edge infrastructure, we ran into numerous limitations imposed by Amazon's infrastructure that are presented here.

2 The Red Wedding Problem

We present the *Red Wedding Problem*[1][2]: an industry use case presented to us by a large commercial CDN around the handling of traffic spikes at the edge.

Game of Thrones [9] is a popular serial fantasy drama that airs every Sunday night on HBO in the United States. During the hours leading up to the show, throughout the hour-long episode, and into the hours following the airing, the Game of Thrones Wiki [6] experiences a whirlwind of traffic spikes. During this occurrence, articles related to characters that appear in that episode, the page for the details of the episode, and associated pages to varying concepts referenced in the episode undergo traffic spikes—read operations upon viewing related pages—and write spikes while updating related pages as events in the episode unfold.

While handling read spikes is what CDNs were designed for, the additional challenge of handling write spikes is what makes the *Red Wedding Problem* interesting. More specifically, the *Red Wedding Problem* requires that the programmer be made aware of and optimize for the following constraints:

- **Low latency writes.** By accepting and servicing writes at the PoP, user's experience lower latency requests when compared to an approach that must route writes to the origin;
- **Increased global throughput.** By accepting writes at the PoP, and avoiding routing writes through to the origin DC, write operations can be periodically sent to the origin in batches, removing the origin DC as a global throughput bottleneck.

However, several design considerations of the *Red Wedding Problem* make the problem difficult to solve.

- **Storing state.** How should state be stored at the edge, especially when leveraging "serverless" infrastructures at the edge which are provided by most cloud providers today?
- **Arbitrating concurrency.** How should concurrent writes be arbitrated when accepting writes at the edge to minimize conflicts and maximize batching?
- **Application logic.** As clients do not communicate with the database directly in most applications, how should application logic be loaded at the edge?

[1] Private communication, Fastly.

[2] Many examples of the Red Wedding Problem exist: live European football game commentary posted on Reddit is one such example.

3 Solving the Red Wedding Problem

Our proposed solution is presented in Figure 1. In solving the *Red Wedding Problem*, we had the following design considerations:

- **Application logic at the edge.** Application logic for authentication and authorization, as well as for mapping user requests into database reads and writes, must also live at the PoP;
- **Elastic replica scalability at the PoP.** To avoid moving the bottleneck from the origin to the PoP, there must be elastic scalability at the PoP that supports the creation of additional replicas on demand;
- **Inter-replica communication.** Under the assumption that communication within the PoP is both low latency and cost-effective, as traffic never needs to leave a facility, replicas executing within the PoP should be able to communicate with one another for data sharing. This avoids expensive round-trip communication with the origin server;
- **Convergent data structures.** As modifications will be effecting data items concurrently, the data model needs to support objects that have well-defined merge semantics to ensure eventual convergence;
- **Origin batching.** To alleviate load on the origin server updates should be able to be combined to remove redundancy and leverage batching to increase throughput when propagating updates periodically to the origin server.

Our solution assumes that application code is containerized and can be deployed to the PoP to interpose on requests to the origin: enabling the application to scale independently at the edge. As demand from end users ramps up, more instances of the application are spawned to handle user requests. These instances of the application generate read and write traffic that is routed to the database. In traditional architectures, these application instances would normally communicate with a database operating at the DC.

We use a containerized database to enable data storage at the PoP. Each of these database instances would be scaled in the same manner as the application code running at the PoP, and state would be bootstrapped either from other replicas operating in the same facility, or the DC. These instances of the database would be instantiated on demand and interpose on read and write operations from the application code inside of the PoP. By leveraging other replicas, the system can increase the probability of using the freshest state possible.

Finally, the use of concurrent data structures can be used at the edge to avoid conflicting updates that cannot

