Capybara: an Edge-Friendly Distributed Object Store for Diverse Serverless Functions

Xin Chen
Georgia Tech
Atlanta, GA, USA
xchen384@gatech.edu

Manoj Prabhakar Paidiparthy
Virginia Tech
Blacksburg, VA, USA
pmanojprabhakar@vt.edu

Chen Qian
UC Santa Cruz
Santa Cruz, CA, USA
cqian12@ucsc.edu

Liting Hu*
UC Santa Cruz
Santa Cruz, CA, USA
liting@ucsc.edu

Abstract

While originally designed for the cloud, the benefits of the serverless paradigm are also vital in Edge/Fog computing environments. This paper presents Capybara, a new scalable, programmable distributed object store for storing and sharing serverless function data objects (state) on edge infrastructures. The key innovations here are (1) achieving scalability and avoiding the significant DRAM cost of indexing metadata servers through a "game-theoretic" DHT-based P2P architecture; (2) providing edge users with a "programmable" handler abstraction to customize data management policies, such as different function image caching policies, warm container "keep-alive" durations, data access control methods, and data replication policies.

We implement Capybara prototype on the Pastry DHT, deploy it on 150 Amazon EC2 nodes, and evaluate it by conducting real-world experiments, demonstrating its significant gains in data locality, application-specific customization, and scalability compared to the state-of-the-art.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; • **Information systems** → **Distributed storage**.

Keywords

Distributed object store, serverless, edge computing.

ACM Reference Format:
Xin Chen, Manoj Prabhakar Paidiparthy, Chen Qian, and Liting Hu. 2025. Capybara: an Edge-Friendly Distributed Object Store for Diverse Serverless Functions. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721462.3730946

*Corresponding author: Liting Hu, Computer Science and Engineering, University of California, Santa Cruz.

Middleware '25, Nashville, TN, USA

ACM ISBN 979-8-4007-1554-9/25/12
https://doi.org/10.1145/3721462.3730946

1 Introduction

Serverless computing is an emerging paradigm, referring to a software architecture where an application is decomposed into 'triggers' (events) and 'actions' (functions), and there is a platform that provides a seamless hosting and execution environment, making it easy to develop, manage, scale, and operate them. While originally designed for the cloud, the benefits of the serverless paradigm are also vital in Edge/Fog computing environments.

Compared to cloud serverless, edge serverless provides compute resources closer to end-users (approximately one or two network hops, i.e., only milliseconds away). This means that edge serverless can significantly reduce service response time, making it ideal for latency-sensitive edge applications. Take AR gaming as an example, a player's interaction with the virtual environment might trigger real-time object recognition, personalized content delivery, dynamic game element generation, and player analytics, all of which could be handled by separate serverless functions running on edge servers, ensuring low latency responses. Another real-life example is Future PLC [23], which uses edge serverless for online advertising. By employing serverless scripting at the edge provided by StackPath [43], Future PLC was able to expedite cookie syncing, resulting in a 30% reduction in costs and a notable enhancement in user experience [18].

Existing serverless platforms [9, 12, 22, 24, 32, 33] mostly rely on cloud storage services to store serverless function data, such as AWS S3 [6] and Google Cloud Storage [25]. These solutions, however, are not well-suited for serverless edge applications for the following reasons:

- They may cause long delays and strain the backhaul network bandwidth. This is especially the case when functions are instantiated for the first time, container images need to be pulled from remote repository pools (e.g., Docker

Hub [16], Amazon ECR [4]), causing expensive data shipping costs and cold starts.

- During execution, serverless functions must write their intermediate results to the cloud, which might be huge. For example, a hash-based shuffle from 10^5 *map* tasks to 10^5 *reduce* tasks leads to 10 billion intermediate files being created instantly on the storage system, which may lead significant slowdown due to the lack of local storage.

In this paper, our goal is to build a scalable, programmable distributed object store for storing and sharing serverless function data objects (state) on edge infrastructures.

The challanges. We face significant challenges due to high diversity and scalability requirements introduced by emerging serverless edge applications.

First, *how do we support application-specific customization for data management?* The cloud storage systems, such as S3 [6], Google Cloud Storage [25], and DynamoDB [3], use a client-server architecture, which relies on a central controller or proxy server to manage client requests and data distribution among storage nodes. Unfortunately, there is a critical lack of application-specific customization for data management. Serverless applications and their function data are treated uniformly, with fixed function image caching policies, warm container "keep-alive" durations, data access control methods, and data replication policies. This is problematic for edge applications because they are quite diverse in terms of popularity, invocation frequency, SLO, and state management, necessitating a customizable object store.

Second, *how can we scale effectively to manage data for a large number of edge applications?* Edge computing presents a unique challenge: edge applications are distributed geographically, generating diverse data objects (state). Similarly, edge nodes are geographically distributed, experiencing hardware heterogeneity and churns as they can freely join or leave the systems. This unpredictability makes it exceptionally challenging to implement effective scaling solutions for the object store.

Our solution. To address the above challenges, we present Capybara, a first-of-its-kind edge-friendly distributed object store that achieves the desired properties for edge serverless: ***full scalability*** and ***data management customization***.

The key innovations of Capybara include: (1) It achieves full scalability through a "game-theoretic" Distributed Hash Table (DHT)-based Peer-to-Peer (P2P) architecture, which addresses critical issues such as load imbalance, avalanche effect, and placing replicas into the same failure domain. (2) It supports data management customization through a "programmable" handler abstraction. Each serverless application has a set of handlers, allowing users to define procedural code that is executed in response to storage operations, such as *read* and *write*. By doing that, users can customize their data management policies (e.g., different function image caching policies, warm container "keep-alive" durations, data access control methods, and data replication policies).

In summary, our contributions are as follows:

- We study the software architecture of existing storage systems and discuss their limitations when applied to the edge settings at scale (§2).
- To the best of our knowledge, we are the first to propose an edge-friendly distributed object store tailored for diverse serverless functions. Our design includes three complementary components: (1) *a "game-theoretic" DHT-based P2P architecture* for dynamic keyspace distribution across heterogeneous edge nodes (§3.2), (2) *a locality-aware distributed key-value store* that optimizes data locality (§3.3), and (3) *a "programmable" handler abstraction* that enables application-specific customization (§3.4).
- We implement Capybara on Pastry DHT [57], deploy it on 150 Amazon EC2 nodes, and evaluate it by conducting real-world experiments, demonstrating its significant gains in data locality, application-specific customization, and scalability compared to the state-of-the-art.

2 Motivation and Background

Serverless applications generate two types of data objects (state): ephemeral and durable. Ephemeral data is temporary and exists only during the lifetime of an application. For example, a MapReduce application's ephemeral data refers to the intermediate results between stages. Durable data, on the other hand, needs to be stored long-term. Examples of the durable data include serverless function's container image metadata, user data, database records, as well as input and output files.

