CloudPath: A Multi-Tier Cloud Computing Framework

Seyed Hossein Mortazavi
University of Toronto
mortazavi@cs.toronto.edu

Mohammad Salehe
IEEE Student member
salehe@ieee.org

Carolina Simoes Gomes
University of Toronto
cgomes@cs.toronto.edu

Caleb Phillips
University of Toronto
caleb@cs.toronto.edu

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

ABSTRACT

Path computing is a new paradigm that generalizes the edge computing vision into a multi-tier cloud architecture deployed over the geographic span of the network. Path computing supports scalable and localized processing by providing storage and computation along a succession of datacenters of increasing sizes, positioned between the client device and the traditional wide-area cloud datacenter. CloudPath is a platform that implements the path computing paradigm. CloudPath consists of an execution environment that enables the dynamic installation of light-weight stateless event handlers, and a distributed eventual consistent storage system that replicates application data on-demand. CloudPath handlers are small, allowing them to be rapidly instantiated on demand on any server that runs the CloudPath execution framework. In turn, CloudPath automatically migrates application data across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Mobile Edge Computing, Cloud Computing, Path Computing, Computer Systems Organization

ACM Reference format:
Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. 2017. CloudPath: A Multi-Tier Cloud Computing Framework. In *Proceedings of SEC '17, San Jose / Silicon Valley, CA, USA, October 12–14, 2017,* 13 pages.
DOI: 10.1145/3132211.3134464

1 INTRODUCTION

Current mobile networks are not able to support next generation applications that require low latency, or that produce large volumes of data that can overwhelm the network infrastructure in a carrier network. Examples include intelligent personal assistants, medical patient monitoring [21], and safety-critical applications, such as face recognition applications for airport security [35] and intelligent transportation systems [38]. The use of servers on the wide-area cloud is also not an option due to the same low-latency requirements. To address these challenges, the research community and the telecommunications industry are exploring ways to add computation and storage capabilities to the edge of the network. These approaches, variously referred to as cloudlets [33], micro datacenters [8], or fog [12], augment the traditional cloud architecture with an additional layer of servers that are located closer to the end user, typically one-hop away.

This paper introduces *path computing*, a generalization of edge computing into a multi-tier cloud paradigm that supports processing and storage on a progression of datacenters deployed over the geographic span of a network. Figure 1 illustrates how path computing extends the traditional cloud architecture. At the top and bottom of the figure are the traditional wide-area cloud datacenter and the end-user devices, respectively. Path computing enables the deployment of a multi-level hierarchy of datacenters along the path that traffic follows between these two end points. Path computing makes possible different classes of applications, including workloads that aggregate data (such as IoT applications), or services that cache data and process information at different layers. Path computing provides application developers the flexibility to place their serve functionality at the locale that best meets their requirements in terms of cost, latency, resource availability and geographic coverage.

We described CloudPath, a new platform that implements the path computing paradigm and supports the execution of third-party applications along a progression of datacenters positioned along the network path between the end device (e.g., smartphone, IoT appliance) and the traditional wide-area cloud datacenter. CloudPath minimizes the complexity of developing and deploying path computing applications by preserving the familiar RESTful development model that has made cloud applications so successful. CloudPath is based on the key observation that RESTful *stateless* functionality decomposition is made possible by the existence of a common storage layer. CloudPath simplifies the development and deployment of path computing applications by extending the common storage abstraction to a hierarchy of datacenters deployed over the geographical span of the network.

CloudPath applications consist of a collection of short-lived and stateless functions that can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. Developers determine where their code will run by tagging their application's functions with labels that reflect the topology of the network (e.g. edge, core, cloud) or performance requirements, such as latency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SEC '17, San Jose / Silicon Valley, CA, USA
© 2017 ACM. 978-1-4503-5087-7/17/10…$15.00
DOI: 10.1145/3132211.3134464

bounds (e.g. place handler within 10ms of mobile users). CloudPath provides a distributed eventual consistent storage service that functions use to read and store state through well-defined interfaces. CloudPath's storage service automatically replicated application state on-demand across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

We evaluated the performance of CloudPath on an emulated multi-tier deployment. Our results show that CloudPath can deploy applications in less than 4.1 seconds, has routing overhead bellow 1*ms*, and has negligible read and write overhead for locally replicated data. Moreover, our test applications experienced reductions in response time of up to 10X when running on CloudPath compared to alternative implementations running on a wide-area cloud datacenter.

The rest of this paper is organized as follows. Section 2 introduces *path computing*, a generalization of the edge computing design into a multi-tier cloud architecture that supports processing and storage on a progression of datacenters deployed over the geographic span of a network. Section 3 introduces CloudPath, a new platform that implements the path computing paradigm. Section 4 describes the design and implementation of our CloudPath prototype. Sections 5 and 6 present our experimental setup and the results from our evaluation. Section 7 describes related work. Finally, Section 8 concludes the paper and discusses future work.

2 PATH COMPUTING

Edge computing expands the traditional flat cloud architecture into a two-tier topology that enables computation and storage at a locale close to the end user or client device. We introduce *path computing*, a generalization of this design into a multi-tier cloud architecture that supports processing and storage on a progression of datacenters deployed over the geographic span of a network.

Figure 1 illustrates how path computing extends the traditional cloud architecture. At the top and bottom of the figure are the traditional wide-area cloud datacenter and the end-user devices, respectively. The figure also shows the path that traffic between these two end points follows over a collection of intermediate network links and routers. Path computing enables the deployment of a multi-level hierarchy of datacenters along this path, with the traditional wide-area datacenters at the root of the hierarchy.

We refer to a datacenter that is part of the hierarchy as a *node*. Nodes along the hierarchy can differ vastly in the amount of resources at their disposal, with storage and execution capacity expected to decrease as we descend levels in the hierarchy and move closer to the end-user device. Wide-area nodes are assumed to have access to virtually limitless computation and storage; in contrast, nodes close to the edge of the network may have just a handful of servers at their disposal. The number of nodes at any given level of the hierarchy is expected to grow dramatically as we get farther from the root. For example, a path computing deployment may consists of a handful of wide-area nodes, tens of nodes running at the network-core of various mobile carriers, hundred of nodes running on region-level aggregation switches, and tens of thousands of nodes running on the edge of the network.

Path computing can be materialized in a variety of different topologies and networking technologies. For example, a simple

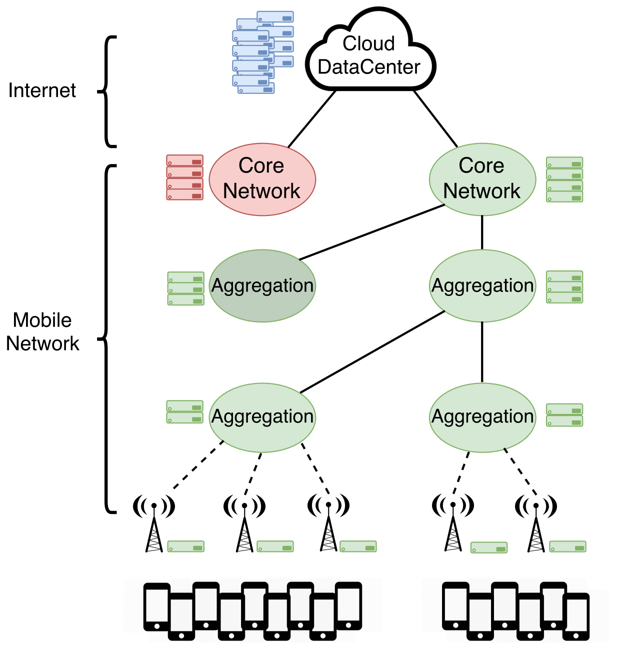

Figure 1: Path Computing Architecture: Path computing provides storage and computation along a succession of datacenters of increasing sizes, positioned between the client device and the traditional wide-area cloud datacenter.

two-tier topology that is the focus of most edge computing research, could consist of a layer of nodes running on or close to WiFi access points and a cloud layer. This simple topology could be expanded to include additional tiers inside the Internet Service Provider's network at convenient aggregation points, at the city and regional levels. Similarly, the architecture could be incorporated into mobile cellular networks. LTE networks by default encapsulate packets and send them to the network's core for processing; however, a growing number of product offerings, such as Nokia RACS gateway [24] and Huawei's Service Anchor [4], have the potential to enable in-network processing by selectively diverting packets for processing. Looking ahead, 5G, which is currently in the process of being standardized, opens the possibility for packet processing at the edge. CloudPath nodes could be incorporated on the base station (EnodeB) or the Centralized Radio Access Network (C-RAN) [1], as well as at aggregation switches along the path to the network core and at the core itself.