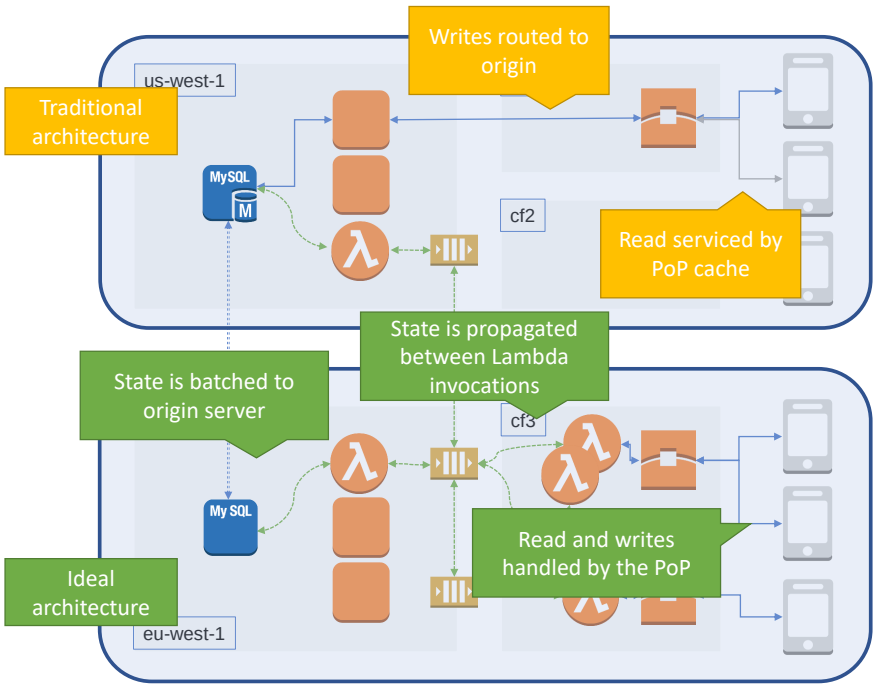

Figure 1: Architecture diagram with both the traditional and ideal architectures presented, where (traditional) writes are sent to the origin and reads are serviced by the edge; and (ideal) writes are stored in transient database instances at the edge as the load increases.

be merged. For instance, several previous works [5, 13, 3] have identified how to build abstract data types where all operations commute or have predefined merge functions to avoid issues where changes do not commute.

4 Implementation

For our prototype, we chose Amazon's Lambda as a starting point over traditional persistent VMs. The reason for this choice is that functions written for Amazon's Lambda environment can easily be extended to run in Amazon's edge environment, Lambda@*Edge*. If we had chosen traditional VMs, our application would be constrained to operate only in Amazon's DCs.

Our prototype enables us to store content inside of transient serverless invocations using Lambda. To achieve this, we have taken an open source distributed key-value database [10, 11] written in Erlang and have embedded it inside of a Node.JS application that is deployed to Lambda in several AWS regions.

4.1 Lambda

Amazon's Lambda is a stateless computation service for end users, allowing them to upload functions that will be invoked in response to triggered events. With Lambda, users specify the events that uploaded code should respond to and users' application code is guaranteed to automatically scale, when necessary, by creating additional invocations to deal with the increasing demand.

Invocations in Lambda are designed to be opaque. Users upload a compressed file containing their application code and upon the first event, the application code will be decompressed and executed inside of a container. When the invocation completes, the container is paused until the next invocation; however, after a period of somewhere between 5 to 60 minutes, the container will be terminated. Containers will be reused for invocations when possible, but concurrent invocation will cause additional containers to be created on demand. This mechanism is transparent to the end user, as the user only interacts on a per-event level.

As a result of this, invocations are incentivized to be completely stateless, or authorized and able to store and fetch required state from an external service, such as S3 or DynamoDB, as Amazon recommends.

4.2 Prototype

We built an eventually-consistent replicated peer-to-peer database that runs in Lambda. The interface provided to the user is Redis-like; read and update operations are

identified by a unique key and a data structure type that are issued against the database.

Data structures provided by the data store are Conflict-Free Replicated Data Types [13] (CRDT), distributed data structures with a merge function to ensure that state remains convergent in the event of concurrent modifications at multiple locations. CRDTs come in a variety of different flavors, providing distributed versions of many common data types: ie. booleans, sets, dictionaries.

Operations are always performed at a local replica and asynchronously propagated to other nodes in the system using an anti-entropy [4] protocol: this prevents blocking the system for write acknowledgements.

When the Lambda function is invoked, an instance of the database starts up. There can be multiple instances of the database running in each Amazon region. Every time a database instance is invoked, it kicks off a compulsory anti-entropy session. The databases bootstrap one another by running anti-entropy sessions over AMQP. All data structures in the database are CRDTs in order to avoid conflicting updates.

4.3 Limitations of Lambda

In building our prototype on Lambda, we ran into a number of complications. We discuss those here.

