EdgeSys: A Decentralized Edge Computing Framework

Victor Chen*†, Geoffrey Phi C. Tran*†, John Paul Walters†, and Stephen P. Crago*†

*Ming-Hsieh Department of Electrical and Computer Engineering
University of Southern California, Los Angeles, CA 90089
Email: chen116@usc.edu

†Information Sciences Institute
University of Southern California, Arlington, VA 22203
Email: {gtran, jwalters, crago}@isi.edu

***Abstract*—We present EdgeSys, a decentralized edge computing framework for coordinating networks of resources (cloudlets) to run edge applications. EdgeSys is capable of returning processed data back to the mobile device, continuing application operation under dynamic network connection, and is compatible with Heron API. In this paper, we first establish the edge computing and application model. After that, we present the details of the EdgeSys framework. Then we evaluate EdgeSys with a Heron application and a Multi-agent Path Finding (MAPF) application. We show that Heron application deployment using EdgeSys does not require a fully connected cluster and can return result data to the mobile data source, which are not possible using the standard Heron framework. We also show that under cloud-like setting with fully connected cluster, Heron application deployed using EdgeSys on average only incurs 1.49 ms overhead comparing to using the standard Heron framework. Finally, the MAPF application experiment demonstrates EdgeSys's ability to dynamically reroute application data and maintain 0 deadline miss when experiencing connection failures in a cloudlet cluster.**

***Keywords*-decentralized computing; distributed computing; edge computing; cloudlet; mobile client; Heron API**

I. Introduction

The current cloud computing model, which focuses on running applications and processing data in data centers, is ineffective for geographically dispersed applications that demand intelligence at the edge. Relaying data to the remote cloud leads to high propagation delay, which cannot be overcome without moving computation closer to the edge devices. This is the premise of edge computing: bringing computational resources physically closer to the data and reducing dependence on a backend cloud.

In edge computing, distributed edge resources can process real-time data from devices with lower latency. A motivating edge application example is autonomous driving system which has end-to-end latency constraint of 100 ms for processing each frame captured by the vehicle's camera [1]. Measurements from CloudHarmony using the RIPE Atlas network shows the average end-to-end latency to Google Compute Engine and Amazon EC2 ranges from 30ms to 70ms [2], [3], which is a significant overhead for edge workload that has latency constraint in 100ms range. Edge computing eliminates those network latency overhead and enables resource-intensive and time-sensitive applications to be deployed at the edge [4]. However, new challenges arise in managing edge resources efficiently to provide a reliable platform for deploying edge applications.

Open Edge Computing Initiative [5] defines edge computing as micro data centers (cloudlets) providing low latency compute and storage resources with just one hop away from the users. We adopt this definition and envision it to be dynamic and reconfigureable as shown Figure 1. In Figure 1, each cloudlet is a micro data center comprised of one of more wired connected physical servers. The cloudlets dynamically connect with other cloudlets, forming an ad-hoc cloudlet network. Mobile devices are free to establish connection with any cloudlet to offload computation as the devices move across the cloudlet network.

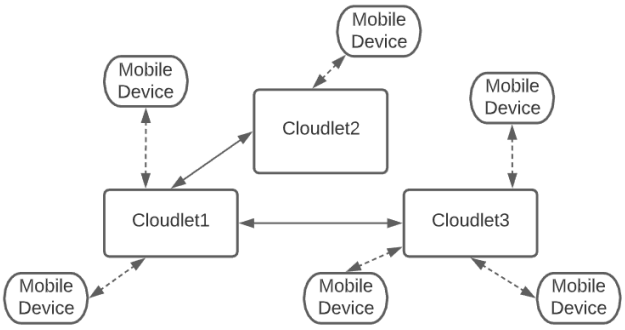

Figure 1: An ad-hoc Cloudlet Network with Mobile Devices

Many edge applications that process data streams generated by client devices such as cognitive assistance, remote monitoring, and autonomous vehicles utilize the dataflow programming model [4], [6], [7]. There are many existing stream processing frameworks that adopt the dataflow programming model such as Heron [8], Flink [9], and Samza [10]. But those frameworks' architecture including fault tolerance mechanism were designed based on the premise that the applications are deployed in data centers with fully connected clusters. In edge computing, a cloudlet cluster is not always fully connected and the connections are not as reliable as if they were in a data center. So if one were to deploy a an edge application using one of the above-mentioned frameworks, the application deployment size is limited by the available resource in a single cloudlet and not

able to take advantage of the aggregated resource provided by the cloudlet cluster.

We identify the need for an edge computing framework that is suitable for ad-hoc cloudlet network. To fill this gap, we create the EdgeSys framework to run edge application that complies with the dataflow programming model. In dataflow programming model, an application is represented as a topological graph with application operators as nodes. Application data flows through the operators starting from the data source and ends at the data sink. Using Heron's data model as an example shown in Figure 2, the spout (data source) emit data stream for the bolts (operators) to perform computation on the data. In a standard cloud framework such as Heron, the topology shown in Figure 2 is deployed in data center with static data source, whereas EdgeSys' objective is to run the same topology in an edge cloudlet cluster. Figure 3 shows a high level view on how the topology from Figure 2 can be deployed using EdgeSys. Under EdgeSys, the data source is replaced by mobile devices which generate multiple input data streams, and application operators are deployed across the cloudlets. The EdgeSys framework handles application operators deployment and data routing. In addition, we observe that many edge applications require processed data to be returned to the source device, so unlike conventional cloud frameworks which do not consider returning the result data to the data source, EdgeSys is able to return processed data to the mobile client devices.