2.1 Opportunities and Challenges

Path computing creates new opportunities for application developers. Today, mobile and IoT applications are typically developed based on the client-server model, which requires developers to partition application logic and state between a client running on the end-user device and a server located on the wide-area cloud. In

[1] C-RAN is a proposed architecture for future cellular networks that connects a large number of distributed low cost remote radio heads (RRH) to a centralized pool of baseband units (BBU) over optical fiber links [31].

contrast, path computing provides developers the opportunity to run their server-side functionality on a number of different locations making possible different classes of applications, including workloads that aggregate data (such as IoT applications), or services that cache data and process information at different layers. Path computing provides applications developers the flexibility to control the placement of their application components or tasks at the locations that best meet their requirements in terms of cost, latency, resource availability and geographic coverage.

It is generally accepted that the cost of computation and storage is inversely proportional to datacenter size [7]; therefore, it is reasonable to assume that the unit cost of deploying and managing computation and storage increases as we get closer to the edge and nodes become smaller and more numerous. Conversely, the network cost of serving a request goes down as we move closer to the edge and fewer links need to be traversed. To a first approximation, the cost of running a compute intensive task can be optimized by placing it on the datacenter node that is farthest away from the edge, but still meets the latency and hardware requirements of the task. On the other hand, the cost of a network intensive task can be optimized by running it on the datacenter node that is closest to the edge, while still meeting the task's hardware requirements (i.e., availability of a particular accelerator).

Optimal task placement may also depend on other factors such as the geographic coverage provided by a datacenter node, the size of the population it serves, and user mobility patterns. For example, the effectiveness of a data reduction tasks, such as computing an average over streams of sensor data produced by a farm of IoT devices, is a complex product of the number of available incoming streams, the aggregation factor, and the cost of the computation and network bandwidth. On one hand, the network benefits of aggregation decrease as we get father away from the edge. On the other, the geographic coverage area served by a datacenter node grows as we get away from the edge creating more opportunities for data aggregation (i.e., there are more streams). Similarly, task placement affects how an application component experiences user mobility. For example, an application component running on a city-level node will experience a much lower level of user handover than one deployed on a node closer to the edge, such as WiFi access point.

Unfortunately, taking advantage of the added flexibility introduced by path computing is not easy. It requires developers to partition their server-side functionality, and manage the placement of code and data based on complex calculations that trade off proximity to the user with resource availability and cost. In addition, the limited capacity of the datacenters on the lower levels of the hierarchy puts a hard bound on the number of applications and datasets that can be hosted simultaneously requiring application code and data to be dynamically provisioned. Section 3 introduces CloudPath, a new platform designed to address these challenges.

2.2 Practical Considerations

Path computing datacenters need to be in or near the network of different ISPs or mobile network providers, so it is likely that they will be owned by the different network providers. In contrast, application developers are used to a deployment model where their application is globally available independently of the carrier used by an individual user [2]. Rather than having individual application developers negotiate service agreements with a myriad of network providers, it is likely that cloud providers (existing or new) will offer a one-stop shop that lets application developers run their code across datacenters managed by different carriers. This model follows the approach taken by content delivery network companies, such as Akamai, which let applications owners serve their content to users across different ISPs.

```
face_detection_and_recognition_service {
    login(credential)->token           : any
    detect_faces(image)->coordinates[]  : 10 ms
    recognize_face(image)->label        : 50 ms
}
```

Figure 2: Server API with application entry points labeled with latency requirements.

3 CLOUDPATH

CloudPath is a platform that implements the path computing paradigm, and supports the development and deployment of applications that run on a set of datacenters embedded over the geographical span of the network. CloudPath assumes a subscription model similar to that of existing wide-area network cloud platforms where anyone with an account on the system can deploy and run applications. In this scenario, the available applications and their data vastly outnumber the resources available at the smaller datacenters, which only have enough resources to run a limited number of applications at any time and can store only a fraction of the data. As a result, CloudPath deploys applications and replicates data on-demand.

CloudPath minimizes the complexity for developing path computing applications by preserving, as much as possible, the familiar development model that has made traditional cloud applications so successful. CloudPath builds on the observation that it is accepted practice for cloud applications to implement server-side functionality as services that are exposed to the client over an API consisting of stateless *entry points*, or functions, that are exposed as unique URIs. For example, Figure 2 shows a simplified server-side API for an application that performs face detection and recognition. The API includes three entry points that let the client device login and authenticate, upload an image on which to perform face detection, and upload an image of a face for recognition. The stateless nature of the entry points improves application modularity, makes it possible to dynamically scale each function independently, and increases fault tolerance.

Our key observation is that this functionality decomposition is made possible by the existence of a common storage layer. CloudPath simplifies the development and deployment of path computing applications by enforcing a clear separation between computation

[2]Some carriers deploy applications that are only available to their customers on their own network, but this is a much less attractive deployment model for third-party applications.

and state, and expanding the common storage abstraction to a hierarchy of datacenters deployed over the geographical span of the network.

CloudPath applications consist of a collection of short-lived and stateless functions that leverage a distributed storage service that provides transparent access to application data. CloudPath functions are implemented using high level languages, such as Java or Python. Since CloudPath functions are small and stateless, they can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. CloudPath provides a distributed eventual consistent storage service that functions can use to read and store state through well-defined interfaces. CloudPath automatically migrates application state across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

Developers determine where their code will run by tagging their application's entry points (i.e., functions) with labels that reflect the topology of the network (e.g. edge, core, cloud) or performance requirements, such as latency bounds (e.g. place handler within 10ms of mobile users). For example, Figure 2 shows annotations that indicate that the authentication can run on any datacenter, whereas face detection and recognition need to run in a datacenter that can be reached within 10ms and 50ms of the mobile client, respectively.

CloudPath does not migrate a running function between datacenters. Instead, CloudPath supports code mobility by terminating an existing instance (optionally waiting for the current request to finish) and starting a new instance at the desired location. Similarly, for mobile users, network handoff between cells may result in a change in the network path with traffic flowing through a different set of CloudPath datacenters. CloudPath does not migrate network connections between datacenters. Instead, it terminates existing connections and leaves it to the application to establish a new connection with the new datacenter. While this approach requires application modifications, CloudPath provides a client library that automates the re-connection process.

The next section describes the design and implementation of CloudPath in detail.

4 DESIGN AND IMPLEMENTATION

CloudPath organizes datacenters into a simple tree topology overlaid over a collection of underlying mobile networks and the public Internet. We refer to a datacenter that is part of the CloudPath deployment as a *node*. The CloudPath tree can have arbitrary depth, and does not have to be balanced; different branches of a CloudPath network can have different height. New nodes can be attached to any layer of the existing tree.