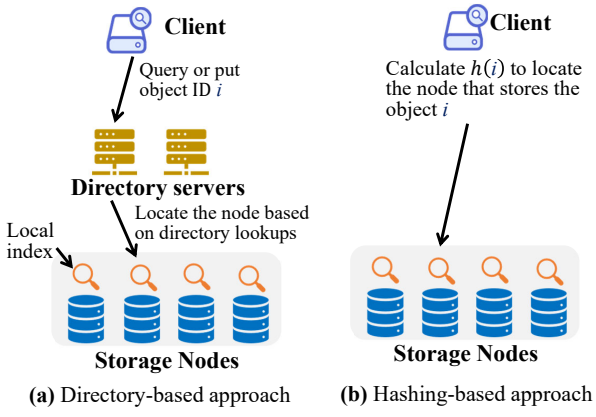


Figure 1: Different object placement and lookup strategies.

Table 1: Comparison of cloud storage systems and Capybara.

Storage System	Storage Type	Lookup Unit	DRAM Cost	Read/Write Latency	Co-locate Function with Data	Application-specific Customization
Amazon S3 [6]	Object storage	Directory	High	High	✗	✗
Google Cloud Storage [25]	Object storage	Directory	High	High	✗	✗
Microsoft Azure Blob Storage [11]	Object storage	Directory	High	High	✗	✗
IBM Cloud Object Storage [29]	Object storage	Directory	High	High	✗	✗
Alibaba Cloud Object Storage [1]	Object storage	Directory	High	High	✗	✗
IndexFS [56]	File system	Directory	High	Medium	✓	✗
InfiniFS [54]	File system	Directory	High	Medium	✓	✗
OpenStack Swift [39]	Object storage	Local index	Low	Medium	✗	✗
CRUSH (Ceph) [13, 63]	Object, block, and file	Local index	Low	Medium	✗	✗
MapX [61]	Object storage	Local index	Low	Medium	✗	✗
CouchDB [7]	Object storage	Local index (B-tree)	Low	Medium	✗	✗
Capybara (this work)	Object storage	"Game-theoretic" distributed hashing	Low	Medium	✓	✓

Existing serverless platforms mostly use cloud storage [3, 6, 25] to store serverless application's data. In such a system, each data object (state) is uniquely identified by a bit string, called an identifier (ID), name, or key.

To manage data objects at a massive scale, there are two typical object placement and lookup strategies.

1. Directory-based approach. As shown in Figure 1(a), this approach stores ID-location mappings in centralized directory servers or metadata servers. Clients receive object locations by querying the server. It has the following limitations when applied to edge serverless: (1) *Centralized bottleneck.* The number of function invocations can be quite high, reaching millions per day. The central directory server may become a bottleneck for handling a large number of requests from clients. (2) *High DRAM cost.* The DRAM resources required to house the directory are significant. For instance, storing 10 billion ID-location mappings requires $> 400GB$, where the majority is used to store IDs, as in practice, the average size of IDs is tens of bytes such as 16 bytes in Ceph [13] and 40 bytes in Twitter [66] or Facebook [44].

2. Local index approach. As shown in Figure 1(b), this approach places data to storage nodes based on the hash value of its ID $h(ID)$ [47, 63]. Each node in the cluster is responsible for a specific range of data based on the key of data.

We show a comparison of cloud storage systems in Table 1. Unfortunately, these systems are not suitable for serverless edge applications, and we identify three main limitations.

- *Limited scalability.* They leverage a client-server architecture, which limits scalability, as they rely on a central controller or proxy server to manage client requests and data distribution among storage nodes.
- *High latency.* They are mostly designed for the cloud service, which may cause high latency, as data is stored in the cloud and may be far away from the edge nodes where functions are invoked.
- *Lack of application-specific customization.* These systems do not offer features or interfaces that are tailored to a specific use case, making it difficult to meet different serverless applications' needs. For example, they use a fixed "keep-alive" policy that stores a function's durable data (e.g., container image) in memory after the function execution (the timeout is 10 to 20 minutes), but do not consider application and function's skewed popularity distributions. They use a fixed isolation mechanism, but do not provide different access control methods for different priority data objects (state).

3 Design

We are not attempting to build a general-purpose distributed object store, such as S3 [6]. Rather, our goal is to support relatively simple operations on data objects (state).

Interestingly, even with simple operations, we can build a powerful edge-friendly distributed object store to realize diverse data management policies for serverless functions.

3.1 Overview

Figure 2 shows the overview of Capybara. It consists of three layers.

Layer 1: "Game-theoretic" DHT-based P2P architecture. By incorporating game-theoretic principles, edge nodes strategically self-organize into a "game-theoretic" DHT-based P2P architecture, autonomously deciding on offering idle storage resources in exchange for rewards, which addresses critical issues such as load imbalance, avalanche effect, and placing replicas into the same failure domain. We assume a single administration unit in the system, which runs on a server to issue one public key certificate to every legitimate node. Authentication between the communication nodes in the system is achieved by exchanging the public keys and their certificates. This method can defend against security attacks in a distributed system such as Sybil attacks.

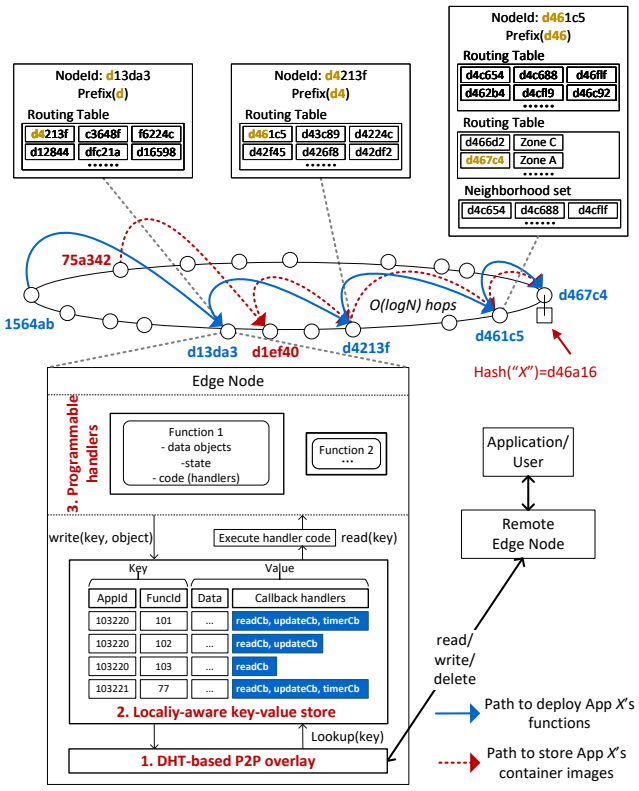


Figure 2: The Capybara system architecture.

Layer 2: Locality-aware distributed key-value store. Built upon Layer 1, we implement a persistent storage utility for storing and sharing function data objects (state). Each function has a unique key, which is computed as the secure hash (SHA-1) of the function's name, the application's name, and a randomly chosen salt. Its data objects (state) are stored in the m nodes whose NodeIds are numerically closest to the key. To retrieve a function's data, the routing substrate typically applies a hash function to the key to compute the IDs of the node that store the associated value.