Inter-replica communication. Inter-replica communication is required for the compulsory anti-entropy sessions amongst database replicas. Lambda does not allow processes to bind ports inside of the container they are executing in. Therefore, our key-value store was unable to open sockets and receive incoming connections from other nodes in the system. To work around this limitation, we used an external message queue service that provided AMQP and sent out all of the inter-replica communication on this transport layer. Given nodes could not accept incoming connections, we established connections out to an external AMQP broker and reused those connections to receive traffic from other nodes.

Concurrent invocations. Utilizing multiple invocations at the same time is key to elastic scalability and the operation of multiple replicas. Lambda's unique design does not provide the developer any insight to the number of containers that are currently being invoked, and scale-out is transparent as demand calls for it. Therefore, we needed a mechanism for all nodes to identify one another in order to route messages to each other and participate in an anti-entropy session. To achieve this, we set a single topic on the AMQP broker for membership communication. On this specific channel, a single CRDT set containing all of the members of the cluster gets periodically broadcast to all nodes in the system. Upon receipt of this membership information, each node updates its local view of membership.

Per-invocation work. Keeping the instances alive long enough to perform anti-entropy is important for scaling the throughput of the system and ensuring no updates are lost. Each Lambda has a maximum invocation of 300 seconds. Therefore, we ensure that a Lambda is invoked per-region faster than that interval, and after servicing a request, each invocation performs a compulsory anti-entropy session with its peers to populate incoming replica state and establish that existing replicas disseminated their state before termination. Scaling out the number of concurrent invocations only required increasing this interval per-region.

4.4 Lambda@*Edge* Challenges

Operating our key-value store on Lambda was the necessary first step towards moving to Lambda@*Edge*, Amazon's extension of Lambda to CloudFront PoPs. Once running on Lambda@*Edge*, we should be able to interpose on writes to and from the origin servers through CloudFront, and, therefore, provide low-latency writes to clients, serviced from their local and closest PoP, rather than from their origin server.

In planning our migration to Lambda@*Edge*, we ran into a number of other complications.

Message brokering at the edge. Inter-replica communication at the PoP is important for keeping the anti-entropy sessions as efficient as possible and increasing system scalability. Since Lambda invocations cannot communicate with one another directly, this requires that Lambdas must communicate through their closest Amazon region and availability zone, inflating latency costs to the local availability zone for anti-entropy between instances. As nodes will communicate solely with their local broker, and therefore may not observe messages in the same order as other nodes in the system, our convergent CRDT-based model is essential.

Application code. Execution of application code at the edge enables our system to keep most of the processing local, within the PoP, and not rely on the DC. Therefore, our design assumes that users will be able to run a component of their application at the PoP.

4.5 Infrastructure Support

In hypothesizing which features would have allowed us to better construct our prototype, we would highlight the following, currently unsupported, features.

Instance visibility. Awareness and insight into what the other instances are executing at the edge would assist applications in concurrency control or for data sharing.

Notification of termination. Giving applications a way to register and relay a notification before instance termination would allow instances to handoff data or tasks to other running instances in advance.

Local communication. Providing a message broker in the PoP or providing the ability to directly pass messages between different invocations would allow for lower latency communication and serve to reduce latency.

4.6 Preliminary Evaluation

We setup an experiment to ensure that we had communication working between different Lambda invocations. Our experiment follows a simple workflow: a Lambda function is deployed, that when invoked by a HTTP event, performs an anti-entropy session with the other concurrently invoking replicas and then sleep for 20 seconds before termination.

Lambda overhead. Upon invoking a Lambda for the first time after deployment, we observed that it takes 7 seconds for the container to be deployed and the application code to be executed. Subsequent invocations of that Lambda, once paused after the initial invocation, occur after less than 1 second when measuring from request to application code. We did not have the ability to measure with more precise granularity using the API provided by Amazon. We also observed that instances usually stay paused from anywhere between 2 to 5 minutes after the last invocation before they are reclaimed and the next request is forced to pay the 7 second initialization cost.

Concurrent invocations. In order to have concurrent invocations of the same Lambda function, a request must be issued while an existing invocation is being processed; otherwise, the request will go to a paused instance. To achieve this, we set a timer inside of application that prohibits termination of a request until 20 seconds have elapsed, providing ample time to issue a subsequent request that will guarantee additional invocations.