While simple, this structure can accommodate different classes of applications, including workloads that aggregate data (such as IoT applications), or content delivery applications that cache data at different layers. This simple topology is a natural fit to the way mobile networks are currently organized in the physical substrate, and it also simplifies routing and configuration, as a node only needs to know its parent to join the network. However, other topologies may improve fault tolerance, are more robust to failures and allow for optimizations (e.g., direct data transfer between siblings, or load-balancing between siblings). We leave the exploration of alternative designs for future work.

CloudPath nodes are expected to differ widely in the amount of resources at their disposal, with storage and execution capacity expected to decrease as we descend levels in the hierarchy and move closer to the end-user device. Irrespective of size, each CloudPath node is comprised of the following modules:

- **PathExecute**: Implements a serverless cloud container framework that supports the execution of lightweight stateless application functions.
- **PathStore**: Provides a distributed eventual consistent storage system that manages application data across CloudPath nodes transparently. PathStore is also used internally by PathDeploy and PathRoute to fetch application code and routing information.
- **PathRoute**: This module routes requests to the appropriate CloudPath node. The user's location in the network, application preferences, and system state (e.g., application availability, load) are considered when making routing decisions.
- **PathDeploy**: Dynamically deploys and removes applications from CloudPath nodes, according to application preferences and system policies.
- **PathMonitor**: Provides live monitoring and historical analytics on deployed applications and the CloudPath nodes they are running on. Aggregates metrics from other CloudPath modules in each node, collects them using PathStore, and presents the results in a simplistic web interface.

In addition to the modules above, the root node located in the wide-area cloud also contains a module called *PathInit*. Developers upload their application to CloudPath through this module.

4.1 PathExecute

PathExecute implements a serverless cloud container framework that supports the execution of lightweight stateless application functions in each CloudPath node. Function as a Service (FaaS), also known as Serverless Computing, is a cloud computing approach in which the cloud provider fully manages the infrastructure used to serve requests, including the underlying virtual machines or containers, the host operating system, and the application runtime. Despite the Serverless moniker, FaaS applications do require a server to run. Serverless reflects the fact that the *application owner* does not need to provision servers or virtual machines for their code to run on. FaaS applications are composed of a collection of light-weight stateless functions that run on ephemeral isolated environments. We argue that the small size and stateless nature of FaaS functions make them ideal candidates for our multi-tier path computing deployment.

Our current prototype requires functions to be implemented as Java Servlets and requests for these servlets arrive using HTTP. For each application running on a node, we spawn a separate Docker [30] Ubuntu container running Jetty web server [3]. Functions of the same application can share the same container, and the same container is reused across multiple requests; however, a container may be terminated by the framework without notice

```
public class ClockService extends Action {
    public String getTimeZone() {
      Select s = QueryBuilder.select().all().from("clock");
      s.where(QueryBuilder.eq("userId", CurrentUserID));
      ResultSet results = pathstore.execute(s);
      Row row = rowList.results.one();
      int tzOffSet = row.getInt("tzOffset");
      return "<p>The zone is:" + tzOffSet + "</p>";
   }

    public String getPreferences() {
           ......
           ......
    }
}
```

(a) Function definition

```
<CloudPath_app>
  <mapping>
    <uri_pattern>/timeZone</uri_pattern>
    <function>ClockService.getTimeZone</function>
    <loc_pref>edge</loc_pref>
  </mapping>
  <mapping>
    <uri_pattern>/prefs</uri_pattern>
   <function>ClockService.getPreferences</function>
    <loc_pref>core</loc_pref>
  </mapping>
  <sub-domain>clockapp</sub-domain>
</CloudPath_app>
```

(b) Function registration

Figure 3: CloudPath application example. The application consists of two functions: *getTimeZone()* and getPreferences(). These functions are registered as CloudPath entry points(*/timeZone* and */prefs*) by mapping the function to a URI using the *web.xml* file shown in part (b). The location where each function needs to run is also specified in this file. The full URI will include the application name and *CloudPath.com*, e.g., *clockapp.cloudpath.com/prefs*.

and developers should not make any assumption about the local availability of state generated by previous function invocations. Our applications can scale horizontally in a datacenter by adding a load balancer for the application.

We implement PathExecute based on Nomad [5], a cluster manager and task scheduler that provides us with a common workflow to deploy applications across each of our CloudPath nodes. Nomad interacts with Consul [2], a highly available service registry and monitoring system inside each node. Information about running applications required for running Nomad is stored in Consul.

While we anticipate that most CloudPath applications will be written from scratch to take advantage of the unique execution environment afforded by the platform, PathExecute lets application developers leverage existing code and libraries by including them in their deployment package as statically linked binaries. In addition, PathExecute containers can be configured by the CloudPath administrator to include popular binary libraries, such as OpenCV [25].

Cloudpath uses URIs (Uniform Resource Identifier) to identify individual functions. These URIs consist of the application's name concatenated with the suffix *CloudPath.com* followed by the name of the function. Developers determine how URIs are mapped to functions using a deployment descriptor file that should be included in the application package. In addition, developers also specify their preferences for where functions should be deployed in CloudPath hierarchy using the deployment descriptor file. In our current implementation, the standard Java deployment descriptor for web applications (the *web.xml* file) is used to describe how and where the application and its functions should be deployed. Figure 3 illustrates how two URIs are mapped to functions and their preferred location to run (edge for /timeZone, and core for /prefs).

4.2 PathStore

PathStore provides a hierarchical eventual consistent database that makes it possible for CloudPath functions running in PathExecute containers to remain stateless by automatically replicating application state close to the CloudPath node where the function executes.

PathStore's target environment poses three interesting challenges. First, most nodes can only store a small fraction of the data stored on the wide area cloud nodes that are at the root of the hierarchy. Nevertheless, most reads and writes executed by a CloudPath function should be executed locally; running code close to the edge of the network has little benefit if most data accesses have to go to the cloud. Second, the large number of nodes in the system requires keeping to a minimum the amount of meta-data regarding the current location of data replicas. Third, the geographic distribution of nodes, and the high network latency typical of many paths between nodes requires minimizing coordination and the ability to operate (albeit at diminished capacity) even in case of temporary network or node failure.

To address these challenges, we structured PathStore as a hierarchy of independent object stores. The database of the PathStore node at the root of the hierarchy is assumed to be persistent, while all other levels act as temporary partial replicas. To simplify the implementation, PathStore requires the data replicated by a node to be a superset of the data replicated by its children. To provide low-latency, all read and write operations are performed against the local database node to which an application server is attached. PathStore supports concurrent object reads and writes on all nodes of the database hierarchy; updates are propagated through the node hierarchy in the background, providing eventual consistency.