Layer 3: "Programmable" handler abstraction. In sharp contrast to traditional storage systems that merely handle key-value pairs, our approach innovatively integrates operational code directly with these pairs. This operational code is structured as a set of *programmable handlers*—`readCb`, `updateCb`, `timerCb`—that specify how the application behaves. These handlers enable the system to make intelligent decisions based on the data's access history, the current number of replicas, and even the time of day. For instance, when a client performs a "*read*" operation to access a data object (e.g., a function's container image), the `readCb` handler will be invoked to perform a specific task, such as incrementing a counter for the number of *reads*. If this counter surges rapidly within a short timeframe, the system identifies the container image as "popular". This innovative feature allows the handler to adjust the keep-alive time of the container image dynamically, ensuring it remains in memory longer and significantly reducing cold-start latency.

The key to efficiency comes from several factors. 1. *Scalability and low DRAM cost.* We use a decentralized architecture, allowing data to be stored and retrieved directly between nodes without the need for centralized directory (metadata) servers. 2. *Application-specific customization.* Our system can support various storage lifetimes, access control methods, or state replication policies, in a way that satisfies diverse edge applications' QoS requirements.

3.2 "Game-theoretic" P2P Architecture

We start by organizing distributed edge nodes into a DHT-based P2P ring overlay that implements the value/node mapping. This is similar to the BitTorrent nodes that use the Kademila DHT [55] for "trackerless" torrents. Each edge node is assigned a unique 128-*bit* identifier (NodeId) in a very large circular NodeId space (e.g., $0 \sim 2^{128}$). The DHT-based routing substrate guarantees that, *no matter where the function is invoked, we can find nodes that store its data objects (state) within $O(logN)$ hops, where N is the total number of nodes in the system.*

To do that, each node needs to maintain three data structures: *a routing table, a leaf set*, and *a neighborhood set.*

- *Routing table* is used for object placement and lookup. The routing works based on prefix-based matching. Every node knows m other nodes in the ring and the distance of the nodes it knows increases exponentially. It jumps closer and closer to the destination, like a greedy algorithm, within $\lceil log_{2^b} N - 1 \rceil$ hops, where $2^b - 1$ is the number of entries in the routing table.
- *Leaf set* is used for failure recovery. It assists in rebuilding routing tables when any node fails or leaves the system.
- *Neighborhood set* is used for maintaining the locality properties.

Next, we enable edge nodes to contribute their idle storage resources to establish a distributed object store. However, edge systems present unique challenges. When numerous edge nodes within the same zone contribute excessive storage, they may end up managing a large keyspace in a DHT. This concentration can create a "hotspot" of data, leading to high read and write traffic, akin to an "avalanche effect". Therefore, while it's crucial to incentivize edge nodes to contribute resources through rewards, it's equally important to impose costs that discourage excessive contributions from too many nodes in the same edge zone.

To address this challenge, we propose a game-theoretic framework to model the interactions between edge nodes

and the edge provider as a non-cooperative game, in which each edge node autonomously decides on offering idle storage resources in exchange for rewards (e.g., monetary compensation, reduced data plan costs, or priority access to computing resources) without any centralized control.

Costs. Consider a set N participating edge nodes, where each edge node is indexed by i. Let $x_i \geq 0$ denote the idle storage (the number of equally sized virtual nodes) that an edge node is willing to provide. These decisions form a profile vector $X = [x_1, x_2, \ldots, x_N]$. Each edge node aims to maximize its own utility by determining its resource allocation decision, taking into account costs and rewards. We model this cost as $Cost_i = Q_i x_i^2 + L_i x_i$, where Q_i represents the quadratic cost, and L_i is the linear cost associated with x_i. The quadratic cost function discourages excessive storage allocation. Moreover, considering edge nodes are heterogeneous, we require $0 < x_i < S_i$, where S_i denotes the maximum storage capacity of node i.

Rewards. The edge provider, as a price-maker, has the flexibility to adjust rewards based on market demand. The price for each unit of resource (equally sized virtual node) can be captured by $p(x) = \overline{R} - \varepsilon \sum_{j=1}^{N} x_j$, where $\overline{R}$ is the maximum unit reward set by the provider, and ε is the weighting parameter controlled by the provider, measuring the negative impact of decisions made by other edge nodes. Overall, the reward function is a function of x_i as well as x_{-i} in the set (x_{-i} denotes the joint action of all other nodes except x_i.), which penalizes increases in the sum of x_i across all nodes. We model this reward as $Reward_i = p(x)x_i$.

Optimization objective. Each node seeks to minimize its local objective function J_i, without coordinating with others:

$$J_i = Cost_i - Reward_i = (Q_i x_i^2 + L_i x_i) - (\overline{R} - \varepsilon \sum_{j=1}^{N} x_j)x_i, \forall i.$$

We propose a decentralized algorithm to compute the Nash Equilibrium (NE) point for the proposed game-theoretic model. Each edge node starts with some initial point $x_0^i \in X_i$, and follows a step-size α. At each time slot τ, each edge node exchanges its decision estimate with DHT's neighborhood set nodes, calculates the Metropolis weights [65], and updates its decision accordingly:

$$x_{\tau+1}^i = \prod X_i \left[x_\tau^i - \alpha \bigtriangledown_i J_i \left(x_\tau^i, x_\tau^{-i} \right) \right]$$

3.3 Locality-aware Distributed Key-Value Store

As the second layer, our goal is to achieve data locality by co-locating serverless function data with function invocations, thereby reducing latency.

Algorithm 1 Decentralized algorithm for NE computation

1: Nodes are instructed to use step-size α.
2: Each node i initializes with arbitrary initial vectors.
3: **for** time slot $k \geq 1$, each node i **do**
4: Exchange the decision estimate $z_k^{i,-i}$ with its neighbors;
5: Calculates the weights $[W_k]_{ij}$ using Metropolis weights;
6: Updates local estimate $z_{k+1}^{i,-i}$ and next action x_{k+1}^i by
7: $z_{k+1}^{i,-i} = \sum_{j=1}^{N} [W_k]_{ij} z_k^{j,-i}$,
8: $x_{k+1}^i = \prod x_i \left[\sum_{j=1}^{N} [W_k]_{ij} [z_k^j]_i - \alpha \bigtriangledown_i J_i \left(\sum_{j=1}^{N} [W_k]_{ij} z_k^j \right) \right]$

We create a "bucket" data structure in each edge node and organize them into a locality-aware distributed key-value store. The key idea is co-locating function data with function invocations on edge nodes that minimize the data shipping cost, leveraging the same DHT overlay networks for placing functions and their data objects in $m * O(logN)$ steps regardless of their geographical locations (m is the number of edge zones). As shown in Figure 2, the key-value store is composed of the same edge nodes in Layer 1. Each node can act as a storage node or a user access point for routing *read*/*write* requests and deploying functions. This is in contrast to existing studies [9, 12, 24, 28] that "separately" design the storage component with the scheduling component.