Function wrapper overhead. Erlang is not one of the directly supported platforms on Lambda. Therefore, following Amazon's recommendation, we wrap our application in a Node.JS wrapper that spawns it using a child process. This introduces additional overhead, as communication from the outer Node.JS wrapper to the Erlang process can introduce an observed overhead of 5 to 20 milliseconds.

Inter-replica latency. We observed the latency for sending 64 byte payloads between two Lambda invocations located in the same DC using our AMQP transport layer is roughly 2 to 3 milliseconds. We assume that if an AMQP provider was available in the Lambda@*Edge* setting, it would provide similar performance.

5 Related Work

ExCamera [8] is a system for parallel video encoding developed on Lambda that uses a work-queue pattern for partitioning work across many concurrent invocations.

ExCamera relies on the use of a rendezvous server for passing messages between concurrent invocations to work around the inter-replica communication problem. State in the ExCamera system is stored in Amazon S3, rather in the instances themselves. Finally, ExCamera was primarily designed for work dispatch and as fast as possible concurrent processing by a large number of Lambda instances whereas our design focuses on transient state management at the edge to reduce real-time latency costs during traffic spikes.

CloudPath [12] is a system designed for *path computing*: a generalization of edge computing where nodes with different compute and storage capabilities are arranged hierarchically based on their proximity to the edge. Both state and processing are partitioned and replicated in the hierarchy, enabling application optimizations by moving processing closer to the end user.

Requiring more infrastructure support, each node in the hierarchy is a DC composed of application servers for hosting containerized stateless application code, servers for caching read operations, and a Cassandra cluster for data storage. Similiarly, operations occur against local replicas and state is persisted at the root. Replicas are always bootstrapped by one or more ancestors in the hierarchy. Writes are timestamped and sequenced at the root and concurrent operations are arbitrated by picking the largest timestamp: this may result in shipping operations that will be lost due to arbitration or are redundant.

6 Conclusion

Recent innovations in publicly available edge computing services, such as Amazon's Lambda@*Edge*, enable developers to run application code closer to end users, taking advantage of proximity resulting in both lower latency interactions and increased scalability. However, most of these services available only enable stateless computation at the edge or require collaborative applications that operate with shared state use a data store that's located at one of the provider's data centers, thereby reducing the full potential for edge computing. This paper presents an alternative design for a system that enables these interactions at the edge.

Acknowledgements. This work is partially funded by the LightKone project in the European Union Horizon 2020 Framework Programme under grant agreement 732505 and by the Erasmus Mundus Doctorate Programme under grant agreement 2012-0030.

References

[1] AKAMAI TECHNOLOGIES. Akamai: Cloudlets. `https://cloudlets.akamai.com/`. Accessed: 2018-03-14.

[2] AMAZON WEB SERVICES. AWS Lambda. `https://aws.amazon.com/lambda/`. Accessed: 2018-03-14.

[3] BURCKHARDT, S., FÄHNDRICH, M., LEIJEN, D., AND WOOD, B. P. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, pp. 283–307.

[4] DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (1987), ACM, pp. 1–12.

[5] ELLIS, C. A., AND GIBBS, S. J. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1989), SIGMOD '89, ACM, pp. 399–407.

[6] FANDOM. Game of Thrones Wiki. `http://gameofthrones.wikia.com/wiki/Game_of_Thrones_Wiki`. Accessed: 2018-03-19.

[7] FASTLY. Fastly Edge SDK. `https://www.fastly.com/products/edge-sdk`. Accessed: 2018-03-14.

[8] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI* (2017), pp. 363–376.

[9] HOME BOX OFFICE. Official website for the HBO series Game of Thrones. `https://www.hbo.com/game-of-thrones`. Accessed: 2018-03-19.

[10] MEIKLEJOHN, C., AND VAN ROY, P. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming* (2015), ACM, pp. 184–195.

[11] MEIKLEJOHN, C. S., ENES, V., YOO, J., BAQUERO, C., VAN ROY, P., AND BIENIUSA, A. Practical evaluation of the lasp programming model at large scale: An experience report. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2017), PPDP '17, ACM, pp. 109–114.

[12] MORTAZAVI, S. H., SALEHE, M., GOMES, C. S., PHILLIPS, C., AND DE LARA, E. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (New York, NY, USA, 2017), SEC '17, ACM, pp. 20:1–20:13.

[13] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.