Figure 4 shows a sample three layer PathStore deployment. PathStore consists of three main components: a *native object store*, the PathStore *server*, and the PathStore *driver*. The native object store provides persistent storage for objects that are temporarily (or permanently in the case of the root) replicated at a node. In our prototype we use Cassandra [26], but the design can be adapted to other storage engines (see 4.2.7). As the figure illustrates, the size of the local Cassandra cluster can differ between nodes. The PathStore server copies data between its local Cassandra instance and the Cassandra instance of its parent node. Finally, the PathStore

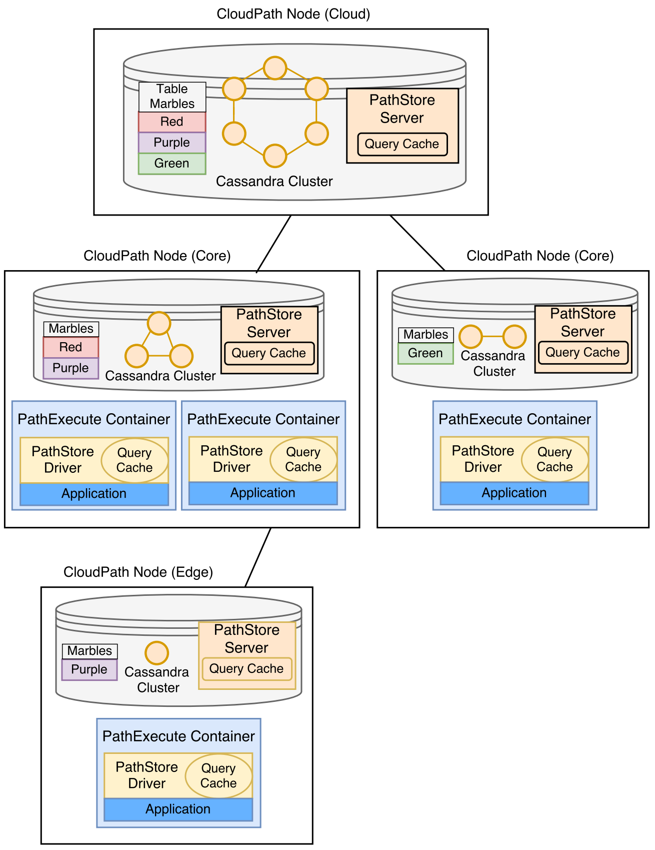

Figure 4: PathStore Architecture

driver provides an API that third-party applications running inside PathExecute containers can use to query the local PathStore node. Our prototype is based on CQL, Cassandra's SQL dialec, which organizes data into tables, and provides atomic read and write operations at row granularity. CQL lets users read and write table rows using the familiar SQL operation *SELECT*, *INSERT*, *UPDATE*, and *DELETE*; however, CQL operations are limited to a single table – there is no support for joins.

4.2.1 On-Demand Replication. PathStore replicates data at row granularity on demand in response to application queries. Applications issue queries using the PathStore driver which executes them against the local PathStore node; however, before a CQL query is locally performed, PathStore server replicates from the parent node all objects that match the query as determined by the conditions in the *where* clauses of the CQL statement. To prevent a node from fetching data on each query from its parent, the PathStore server keeps a *query cache* consisting of all recently executed CQL queries. Subsequent CQL queries that match an existing entry in the cache are directly executed on the local node. Queries in the query cache are periodically executed in the background by a *pull daemon* to synchronize the local node's content with that of its parent (i.e., fetch new and updated records from the parent node). To reduce unnecessary processing, PathStore keeps track of the coverage of cache entries and the pull daemon bypasses queries that are otherwise subsumed by other queries that have a wider scope. For example the query *select * from marbles* subsumes the query *select * from marbles where color = red*.

Figure 4 illustrates this process for a simple table that keeps track of marbles of different colors. In the example, an application running at the edge node issues a query for the purple marble (select * from marbles where color='purple'). Assuming that this query does not match an existing entry in the edge node's query cache, the query is propagated to the core node's PathStore server, which in turn propagates it to the cloud node's PathStore server. Since the cloud node is the root of the hierarchy, it is assumed to contain all the data and the query does not propagate any further. The core node then executes the query against the Cassandra cluster of its parent node, stores the matching row(s) in its local Cassandra cluster, and stores the query in its query cache. This process is repeated by the PathStore server running on the edge node. Finally, the PathStore driver executes the query against the edge's Cassandra instance. As an optimization, the PathStore driver also keeps a query cache with recently execute queries. Since the driver's cache is guaranteed to be a subset of the server's cache, queries that match the driver's can run directly against the local Cassandra instance bypassing the need to first contact the PathStore server.

The obvious disadvantage of fetching data purely on demand in response to application queries is the significant latency associated with fetching data across multiple levels of the hierarchy. It is easy to imagine alternative approaches that pre-fetch data in anticipation of its use. PathStore could leverage its fine grain knowledge about the data used by applications in the past to predict future usage. For example, it may be possible for PathStore to identify data that is requested for each user served by an application. When a new user connects to a node, PathStore could eagerly fetch the data associated with the new user in anticipation of its use. We leave the exploration of prefetching alternatives for future work.

4.2.2 Update Propagation. PathStore applies all modifications locally, and a *push daemon* periodically propagates local updates to higher levels of the hierarchy. PathStore keeps track of modifications using a write log. In Cassandra, every table has a partition key that determines the host(s) in the Cassandra cluster where a given row will be stored. In addition a Cassandra table can have one of more clustering keys. Rows with the same partition key, but different clustering keys are stored together on the same Cassandra host, in a local order determined by the clustering keys. PathStore implements a write log for each row of a table by adding a *version* column as the last element of the table's clustering key. The *version*, is a UUID timestamp that records the time the row was inserted, and the ID of the PathStore node where the modification was originally recorded. PathStore assumes that nodes are tightly synchronized using some accurate mechanism, such as GPS atomic clocks. As modifications get propagated through the hierarchy (up by the push daemon and down by the pull daemon), PathStore uses the version timestamp to determine order between modifications. In the current prototype the modification with the most recent timestamp wins.

PathStore's write log is not visible to applications, and therefore developers do not have to modify their application queries. Instead,

user	movie	version	rating
John	Toy Story	d33d7fe0-195f-5d569c585662	10
John	Toy Story	825968c0-195d-5d569c585662	8
John	Cars	7adf7210-1958-59e16851d966	9
Susan	Finding Nemo	6833c850-1958-59e16851d966	8

Figure 5: Sample PathStore table.

the PathStore Driver automatically collects the multiple versions of a row that match an application's query and returns the most recent data. For example, Figure 5 shows a table that keeps track of personalized movie ratings. Columns *user* and *movie* are the original partition and clustering keys, respectively. Column *version* is added by PathStore to implement the write log. The table show that user John initially assigned a rating of 8 to the movie Toy Story, but later updated this rating to 10. Running the query *select * from movies where user = 'John'* produces the tuples ['John', 'Toy Story', d33d7fe0-195f-5d569c585662, 10] and ['John','Cars', 825968c0-195d-5d569c585662, 8]; however, the PathStore driver returns only the most recent version of each row and hides any PathStore meta columns, i.e., ['John', 'Toy Story', 10]. Finally, to prevent the log from growing unbounded, PathStore runs a daemon at the root of the hierarchy that periodically trims the log.

4.2.3 Data Eviction. Cold query cache entries are deprecated periodically preventing the pull daemon from fetching unnecessary data. Similarly, locally replicated rows that do not match any query in the query cache are periodically deleted. In case of resource contention, our prototype uses a simple LRU policy to free space. Exploring other approaches is the subject of future work.

4.2.4 Local Table. PathStore also provides local tables for temporary storage. Updates to local tables are not propagated to other nodes. In Section 5.1 we describe an application that uses local tables to aggregate sensor data at the edge of the network.

4.2.5 Consistency Model. At the individual node level, PathStore preserves the storage semantics of its underlying native object store. Our current prototype, which is based on Cassandra provides local durability, row-level isolation and atomicity, and strong consistency based on Cassandra's quorum mechanism. Across nodes, however, PathStore propagates updates at row granularity following an eventual consistency model. The PathStore driver guarantees that code executing on a specific PathStore node will see monotonically increasing versions of a row (i.e., the driver returns only the most recent version of the row in the write log), and that given enough time without new modifications all replicas of a row on all PathStore nodes will converge to the same most recent value.