Object Placement and Lookup. When an application joins the system, an application certificate is generated, which assigns the application a unique 160-*bit* key (AppId), e.g., the secure hash (SHA-1) of the application's textual name, the owner's Id, and a random salt. When inserting data, Capybara routes the data to the k nodes whose NodeIds are numerically closest to the 128 most significant bits of the key (k is a user-defined parameter). When retrieving data, Capybara applies the same hash function to the key to compute the NodeIds that store the data object.

Function Deployment. Capybara deploys an application's functions on edge nodes using the same DHT-based routing substrate. The deployment process includes (1) generating a key by calculating the secure hash of the application's key (AppId); (2) routing an "invocation query" toward the key, which specifies function code and triggers; (3) delivering the query to the node whose NodeId is numerically closest to the key, and (4) spawning containers on this node and/or the neighboring nodes. When the workload changes, the neighboring nodes are used for scaling containers.

This procedure ensures (1) the data insertion and function deployment share the same key ($key = hash(\text{“}application\text{”})$), so their routing paths converge at the same destination node. Therefore, the function data is placed close to the node where the function will be invoked, enabling data locality. As shown

Table 2: Capybara programmable handlers (we use data object and state interchangeably in this paper).

Programmable Handler	Description
`createCb(caller)`	Invoked upon the initial creation of a function's data (state), such as during application registration. Returns the data to be stored by the node (e.g., itself or nil).
`readCb(caller, args)`	Invoked when a *read* operation is performed on the data (e.g., when serverless scheduler tries to access function's container info during invocation). Returns the function data. It may modify the data and write it back to the storage system depending on the handler code.
`updateCb(caller, new_data)`	Invoked when updating an existing function data (state). Returns the new data that needs to be stored.
`timerCb()`	Invoked periodically at intervals set by the system. This handler has no return value. It is used to perform periodic tasks such as container cleanup, data replication, and health monitoring.

Table 3: The system API exposed to programmable handlers.

System API	Description
`getCurrentTime()`	Returns system coordinated universal time.
`getNodeId()`	Returns the local node's NodeId.
`getLocalIp()`	Returns the local node's IP.
`deployContainer(key)`	Triggers the container deployment process of the function corresponding to the key.
`deleteContainer(key)`	Triggers the container deletion process of the function corresponding to the key.
`read(key)`	Returns the data and nodes storing copies of the data.
`write(node,key,value)`	Stores the <key, value> pair on the specified node.
`getLeafSet(n)`	Returns set of "n" nodes whose NodeIds are numerically closest to the current node.

in Figure 2, application X's functions and X's container images are placed around the same location; and (2) following with the load balance property of DHTs, the function data are stored in a well-balanced manner.

3.4 "Programmable" Handlers Abstraction

As the third layer, we develop a new programmable handler abstraction. We implement a store controller for executing these handlers. When a request is sent to the key-value store to access the data, the store controller spawns a code runner process to retrieve and execute the relevant handler code.

Table 2 shows the programmable handlers. These handlers link to various object store events, such as *create*, *read*, *update*, and as well as *timer* events executed periodically during an application's lifetime.

Table 3 shows the system API that Capybara exposes to programmable handlers. The system API is the only way for programmable handlers to interact with the Capybara system. These handlers use the system API to obtain information about the local node and trigger basic Capybara functionalities like deploying a container, deleting a container, sending subscribe messages, and replicating a function's data.

3.5 Failure Recovery

Overlay churn. The P2P overlay is self-organizing and self-repairing. When any edge node fails or leaves the system, the overlay can automatically rebuild the routing tables from the failed node's leaf set nodes and reconstruct the P2P ring.

Function recovery. If any function worker (container) fails due to churn, the system starts a new container on a new node selected from the failed node's neighborhood set.

Data recovery. Users can leverage the programmable handler to adjust replication factors and frequencies to implement their own data recovery policy.

4 Use Cases

Here are three use cases demonstrating how various data management policies can be implemented through programmable handlers and the system API.

4.1 Adaptive Keep-Alive Policy

```
var readCb = function (state, args) {
     state.intervals.add(
          sys.currentTime() - state.lastExecTime,
     );
     state.keepAlive =
          dht.currentTime() + max(intervals) * minutes;
     state.coldWait =
          dht.currentTime() + min(intervals) * minutes;
     write(this, sys.getKey(), state);
     return state;
};
var timerCb = function (state) {
     if (sys.currentTime() > state.coldWait) {
          sys.deployContainer();
          reset(state.coldWait);
     }
     if (sys.currentTime() > state.keepAlive) {
          sys.deleteContainer();
          reset(state.keepAlive);
     }
     write(this, sys.getKey(), state);
};
```

Listing 1: Adaptive keep-alive policy.

Listing 1 shows the code snippets of an adaptive keep-alive policy. Here, we introduce two parameters: *the cold-waiting window* and *the keep-alive window*. The cold-waiting window represents the time between the last execution and when the system loads the application image to memory.

The keep-alive window determines how long the application will remain in memory after its image is loaded to memory. A long cold-waiting window reduces resources but may cause cold starts, in case the next invocation occurs sooner than expected. A long keep-alive window reduces cold starts but may waste resources, in case the node is left idle doing nothing. Our goal is to dynamically set cold-waiting window and the keep-alive window. For example, if idle times are consistently short, indicating a high invocation frequency, we will reduce the cold-waiting time, causing the application image to be loaded to memory more frequently. The `readCb` handler is invoked when the function data is read, such as when the system needs to deploy the function container on edge nodes. The `timerCb` handler wakes up periodically, which is responsible for loading the application image to memory after the cold-waiting window, recycling the container after the keep-alive window, and resetting both windows.

4.2 Data Subscription Policy

Our goal is to customize Capybara to enable clients to "subscribe" their interested data and receive update notifications. Some edge applications, such as social media applications that process graphs, need to provide their clients with a continuously fresh, up-to-date view of the parts of the social graph they are interested in. Many systems rely on traditional polling methods, which have devices periodically poll the backend infrastructure for graph updates. However, since many clients at the edge are on mobile devices, frequent polling can easily saturate the bandwidth and edge device battery. Facebook's analysis reveals that 80% of polling queries produce no new data because no updates have occurred since the last poll, thus recommends only publishing selected updates to interested clients [46].

```
var readCb = function (state, args) {
     var client = args[0];
     state.subscribers.add(client);
     return data;
};
var updateCb = function (new_state) {
     for (client in state.subscribers) {
          write(client, sys.getKey(), new_state);
     }
     return data;
};
```

Listing 2: Data subscription policy.

Listing 2 shows the code snippets of this data subscription policy. Upon receiving a read request, the `readCb` handler is invoked, which extracts the client information and sends a subscribe message the client, allowing the client to be notified of subsequent updates to the object. When an update occurs to the object, the `updateCb` handler is invoked, which publishes the update to all subscribed clients. Optionally, we can use a `timerCb` handler to clear the subscriber list for each function periodically.