Whereas PathStore does not enforce system-wide strong consistency, an application can nevertheless achieve stronger consistency for requests emanating from a subset of the CloudPath hierarchy by instructing the platform to execute its sensitive functions at a common ancestor node. For example, a function running at city-level nodes will provide a consistent view of the data for all users in any given city, irrespective of the edge node they each use to connect to the network; users in different cities, however, may see inconsistent data while updates propagate through the hierarchy. An application can enforce global consistency by limiting its functions to run at the root of the hierarchy. The stronger consistency, of course, comes at the cost of increased network latency and obviates the benefits of path computing. In the future, we plan to explore other consistency models that will enable applications to control how updates are applied across the storage hierarchy.

4.2.6 Fault Tolerance. PathStore can continue to serve read queries for data that is locally replicated even in the event of network partition; however, queries that are not already in the query cache (of the current node and its reachable ancestors) will fail if an ancestor becomes unreachable. On the other hand, write queries should be able to execute as long as the local Cassandra instance is reachable. A write returns when it is persisted in the local Cassandra instance, and it is guaranteed to remain stored in the local instance until it is propagated to the parent node. A row is marked *dirty* when it is inserted into the local Cassandra instance. PathStore only marks the row as *clear* when the parent acknowledges reception and storage of the write. If there is a failure, PathStore retries propagating the write. If a PathStore node experiences a temporary failure, upon recovery it will retry propagating all writes locally marked as dirty. Data is only permanently lost if a PathStore node experiences a permanent failure before a dirty update is successfully propagated to the parent. We anticipate that permanent PathSore node failure will be an very rare occurrence as PathStore relies on replication in Cassandra to handle individual machine failures.

4.2.7 Other Storage Engines. Whereas the current PathStore implementation leverage Cassandra other similar object stores could be adopted as long as they provide (at the local node level) persistent object storage, row-level isolation, and atomic timestamps.

4.3 PathRoute

PathRoute is responsible for routing CloudPath requests to running functions using the URI included in the request. In our current implementation, which uses HTTP to transfer requests to functions, CloudPath applications gets a unique sub-domain within the *CloudPath.com* name-space after registration. CloudPath requests should consist of the application sub-domain concatenated with *CloudPath.com* followed by the function name in the web-address such as in: *app_name.CloudPath.com/function_name*. In the clock application example of Figure 3, requests to the *getTimeZone()* function should be made to the *clockapp.CloudPath.com/timeZone* web-address.

To divert CloudPath traffic from other traffic flowing in the network, we use a DNS *A record* entry for *CloudPath.com* to map all CloudPath sub-domains to a single IP address. Hence, all CloudPath application requests across the entire network will have the same destination IP address which is the IP address of the PathRoute module on every edge node. The network operator is required to route all packets destined for this IP address to the edge CloudPath node connected to the user. The major benefit here is that by using only one static route on the edge routers, CloudPath traffic can be diverted to the PathRoute module.

For each new HTTP request received by the proxy, a look-up is made on a local in-memory state cache using a small script to determine whether a deployment request for that application on the node has been previously made or not. If not, the application identifier is extracted from the request and sent to the PathDeploy module where a decision for application deployment is made. When PathDeploy decides to deploy the application on the node and PathExecute completes the application deployment, the proxy cache is updated. Future requests are then proxied to the PathExecute container running the application (the function preferences should also match the node's location).

If PathDeploy decides against deploying the application on the node, subsequent requests will be proxied to the next CloudPath node in the hierarchy (we assume each PathRoute proxy has the address of the PathRoute module of its parent node). In CloudPath, requests can only move upwards towards the root cloud node. We implemented PathRoute using a NGINX [32] proxy.

4.3.1 Handover. When the user moves between edge nodes (e.g., handover), the IP address of the source client and the destination which is the PathRoute module on the edge node node is still valid, but the traffic will flow through a different set of routers and will therefore lead to the execution of the function at a different CloudPath node. In case of a hard handover the existing TCP connections would be terminated, and the client is forced to reconnect and restart their request. When a soft hand over happens, the connection is restarted at the edge CloudPath node. Because of the short-lived nature of the requests, restarting the connection would not lead to significant overhead.

4.4 PathDeploy

PathDeploy is responsible for initiating the process of deploying an application and its functions on a node. Application deployment decisions are triggered by PathRoute requests. The decision on whether to deploy the application on a particular node depends on higher level system policies, user preferences and the resource status on that node. One policy that we include in our prototype is that functions specified by the user to run on a certain level of the CloudPath hierarchy can also run on any higher level and all functions by default run on the cloud node.

Our PathDeploy prototype is a Java HTTP server. When new requests for an application and function deployment arrive at PathDeploy, it retrieves the application and function information, including user preferences from PathStore to decide if the application and function should be running on the node or not. If the decision to deploy an application is positive, PathDeploy deploys the application locally by fetching the application code from PathStore and passing it to PathExecute along with the application meta-data. In parallel, it fetches the application database schema from PathStore and creates the application data tables on the node.

At present, when we deploy an application on a node, all its functions are deployed at once, but requests are only forwarded to a subset of functions that should be deployed on that node. A fine grain deployment scheme will be implemented in future versions of CloudPath.

4.5 PathMonitor

PathMonitor is designed to provide insight into the lifecycle of a deployed application as well as the status of the CloudPath nodes themselves. It consists of both a back-end module that collects data from the various modules and third party applications of CloudPath, as well as a front-end web application to present the collected data.

PathMonitor pulls stastics such as CPU and memory metrics for containers and hosts from the PathExecute module. This data is preprocessed, aggregated, and stored on the PathStore module in the node. In addition, the various CloudPath modules and third party applications create logs depending on the application function; such as access, status, and error logs. PathMonitor acts as a central point for collecting and storing logs that are created by these other modules.

The front-end web interface lets us visualize the current and past state of the CloudPath system through various graphs and infographics, such as; the topology of the system, including the hierarchy of nodes; CPU and memory metrics for application containers; and the same metrics for the hosts and nodes themselves.

4.6 PathInit

In the root cloud node, the PathInit module is responsible for receiving the applications from the developers through a web interface, extracting application and function properties from the *web.xml* file, and saving them along with the application's code and database schema file on PathStore.

PathInit also creates augmented tables from the submitted application database schema file, and deploys the application on the root node. The augmented schema files and the information about the application are used by PathDeploy to deploy the application on other nodes.

5 EXPERIMENTAL SETUP

Our experiments emulate a CloudPath deployment consisting of a cloud node, and two mobile networks each with one core node and two edge nodes. Figure 6 depicts this topology. Each CloudPath node is implemented in a separate computer. The network between the clusters is emulated using Linux's Traffic Control, that enabled us to configure the Linux kernel packet scheduler. Network latencies are chosen based on results from the paper by Hu et al. [24], and we use a normal distribution to describe the variation in delay. The average round trip times of the links is included in Figure 6.

5.1 Test-Cases

We created a series of microbenchmarks to measure deployment time, routing overhead, and the latency and throughput of PathStore. We have also implemented a series of sample user applications to show how our platform supports different categories of applications that can benefit from the architecture:

Face detection: Computational resources on edge nodes can be suitable for offloading resource-intense functions from the mobile end-user device. When offloaded to the edge, applications can benefit from an increase in execution speed and battery lifetime [10]. Our face detection application is deployed as a Servlet and uses the image processing library OpenCV [25] through its Java interface JavaCV to detect faces in an image. The input to this application is an image sent in an HTTP request. The application finds faces in the image and saves them in PathStore.

Deployment Location	PathDeploy Processing	Nomad Processing	Container Spawning	Application Initialization	Consul Update	Database Initialization	Total Time
Cloud	463.9 (118.1)	131.6 (29.7)	1754.9 (28.3)	841.0 (22.8)	33.1 (13.6)	234.3 (22.1)	3531.2 (133.8)
Core	823.0 (140.1)	129.7 (32.8)	1772.1 (53.6)	887.0 (27.6)	36.6 (19.5)	622.5 (35.2)	3849.8 (154.5)
Edge	1005.2 (155.5)	133.6 (45.3)	1709.3 (45.4)	866.4 (18.9)	39.4 (20.2)	974.3 (55.4)	4084.3 (181.3)

Table 1: Breakdown of average time required to deploy an application in milliseconds on different locations. Standard deviation in parenthesis.

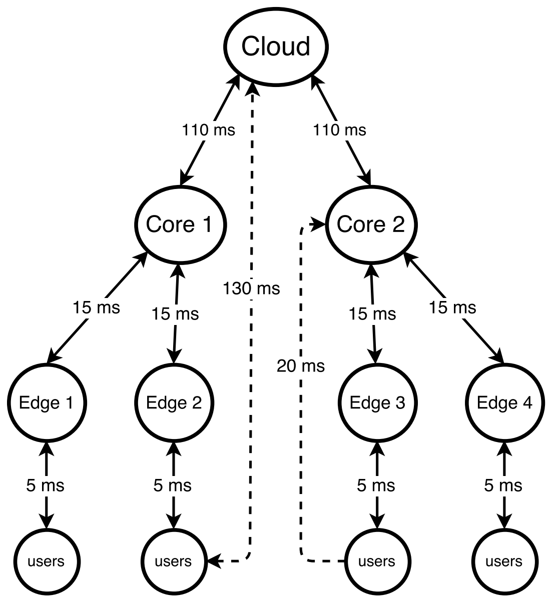

Figure 6: Topology of our experimental setup network with average round trip times of the links

Localized face recognizer: Using PathStore, applications can push localized content to edge nodes based on the geographic location of end-users. One example is face recognition classifiers which have been trained on a specific dataset, relevant to a given geographical location. We use the AT&T face dataset [1] consisting of a total of 400 face images, of 40 people (10 samples per person) and divide it into 4 separate smaller datasets. We then train 5 different classifiers using the FisherFaces algorithm in OpenCV on these smaller datasets. We store the classifiers in PathStore and the face recognizer application running on each edge node retrieves them.

IoTStat aggregator: Another important benefit of having multiple processing edge nodes close to the user is their ability to filter and aggregate streams of data. As the number of IoT devices using the Internet is likely to raise significantly in the future, processing and filtering data on the edge will decrease the amount of traffic from these devices that need to go through the Internet. We implemented a sample application that performs aggregation functions (average, min, max) on data received from sensors on edge nodes. The first handler of this application receives and parses HTTP/JSON requests containing the sensor data, and stores the extracted information onto a local PathStore table. A second handler, that can be called periodically using HTTP requests, then performs MIN, MAX and AVERAGE queries on the data stored by the first function within specific time frames. This processed data is then saved in another regular PathStore table, which is pushed to the core and cloud.

6 RESULTS

We evaluate CloudPath and its applications from different aspects:

- The deployment time of applications on a specific node
- The minimum routing time for applications
- The performance of PathStore and its overhead
- Connection handover between edges
- Benefits for applications

6.1 Deployment Latency

We measure the performance of our system in terms of average deployment time of a sample Hello World application with one table. The process is initialized by an HTTP request received by PathRoute, which triggers a container deployment request in PathDeploy as the application is not already deployed on that node. Table 1 shows the amount of time required by PathDeploy and PathExecute to retrieve and deploy an application on a particular node. Each experiment was repeated 15 times. The initial time to retrieve application and function information from the database and make a deployment decision is shown in the first column (PathDeploy Processing). Then the next steps (Nomad Processing, Container Spawning and Application Initialization) are done in PathExecute while the application database initialization from the stored schema file is done in parallel. As shown in this table, as we move from the cloud towards the edge, the average database initialization time and the PathDeploy processing time increases. We assumed the worst case scenario where the application data has to be fetched all the way from the cloud. In practice, an edge deployment will likely get a hit on the core tier. However the overall processing time is still between 3.5 seconds in the cloud to 4.08 seconds in the edge. If developers have larger applications with more complex databases, this time is likely to increase because more data should be retrieved from PathStore.

A non-FaaS approach requires the full VM or container to be downloaded on the edge node each time it is required. To compare that approach with ours, we measured the overhead time required

Execution Location	Direct Connection	Direct Connection (no latency)	L7 Routing	L7 Routing (no latency)
Edge	5.38 (0.38)	0.413 (0.2)	5.755 (0.411)	0.89 (0.22)
Core	20.33 (1.37)	0.465(0.32)	20.96 (1.64)	1.23 (0.77)
Cloud	130.46 (6.65)	0.443(0.38)	132.20 (7.45)	1.45(0.93)

Table 2: Average RTT for HTTP requests in milliseconds. Standard deviation in parenthesis

to download a minimal container with only Java installed from a cloud repository, which on average was 13.2 seconds. In CloudPath, this time is saved during each deployment because all clusters are pre-loaded with the executing container.

6.2 Routing Overhead

To calculate the overhead that our systems adds to each packet, we measure the average response time of requests using an HTTP benchmarking tool called wrk [17]. We compare the latency of a baseline approach where no proxies exist between the user and the container (direct connection) to our method, where we use layer 7 routing using PathRoute. In our experiment, after creating a single TCP connection with the proxy, a new request is sent when an acknowledgment for the previous request is received. This is repeated for 10 seconds for 16 concurrent connections. The average round trip times (RTT) and standard deviation is presented in Table 2. Furthermore, we compare our L7 routing with the baseline approach, when their isn't any emulated latency in the network. As shown in this Table, our routing method only introduces a slight increase in latency compared to the baseline approach where packets are routed on layer 3. This is specially evident when no emulated latency exists in the network and shows the overhead introduced by our PathRoute proxies is about 0.9ms for the first layer and about 0.3ms for each additional layer.

6.3 PathStore Performance

We next measure the performance of PathStore using micro benchmarks. We present the time required to execute different types *SELECT* and *INSERT* queries both when data is located locally or is available on a parent node.

6.3.1 Local Read Latency. We compare the time to execute *SELECT* queries on a local node using PathStore versus using the native Cassandra driver. In Figure 7, we depict the Cumulative Distribution Function (CDF) of the time it takes to execute these queries. The blue line shows the performance of the native Cassandra driver while the green line shows the performance of the PathStore driver when we execute 1000 *SELECT* queries with *WHERE* clauses that match individual rows and result in a miss in the local client query cache. Each row of our table contains $1KB$ of data. The red line shows the performance of the PathStore driver with the difference that before 1000 individual *SELECT* queries, a single *SELECT* query without any clauses on the same table is made so that data would be cached locally. This results in hits on the client query cache. As shown in Figure 7, the PathStore driver is on average 1.6ms slower than the native Cassandra driver when it misses the client query

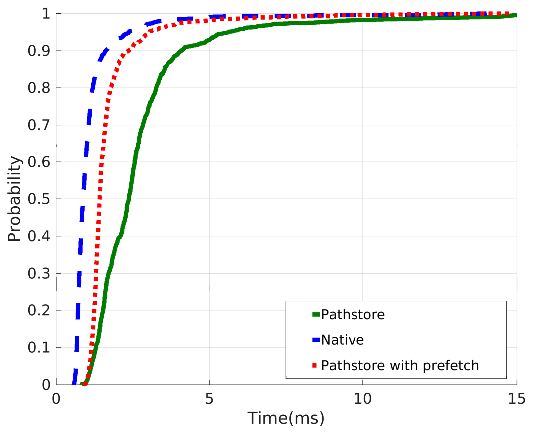

Figure 7: CDF chart for 1000 select queries on a local node.

cache. However the performance of our drier is close to the baseline if we have hits in the query cache.