4.3 Smart Replication Policy

Many storage systems typically employ a fixed replication strategy. For example, in HDFS [59], the default replication factor for all stored values is 3. To enhance flexibility, Capybara can be customized to dynamically adjust the replication factor, the replication interval, and choose nodes on which the data object will be replicated. This flexibility is useful for serverless edge applications with diverse QoS requirements. For example, for a media gaming edge application that values high availability, we can replicate its data (e.g., user profile) to a large number of edge nodes, so that it can serve more client requests simultaneously to reduce response time.

```
var createCb = function (state) {
     var nodes = sys.getLeafSet(5);
     for (var node in nodes) {
          write(node, sys.getKey(), state);
     }
     return state;
};
var readCb = function (state, args) {
     state.count += 1;
     write(this, sys.getKey(), state);
};
var timerCb = function (state) {
     if (state.count > MAX) {
          var node = sys.getNeighbor(1);
          write(node, sys.getKey(), state);
     }
     if (state.count < MIN) {
          sys.delete(state);
          return;
     }
     state.count = 0;
     write(this, sys.getKey(), state);
};
```

Listing 3: Smart replication policy.

Listing 3 shows the code snippets of this smart replication policy. Capybara randomly assigns NodeIds, with a higher likelihood of allocating numerically close NodeIds to geographically distant nodes. Hence in the `createCb` handler, we can replicate nodes to the leaf set of the current node, thereby preventing placing replicas into the same failure domain. Then we monitor the access count in `readCb`, and use `timerCb` to decide whether to increase or decrease the number of replicas on adjacent nodes.

5 Implementation

We implement Capybara on top of the open-source Pastry (v.2.1) [41] and Docker (v.19.03.15) software stacks. Such implementation is motivated by the following considerations: (1) Pastry is an overlay and routing network for implementing DHTs similar to Chord [60] and Kademlia [55]. Rather than developing a new distributed system core, we can take advantage of Pastry's unique feature of proximity-aware routing, self-repairing routing table, message transportation layer, and P2P storage utility. (2) Docker is widely used in serverless systems like OpenWhisk [8], OpenLambda [51], Fission [20], IronFunctions [30], Fn Project [21], Kubeless [33], Knative [32] to containerize and deploy serverless functions. We integrate a Java-based Docker client with Pastry to facilitate the execution of serverless functions. We use the Nashorn script engine [37] to interpret and execute user scripts.

We made the following major implementations: (1) We transformed the previous flat P2P overlay into a game-theoretic DHT-based P2P overlay by integrating a decentralized game-theoretic model for dynamic keyspace distribution (§3.2). (2) We developed a distributed serverless object store (§3.3) that facilitates operations such as registering applications, packaging function code with dependencies as container images, and uploading these images to the store. A corresponding scheduling component was implemented to calculate AppId and FuncId, route requests to nodes with the closest NodeId to the computed hash, spawn containers on the target node and its neighbors for function execution, and store function outputs and logs. (3) We implemented programmable handlers and the system API exposed to these handlers (§3.4). (4) We also realized various data management policies (§4).

6 Evaluation

We deploy Capybara in a real-world distributed environment with 150 Amazon EC2 nodes. First, we evaluate Capybara's data locality and function deployment performance using real-world datasets and industry traces [35]. Second, we evaluate prototyping use cases. We have the following key results.

- Capybara outperforms state-of-the-art serverless systems, enabling attractive data locality property for storing and sharing function data across a wide range of applications and experiments at different scales (§6.2).
- *Use Case 1*: Capybara's adaptive keep-alive policy outperforms state-of-the-art serverless systems' fixed keep-alive policies by reducing the 90_{th} percentile cold start percentage by 42.9% and reducing the 75_{th} percentile idle time percentage by 14.7% (§6.3).
- *Use Case 2*: Capybara's data subscription policy outperforms the traditional polling method by reducing the 90_{th} percentile tail latency by 83.7% (§6.4).
- *Use Case 3*: Capybara's smart replication policy dynamically adjusts the replication factor and replication intervals based on different "popularities" of data objects (§6.5).
- Capybara efficiently adapts to the edge node heterogeneity and network variance (§6.6).

6.1 Methodology

Experimental setup. Capybara is designed to operate on large-scale edge topologies consisting of hundreds of thousands of edge nodes with a diverse combination of computing and storage capabilities. However, such a deployment is prohibitively expensive and impractical. As such, we resort to a cluster of 150 Amazon EC2 nodes, each of which has 4 vCPUs, 16GB of RAM, and 32 GB of disk space (equivalent to Cisco's IoT gateway [14]). We create a real-world heterogeneous edge environment. We launch 5000 heterogeneous edge nodes on the testbed. Each node could randomly host up to 4, 16, 64, or 256 functions.

Baseline. We choose OpenWhisk [40] and OpenLambda [38] as the baseline for evaluating Capybara's data locality and function deployment performance. We use helm [27], a Kubernetes package manager, to bring up OpenWhisk on the Kubernetes cluster. We use Pastry 2.1 [41] for DHT's implementation which is configured with a leaf set of 24, max open sockets of 5000, and a transport buffer size of 6 MB.

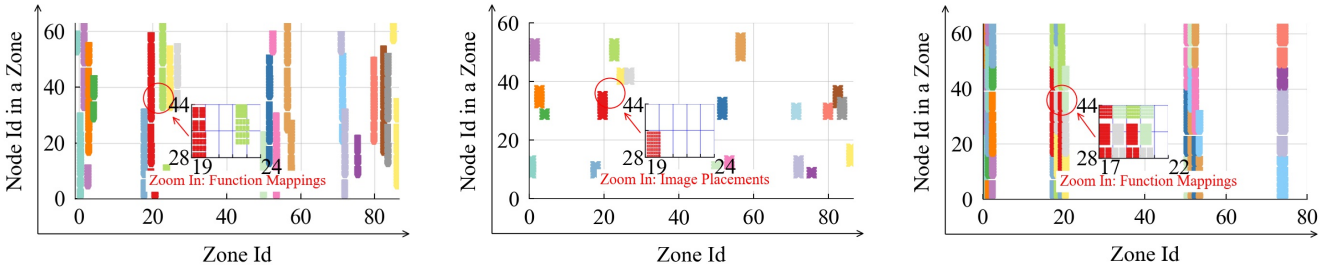
Industry function traces. We use Microsoft Azure traces [35] to simulate a real-world workload to evaluate Capybara's customized policies. The dataset consists of function invocation count, function execution time, application's memory usage, and the type of function trigger.

Real-world applications. We use real-world serverless applications from open-source benchmark suites ServerlessBench [67] and VSP Benchmark [45].

- *ALU*: A CPU-intensive application that spawns multiple threads to perform large arithmetic calculations in a loop with a random number of iterations.
- *Image Resizing*: An application that resizes a given image to three target sizes and a thumbnail.
- *Video Encoder*: An application based on FFmpeg [19], a utility for processing video and audio files.

6.2 Data Locality Performance

We evaluate Capybara's data locality performance and compare it with state-of-the-art serverless systems.