6.3.2 Local Write Latency. We also measured the time required for executing local *INSERT* queries and there were no noticeable difference in terms of performance between the PathStore driver and the Cassandra driver. For *INSERT* queries, our PathStore driver does not add extra overhead to the native Cassandra driver.

6.3.3 Remote Read Latency. We next analyze the time required to retrieve data to the edge and core from the cloud node using *SELECT* queries that match individual rows. These queries are executed from the core and edge nodes where there is a miss in the client and server query cache of the PathStore unit . We measure The retrieval time for 1000 queries and present the CDF in Figure 8. The green and red figure show PathStore's execution time from the core and the edge. The blue and orange lines show the execution time for the native Cassandra driver from the core and the edge. We can see that in Figure 8, PathStore (red and green lines) is nearly twice as slow as the Cassandra driver (blue and orange lines) in retrieving the data. This is because PathStore first checks to see whether data is present on the parent node or not. Then when it gets the response back, it fetches the data from the remote node. This adds one RTT time to each query.

We next measure the time required to fetch different number of entries (a whole table) from edge, core and cloud nodes when the data is only initially located on the cloud node. Again each row or entry is 1KB. We do a *SELECT* query with no clauses to fetch the

Deployment Location	100 Entries PathStore First query	100 Entries PathStore Consequent queries	100 Entries Native Cassandra driver	1000 Entries PathStore First query	1000 Entries PathStore Consequent queries	1000 Entries Native Cassandra driver	10000 Entries PathStore First query	10000 Entries PathStore Consequent queries	10000 Entries Native Cassandra driver
Edge	529.1 (9.7)	4.7 (0.4)	506.4 (7.4)	1664.2 (177.1)	29.5 (9.3)	1035.5 (181.9)	8857.5 (123.9)	113.0 (8.4)	2640 (195.0)
Core	510.4 (13.8)	4.4 (0.1)	398.9 (15.9)	1121.8 (47.3)	24.8 (0.7)	869.0 (34.3)	5244.8 (282.1)	111.1 (2.1)	2268.7 (147.4)
Cloud	6.3 (0.6)	4.9 (0.2)	6.1 (0.1)	29.7 (0.9)	26.9 (1.6)	27.1 (1.78)	137.5 (6.7)	112.8 (6.4)	124.9 (3.1)

Table 3: Time required for querying full tables in milliseconds (standard deviation in parenthesis).

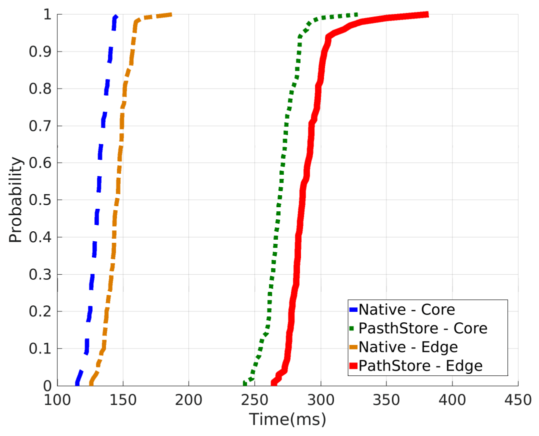

Figure 8: CDF chart for individual select queries on a remote node.

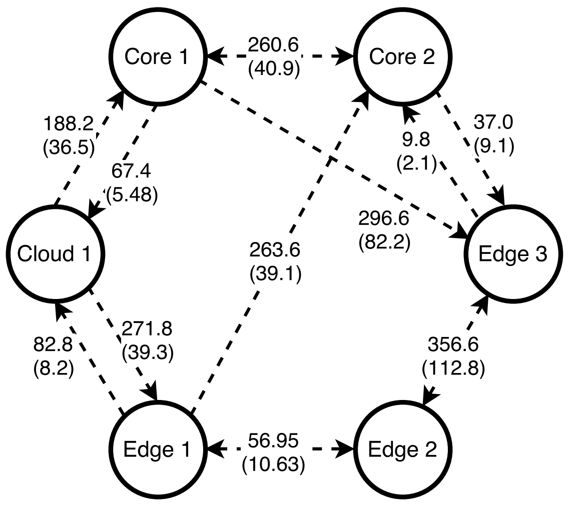

Figure 9: Propagation time (in milliseconds) of an *INSERT* query between nodes (standard deviation in parenthesis). Links are logical links between nodes from Figure 6.

whole table at different nodes with PathStore and the native Cassandra driver. The experiment is repeated 20 times and the results are presented in Table 3. As shown in the table, PathStore's first query takes more time than the native Cassandra driver to retrieve the data, but for consequent queries, as the data is already fetched, we will have a hit on the local client cache and the queries would take the same time as a local *SELECT*. Furthermore consequent queries that are a subset of the first query will also be fetched locally.

6.3.4 Update Propagation Latency. We also measure the propagation latency of a single *INSERT* query. In this scenario we execute a single *INSERT* query at different nodes of the hierarchy and measure the time that the single row update takes to propagate to all other nodes that have previously issued a *SELECT* query. Figure 9 illustrates the results of this experiment . The links shown in this Figure are logical links between nodes from Figure 6. Meaning a query travelling between *Edge*1 and *Core*2 has to traverse nodes *Core*1, *Cloud*1. As shown in this Figure, moving down on the tree takes more time than moving up. This is because pull operations require 2 RTT's while push operations only need half an RTT. *INSERT* and *UPDATE* operations on a node get pushed all the way to the cloud, while nodes can express their interest in a certain query with a *SELECT* query. This results in periodic data pulls from parent nodes when updated information on the parent about the query exists.

6.3.5 Data Overhead. Finally we measure the total overhead of each table in PathStore. We add 5 columns to each of our PathStore tables and in total, 38 bytes is added to each entry.

6.4 Handover Latency

We examined the effects of a soft handover in case of mobile movement between two edge nodes *A* to *B*. When a soft handover happens, then TCP packets will continue to be sent from the user device (as explained before, all PathRoute modules on every edge, have the same IP address) however the data arriving at the PathRoute module of edge will not accept such packets because no such TCP connection exists, so it sends a TCP packet with the RST flag set and the connection will be re-initiated by the user device. We emulate a soft handover in our environment and measure the time required to re-initiate the connection. On average it takes 16.56 milliseconds (4.03 standard deviation for 100 experiments) for the device to start re-initiating the connection, and another 15.2 milliseconds to establish a new TCP connection.

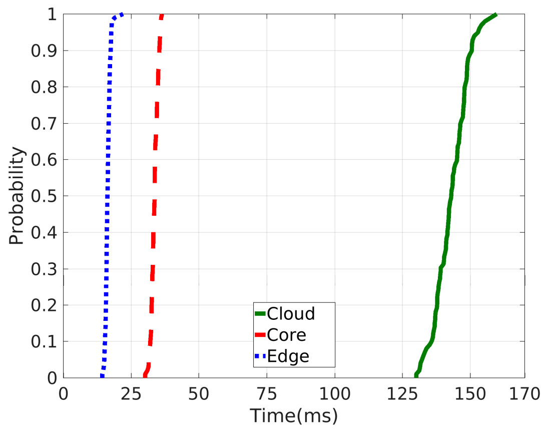

Figure 10: CDF for response time of the Face Recognition application.

6.5 Application Performance

In this section, we will show the benefits of running applications on CloudPath. These benefits come in the form of greatly reduced response time or reductions in data and network traffic.

6.5.1 Face Detection. We measure the average response time of the face detection and face recognition programs when they are deployed at different locations. For the face detection program, the average response time for 100 different requests when the program is running on the edge, core and cloud is: 13.6(2.1), 32.6(1.8), 141.9(6.4) milliseconds (standard deviation in parentheses). The same image was used and the file size was $2KB$. There is a substantial reduction in response time when running on the edge (closer to the client) compared to running on the core and cloud.

6.5.2 Face Recognition. The Face Recognizer program labels an input image (received through HTTP requests with a file size of 11KB) based on a trained model. The results for processing 100 requests are illustrated in Figure 10. Similar to the Face detection program, running on the edge lowers the latency by 88 percent.

6.5.3 IoTStat Aggregator. For the IOTStat aggregator application, the average processing time of each query received at the edge node is 1.6 ms for 3 different sensor values in the same request. This means that an application running on the container (1 core) can handle up to 900 queries per second. The processing time required for aggregating 1000 queries, is about 10ms. If their were no aggregation, then n sensors sending k requests per second will send $n \times k$ messages to the cloud for processing. However if we assume a single layer of aggregation (p edge nodes), assuming that aggregation results are required every second, then the total number of messages sent to the cloud would only be: p. In our example we insert a row containing the aggregation results of each edge node on to PathStore which would push this data to the cloud.

7 RELATED WORK

Early edge computing systems have relied on virtual machines (VM) as the unit of application deployment [19, 33]. These systems rely on optimizations, such as VM synthesis [22] and uni-kernels [29], to reduce the network traffic and deployment time. More recently, several research platforms have switched to operating system containers as the unit of deployment [11, 27]. While operating system containers are smaller than VMs, they can still require the transfer of hundreds of megabytes to instantiate a container.

In contrast, this proposal leverages a new cloud computing model known as Function as a Service (FaaS). Examples of FaaS systems include Amazon Lambda [36], OpenLambda [23], IBM OpenWisk [9], and AppScale [14]. All these systems target the wide-area cloud environment, and assume a flat replicated environment with a relatively small number of large datacenters accessible over the Internet. Our work differs in that it is the first application of FaaS to be running code on a hierarchy of datacenters stretched from the network edge to the wide-area cloud.

Path computing has similarities to previous approaches that have infused networking nodes with processing, such as active networks [13] and the intentional naming system [6]. These previous effort, however, focused on low-level network processing (e.g., encryption, routing, load balancing), whereas CloudPath targets full server workloads.

Previous work has explored automatic application partitioning and migration [15, 16, 18]. In comparison, our approach requires application developers to explicitly partition their applications into clearly-defined functions. We argue that this approach is consistent with existing best practices for web back-end design, which mandate the use of stateless REST functions for scalability and fault tolerance.

A large body of research exists about replicated databases for geographically distributed datacenters both in industry and academia [20, 28, 34, 37]. These systems offer stronger consistently models, but assume a flat overlay structure. In this paper, we use Cassandra as an existing widely used system and use it as the basis for our hierarchical storage system.

8 CONCLUSION

We presented *path computing*, a new paradigm that enables processing and storage on a progression of datacenters interposed along the geographical span of the network. Path computing gives applications developers the flexibility to place their serve functionality at the locale that best meets their requirements in terms of cost, latency, resource availability and geographic coverage.

We also described CloudPath, a new platform that implements the path computing paradigm. CloudPath minimizes the complexity of developing path computing applications by preserving the familiar RESTful development model. CloudPath applications consist of a collection of short-lived and stateless functions that can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. CloudPath makes this functional decomposition possible by providing an eventual consistent storage service that automatically replicates application state on-demand across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

Our experimental evaluation showed that CloudPath can deploy applications in less than 4.1 seconds and has negligible read and write overhead for locally replicated data. Moreover, our test applications achieve up to 10X reductions in response time when running on CloudPath compared to an alternative implementation running on a wide-area cloud datacenter.

In the future we plan to improve support for user mobility by leveraging application access patterns to pre-populate data, as well as new consistency models that improve control over update propagation. Finally, we plan to explore more sophisticated network topologies.

REFERENCES

[1] 2008. AT&T Database of Faces. http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html. (2008).

[2] 2017. Consul by Hashicorp. https://www.consul.io/. (2017).

[3] 2017. The Eclipse Foundation. http://www.eclipse.org/jetty/. (April 2017).

[4] 2017. Huawei Service Anchor. http://carrier.huawei.com/en/products/wireless-network/small-cell/service-anchor. (2017).

[5] 2017. Nomad by Hashicorp. https://www.nomadproject.io/. (2017).

[6] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. 1999. The design and implementation of an intentional naming system. *ACM SIGOPS Operating Systems Review* 33, 5 (1999), 186–201.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, and others. 2009. *Above the clouds: A berkeley view of cloud computing.* Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[8] Paramvir Bahl. 2015. Cloud 2020: The Emergence of Micro Datacenters for Mobile Computing. http://tinyurl.com/hylpmgl. (may 2015).

[9] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. 2016. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems.* ACM, 287–288.

[10] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. Citeseer.

[11] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. 2016. Fast, scalable and secure onloading of edge functions using AirBox. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing.* Washington, DC.

[12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing.* ACM, 13–16.

[13] Kenneth L Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. 1998. Directions in active networks. *IEEE Communications Magazine* 36, 10 (1998), 72–78.

[14] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. 2009. Appscale: Scalable and open appengine application development and deployment. In *International Conference on Cloud Computing.* Springer, 57–70.

[15] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems.* ACM, 301–314.

[16] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM, 49–62.

[17] Will Glozer. 2017. wrk - A modern HTTP benchmarking tool. https://github.com/wg/wrk. (2017).

[18] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. COMET: code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12).* 93–106.

[19] Abhimanyu Gosain, Mark Berman, Marshall Brinn, Thomas Mitchell, Chuan Li, Yuehua Wang, Hai Jin, Jing Hua, and Hongwei Zhang. 2016. Enabling Campus Edge Computing Using GENI Racks and Mobile Resources. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing.* Washington, DC.

[20] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, and others. 2016. Mesa: a geo-replicated online data warehouse for Google's advertising system. *Commun. ACM* 59, 7 (2016), 117–125.

[21] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services.* ACM, 68–81.

[22] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. 2013. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.* ACM, 153–166.

[23] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16).*

[24] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems.* ACM, 5.

[25] Itseez. 2015. Open Source Computer Vision Library. https://github.com/itseez/opencv. (2015).

[26] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[27] Peng Liu, Dale Willis, and Suman Banerjee. 2016. ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing.* Washington, DC.

[28] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage.. In *NSDI,* Vol. 13. 313–328.

[29] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, and others. 2015. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* 559–573.

[30] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[31] Mugen Peng, Yong Li, Zhongyuan Zhao, and Chonggang Wang. 2015. System architecture and key technologies for 5G heterogeneous cloud radio access networks. *IEEE network* 29, 2 (2015), 6–14.

[32] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* 2008, 173 (2008), 2.

[33] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009), 14–23.

[34] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* USENIX Association, Oakland, CA, 351–366. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/sharma

[35] Tolga Soyata, Rajani Muraleedharan, Colin Funai, Minseok Kwon, and Wendi Heinzelman. 2012. Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Computers and Communications (ISCC), 2012 IEEE Symposium on.* IEEE, 000059–000066.

[36] Mario Villamizar, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, Mery Lang, and others. 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on.* IEEE, 179–182.

[37] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 292–308.

[38] Rong Yu, Yan Zhang, Stein Gjessing, Wenlong Xia, and Kun Yang. 2013. Toward cloud-based vehicular networks with efficient resource management. *IEEE Network* 27, 5 (2013), 48–55.