(a) DHT mappings: deploying $20k$ functions (20 applications) onto 5000 heterogeneous nodes.

(b) Capybara achieves data locality by placing the container images close to the function invocations.

(c) Bin pack mappings: deploying $20k$ functions (20 applications) onto 5000 heterogeneous nodes.

Figure 3: Capybara's data locality performance.

Details about the function deployment methods implemented by commercial serverless platforms have not been made public; nevertheless, researchers have revealed some facts through reverse engineering. For example, the investigations performed in [62] and [53] identify that AWS Lambda [9] greedily packs containers on virtual machines (VMs) to improve resource utilization. It maintains a global state of all the workers/invokers and schedules a function invocation on a worker that satisfies the application's resource constraints. However, this bin packing method may perform poorly in balancing the load across edge nodes. Another disadvantage is the lack of data locality. The function data is stored in remote storage, such as CouchDB [7] for OpenWhisk and Amazon S3 [6] for AWS Lambda. This necessitates pulling container images from remote storage which increases the total deployment time.

Figure 3a shows Capybara's snapshot of function worker (container) to node mappings on large-scale edge topologies. The X-axis represents the zones in order in the edge system. The Y-axis represents the edge nodes in order within a zone. The crossing points are functions' mapped locations. Different colors represent different applications. Each application has a different number of functions ranging from 256 to 2048. We can see that functions belonging to the same applications are deployed geographically close to each other, and functions belonging to different applications are evenly distributed over edge topologies. This is because different applications have different keys, and thus fall into different hash buckets and map to different nodes.

Figure 3b shows Capybara's snapshot of container images on node mappings. By comparing Figure 3a with Figure 3b, we can see that each container image is placed close to the node where its function is invoked, enabling data locality. This is because Capybara's locality-aware distributed object store uses DHT-based routings to place the container images, function data, and results. Function deployment and data insertion use the same key, so their routing paths converge at the same destination node.

As a comparison, Figure 3c shows the bin packing method's snapshot. This greedy method implemented by this centralized approach tries to schedule the function on the first available edge node that satisfies the resource constraints. It makes decisions based on the information at hand without considering the possible impact these decisions may have in the future. For example, as shown in Figure 3c, functions of the same color have to travel long paths to communicate with each other during data shuffling, which may significantly impact performance.

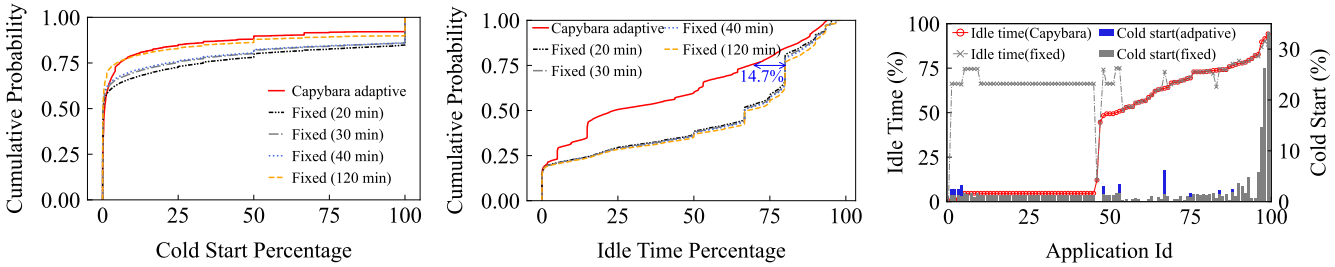
6.3 Measuring Adaptive Keep-Alive Policy

We customize Capybara to implement the adaptive keep-alive policy in Listing 1, driven with real workload traces.

We use Microsoft Azure traces [35], containing a daily collection of function invocations between July 15 and July 28, 2019, sampled every minute for three days. The traces include 15,940 Applications that consist of 39,491 functions in total. We compare Capybara's adaptive keep-alive policy with state-of-the-art serverless systems' fixed policies (20 minutes, 30 minutes, 40 minutes, and 120 minutes).

- *Cold Start (%)*: Percentage of applications resulting in a cold start.
- *Idle Time (%)*: Percentage of time that the spawned application's containers remain idle without any function invocation (measured at 1 minute granularity).

Figure 4a shows the comparison of the percentage of cold start out of the total number of invocations for different policies. Results show that Capybara's adaptive keep-alive policy reduces the cold start percentage at the 90_{th} percentile by 42.9%, as compared to the fixed keep-alive policy (120 minutes). The improvement is more noticeable for shorter fixed keep-alive time (20 minutes, 30 minutes, 40 minutes).

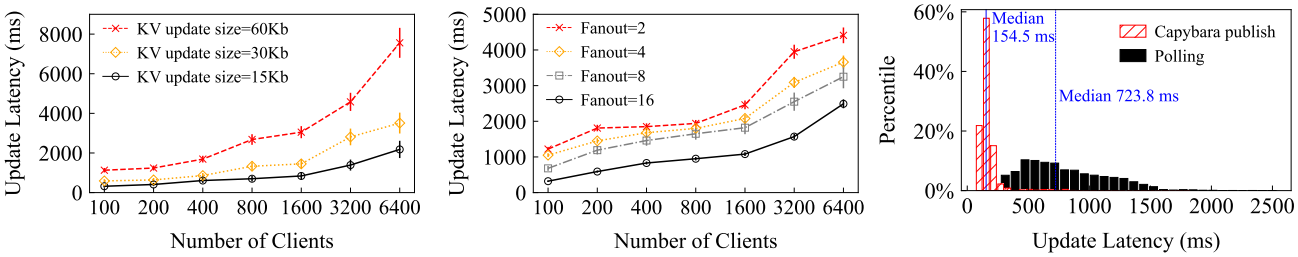


(a) CDF distribution of function instance's cold starts for 15k Applications.

(b) CDF distribution of function instance's idle times for 15k Applications.

(c) Cold start and idle time trade-off for 100 Applications.

Figure 4: Adaptive keep-alive policy.



(a) Update latency for different update sizes.

(b) Update latency for different tree fanouts.

(c) Comparison of publish and polling.

Figure 5: Data subscription policy.

Figure 4b shows the comparison of the percentage of container idle times for different policies. Results show that Capybara's adaptive keep-alive policy reduces the container idle time percentage at the 90_{th} percentile by at least 3.2%~4.3% and 75_{th} percentile by 14.7%, as compared to the fixed keep-alive policies.

Figure 4c shows a trade-off analysis between idle time and cold starts for 100 randomly selected individual applications. Intuitively, if a policy reduces cold starts, it tends to keep container images in memory longer, leading to increased idle time and resource wastage. Therefore, an effective policy should strike a balance between idle time and cold starts. Results show that Capybara's adaptive keep-alive policy has less idle time with comparable cold starts, as compared to a fixed keep-alive policy (20 minutes), and is thus more resource-efficient.

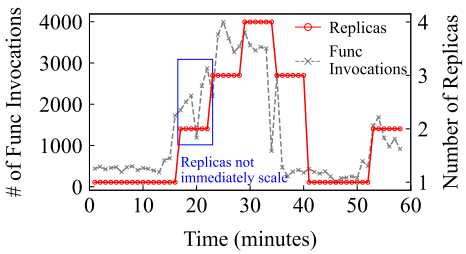
6.4 Measuring Data Subscription Policy

We customize Capybara to implement the data subscription policy in Listing 2 for updating clients with their interested value updates.

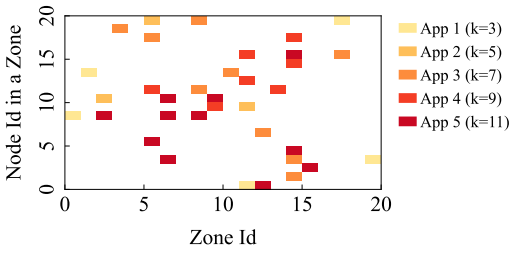
Figure 5a shows the increase in the update latency with an exponential increase in the number of clients subscribed to updates of key-value pairs for different update sizes. The X-axis represents the number of clients, and the Y-axis represents the update latency (i.e., the delay between the occurrence of a data update and the time all clients are notified of the update). We can see that as the number of clients increases *exponentially* (from 100 to 6400), the update latency only increases *linearly*. This is because the update latency is limited by Pastry's publish/subscribe tree depth ($O(logN)$) using the DHT-based routing substrate. As a result, even with millions of clients, Capybara guarantees that only a few extra hops are needed for the clients to receive their interested data update, demonstrating Capybara's scalability.

Figure 5b shows the increase in the update latency with an exponential increase in the number of clients for different tree fanouts. We can see that a larger fanout reduces the update latency because it has less tree depth. However, a larger fanout tree isn't always ideal (less resilient to failures): if an internal node fails or leaves, the new node must rebuild many connections for affected branches to reconstruct the tree.

Figure 5c compares the update latency of Capybara's publish method and the traditional polling method [46]. Results show that Capybara's publish method achieves a median update latency of 154.5 *ms*, whereas the polling method exhibits a median update latency of 723.8 *ms*. The publish method reduces the 90_{th} percentile tail latency by 83.7% compared to polling. Frequent polling may easily burden the query processor and potentially saturate bandwidth. In contrast, the data subscription policy ensures that updates are sent to clients only when there is a change in the value.



(a) Capybara dynamically scales replicas to accommodate the varying workloads.



(b) Geographical distribution of replicas for various applications (k represents the replication factor).

Figure 6: Smart replication policy.

6.5 Measuring Smart Replication Policy

We customize Capybara to implement the smart replication policy in Listing 3 for managing function data, evaluated using real-world applications.

Figure 6a shows how the number of replicas is dynamically changed to accommodate the varying workload. The X-axis represents the time range. The Y-axis represents the function data access rate per minute and the corresponding number of replicas. We can see that, upon detecting a change in the workload, Capybara does not immediately replicate the function data. Instead, it gradually replicates it to ensure stability and responsiveness to fluctuations in the access rate. This controlled adaptation enables Capybara to efficiently manage resources without compromising performance.

Figure 6b shows the distribution of replicas for various applications across a large-scale edge topology, comprising 400 nodes in 20 edge zones. Different applications are configured with varying numbers of replicas, and the results demonstrate an even distribution of all replicas across different edge zones. The DHT-based routing substrate ensures that the replicas placed on the nodes with numerically closest NodeIds are geographically distributed, thereby enhancing resiliency against regional failures.

6.6 Adaptivity Analysis

Figure 7 illustrates the adjustment of network I/O traffic to avoid data hotspots before and after applying the proposed decentralized game-theoretic model. We present the results

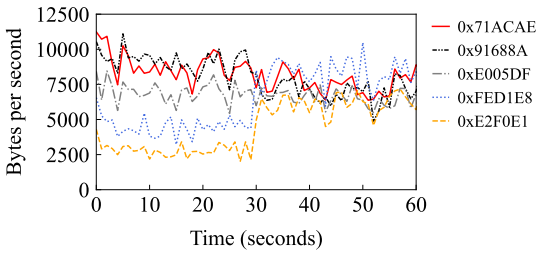


Figure 7: Network traffic adjustment of virtual edge nodes.

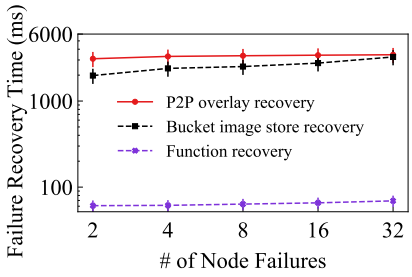


Figure 8: Failure recovery in case of simultaneous failures.

for five representative virtual edge nodes. Initially, due to the heterogeneity of edge nodes and network variability, the resource allocation decisions made by each edge node result in different *read*/*write* requests for each virtual node. This variation leads to load imbalance and suboptimal performance. To address this issue, our game-theoretic model enables each edge server to dynamically adjust the number of virtual nodes it manages. The results demonstrate that our approach effectively ensures load balance, preventing data hotspots and the associated avalanche effect.

Figure 8 shows the overlay recovery time and the function recovery time when multiple edge nodes fail or leave the systems simultaneously. The failure recovery time includes rebuilding the routing table entries, reconstructing the overlay, restarting function containers, and reconstructing the function images from the failed node's neighborhood set nodes. Results show that our system achieves a stable recovery time for an increasing number of simultaneous failures.

6.7 Overhead Analysis

Table 4 shows the system overhead of running Capybara for different prototyping use cases. We provide a detailed breakdown of the maximum execution time associated with each callback function for every use case. We have observed that a significant portion of latency is attributed to the run-time interpretation of the script by the Nashorn script engine [37]. This latency can be further reduced by using pre-compiled binaries for storing user code.

Table 4: Resource consumption comparison.

Use Case	Code Size	Callback Function	Execution Time
Adaptive Keep-alive	1.81K	readCb	152 ms
		timerCb	31 ms
Data Subscription	1.25K	readCb	22 ms
		timerCb	112 ms
Dynamic Replication	0.28K	createCb	77 ms
		readCb	2 ms
		timerCb	34 ms

7 Related Work

Object store acts like a dictionary: the operations include looking up an Id and finding its object. Our work focuses on the placement, lookup, and customization methods of the object store for edge serverless functions.

Cloud Object Storage Systems. Many cloud storage systems [1, 6, 11, 25, 26, 29] use *central directories* to store Id-to-location mappings, where "location" is the network address of a storage node. For example, Amazon S3 [6] allows users to put their objects in a designated bucket and maintains the full object-to-bucket mappings in metadata servers. Lustre [49] replies on metadata servers to tell object locations. HDFS [26] relies on the NameNode server to tell block locations. However, these systems are not well-suited for serverless edge applications because (1) the DRAM resources that need to be spent to house the directory are significant, and (2) it is hard to replicate the huge directory on resource-constrained edge nodes to serve a large number of concurrently running edge applications.

Distributed Object Storage Systems. To address the scalability bottleneck in central directories, many object stores use *hashing* to determine the object locations. For example, OpenStack Swift [39] uses a consistent hashing ring to determine the object locations. CRUSH [63] in Ceph [13] uses a strong multi-input integer hash function to map data objects to storage devices. These systems are mainly designed for cloud datacenters characterized by homogeneous storage nodes and abundant bandwidth. This makes them less optimal for edge environments characterized by heterogeneous storage nodes and unreliable networks. Furthermore, they lack application-specific customization for serverless edge applications.

Extensible Object Storage Systems. The concept of extensible systems has been widely explored in various domains. For example, active networks [64] allow users to inject customized programs into network nodes (e.g., routers) and execute the code when nodes are traversed by packets. Extensible operating systems [48, 58] allow applications to change the operating system's interface and implementation. Comet [50] is a distributed key-value store that allows applications to extend its functionality using Lua [34] sandboxed extensions. Similarly, Splinter [52] is an in-memory key-value store that allows clients to write their own functions in Rust [42] for personalized storage operations. Shredder [68] allows customers to embed small storage functions in JavaScript or WebAssembly within the store. However, these systems often rely on specific support such as kernel-bypass networking and V8 intermediate representation, which may limit their scalability. Splinter [52] was evaluated on two machines—one client node and one server node. Shredder [68] was evaluated on three machines—two client nodes and one server node. Capybara complements them by offering a scalable integrated serverless scheduling and storage solution and a game-theoretic model.

Serverless Computing Platforms. While not directly related to object storage, the serverless paradigm influences the design considerations for serverless edge storage solutions. There are many commercial serverless platforms for the cloud (e.g., AWS lambda [9], Azure functions [12], Google Cloud Functions [24], Function Compute [22], Cloudflare Workers [15], Kinvey [31]), and their edge support (Lamba@Edge [5], Azure IoT Edge [36], AWS IoT Greengrass [2]). AWS Snowball Edge [10] is a hybrid cloud-edge storage solution. EdgeFS [17] is a distributed file system running on top of Kubernetes, solving multi-cloud data access. These systems were mainly designed for cloud datacenters. Capybara complements them by offering data management customization support and a game-theoretic model.

8 Conclusion

Existing object storage systems are mainly designed for the cloud, making them not a good fit for the edge context. Capybara is a new scalable and programmable distributed object store designed for serverless edge applications. The design includes three complementary components: a "game-theoretic" DHT-based P2P architecture for dynamic keyspace distribution across edge nodes, a locality-aware distributed key-value store that optimizes data locality, and a programmable handler abstraction that realizes diverse data management policies. We evaluate Capybara on 150 Amazon EC2 nodes by using real-world serverless applications and industry function traces at different scales, demonstrating its significant gains in data locality, application-specific customization, and scalability compared to the state-of-the-art.

Acknowledgment

We sincerely thank the anonymous reviewers and our shepherd, Paulo Ferreira, for their invaluable feedback. This work was supported by the National Science Foundation under Grants NSF CAREER-2313737, NSF OAC-2313738, and NSF CNS-2322919.

References

[1] Alibaba Cloud Object Storage Service (OSS). https://www.alibabacloud.com/product/object-storage-service.

[2] Amazon AWS Greengrass. https://aws.amazon.com/greengrass.

[3] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[4] Amazon Elastic Container Registry. https://aws.amazon.com/ecr/.

[5] Amazon Lambda@Edge. https://aws.amazon.com/lambda/edge.

[6] Amazon S3. https://aws.amazon.com/s3/.

[7] Apache CouchDB. http://couchdb.apache.org/.

[8] Apache OpenWhisk. https://openwhisk.apache.org/.

[9] AWS Lambda. https://aws.amazon.com/lambda/.

[10] AWS Snowball Edge. https://docs.aws.amazon.com/snowball/latest/developer-guide/whatisedge.html.

[11] Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs.

[12] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[13] Ceph. https://docs.ceph.com.

[14] Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[15] Cloudflare Workers. https://workers.cloudflare.com/.

[16] Docker Hub Container Image Library. https://hub.docker.com/.

[17] EdgeFS - a multi-cloud scalable distributed storage system. https://github.com/continuul/edgefs.

[18] Example of edge serverless. https://www.stackpath.com/edge-academy/what-is-edge-serverless/.

[19] FFmpeg. https://ffmpeg.org/.

[20] Fission. https://fission.io/.

[21] Fn Project. https://fnproject.io/.

[22] Function Compute, Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[23] Future PLC. https://www.futureplc.com/.

[24] Google Cloud Functions. https://cloud.google.com/functions.

[25] Google Cloud Storage. https://cloud.google.com/.

[26] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[27] Helm. https://helm.sh/.

[28] IBM Cloud Functions. https://cloud.ibm.com/functions/.

[29] IBM Cloud Object Storage. https://www.ibm.com/cloud/object-storage.

[30] IronFunctions. https://open.iron.io/.

[31] Kinvey. https://community.progress.com/s/products/kinvey.

[32] Knative. https://knative.dev/.

[33] Kubeless. https://kubeless.io/.

[34] Lua – an extensible extension language. https://www.lua.org/spe.html.

[35] Microsoft Azure Function Traces. https://github.com/Azure/AzurePublicDataset.

[36] Microsoft Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge.

[37] Nashorn Engine. https://github.com/openjdk/nashorn.

[38] OpenLambda Source Code. https://github.com/open-lambda/open-lambda.

[39] OpenStack Swift. https://github.com/openstack/swift.

[40] OpenWhisk Source Code. https://github.com/apache/openwhisk.

[41] Pastry. https://www.freepastry.org/.

[42] Rust Programming Language. https://www.rust-lang.org/.

[43] Stackpath. https://www.stackpath.com/.

[44] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[45] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. On Merits and Viability of Multi-Cloud Serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 600–608, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Jeff Barber, Ximing Yu, Laney Kuenzel Zamore, Jerry Lin, Vahid Jazayeri, Shie Erlich, Tony Savor, and Michael Stumm. Bladerunner: Stream processing at scale for a live view of backend data mutations at the edge. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 708–723, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[48] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995.

[49] Peter Braam. The Lustre Storage Architecture, 2019.

[50] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 323–336, USA, 2010. USENIX Association.

[51] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '16, page 33–39, USA, 2016. USENIX Association.

[52] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 627–643, USA, 2018. USENIX Association.

[53] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169, 2018.

[54] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[55] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, page 53–65, Berlin, Heidelberg, 2002. Springer-Verlag.

[56] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 237–248. IEEE Press, 2014.

[57] Antony I T Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[58] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[59] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[60] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[61] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 1–11, Santa Clara, CA, February 2020. USENIX Association.

[62] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[63] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006.

[64] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 64–79, New York, NY, USA, 1999. Association for Computing Machinery.

[65] L. Xiao, Stephen Boyd, and Sanjay Lall. Distributed Average Consensus with Time-Varying Metropolis Weights. *Automatica*, 01 2006.

[66] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[67] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.