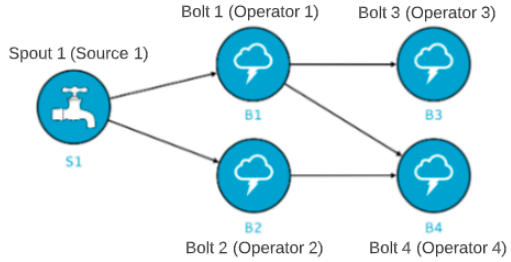

Figure 2: Heron's Dataflow Programming Model Example [11]

In this paper, we make the following contributions: we develop EdgeSys, a decentralized edge computing framework for deploying dataflow applications in ad-hoc cloudlet network. The EdgeSys framework comes with Heron API support and the capability of returning processed data back to mobile clients. Through experiment using Intel's stream processing benchmark [12], we show that using EdgeSys, a Heron application can be deployed in cloudlet cluster that is not fully connected and return result data to the mobile data source, which are not possible using the standard Heron framework. we also show that under cloud-like setting with fully connected cluster, Heron application deployed using EdgeSys on average only incurs 1.49 ms overhead comparing to using the standard Heron framework. Finally, we implement a MAPF application to demonstrate that EdgeSys can maintain application execution without missing any deadline under unstable cloudlet network condition.

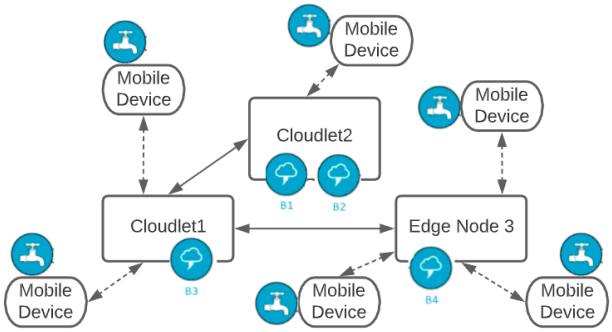

Figure 3: EdgeSys Application Deployment Example

The remaining paper is organized as follows: Section II presents related work on edge computing frameworks. Section III presents the details of the EdgeSys framework. Section IV present the system setup, the Heron and MAPF application used for evaluating EdgeSys, and the experiment results. Finally, Section V presents our conclusion and future work for EdgeSys.

II. Related Work

Fu et al. [13] create EdgeWise, a stream processing engine for the edge. EdgeWise develops a novel congestion-aware scheduler and a fixed-size worker pool to improve both throughput and latency. Their work is similar to ours in that they identity that the current cloud stream processing frameworks, which assume having cloud-class compute and network resources, are not suitable for the edge. But their work still assume compute resources are well connected, where our framework can run applications in ad-hoc cloudlet cluster.

Chad and Stoleru [14] develop R-MStorm, a stream processing framework for the edge. R-Mstrom assigned tasks based on the availability of the compute nodes to increase throughput and latency. Our work differ from their in that they utilizes mobile devices for compute and requires a centralized server for task assignment whereas our EdgeSys framework is completely decentralized, making it more fault tolerant in a dynamic network environment.

Amento et al. [15] design a high-level edge management system, FocusStack, that manages edge devices as infrastructure-as-a-Service clouds. They use location-based situational awareness to solve the inefficient messaging problem between the cloud and devices on a multi-tier geographic addressing network. Their work focuses on improving communication efficiency while our work focuses on managing edge resources to provide a deployment platform.

Wang et al. [16] develop the Edge NOde Resource Management (ENORM) framework, which provide provisioning and auto-scaling edge resources for applications. Their work is implemented on a single node where our framework is

deployed on a multi-node edge cluster. Their framework utilize a centralized resource monitoring and allocation scheme while our work take a fully decentralized approach.

Javed et al. [17], [18] design fault tolerant edge computing frameworks, IoTEF and CEFIoT, which utilize the built-in features of existing cluster and data management system such as Kubernetes [19] and Kafka [20]. Their work also support streaming like applications. However, their frameworks are only suitable for static clients and their work does not consider returning result data to the data source, our framework supports mobile clients with the option of delivering processed data back to the mobile data source.

III. The EdgeSys Framework

In this section, we present the EdgeSys architecture. As mentioned in Section I, cloudlets have less compute capability compared to the clusters in data centers. And cloudlets form ad-hoc connections unlike data center which have fully connected clusters. Base on those differences, we design EdgeSys to be a fully decentralized framework to avoid single point of failure for having a centralized controller. Under the EdgeSys framework, as shown in Figure 4, each cloudlet is a standalone EdgeSys entity that is capable of deploying applications. An EdgeSys multi-cloudlet cluster is formed by connecting and exchanging information with other EdgeSys cloudlet. Such EdgeSys cluster can process data offloaded by mobile clients that travel across cloudlets. In addition, bigger applications with resource requirements that span multiple cloudlets can be deployed using an EdgeSys cluster.

A. EdgeSys Internals

As shown in Figure 4, every EdgeSys unit running inside a cloudlet consists three major components: Backend Service, Application Management, and Cluster Management. In this subsection, we present the details of each module.

—Backend Service:

1) Queueing Service: As mentioned in section I, many edge applications can be modeled as dataflow applications composed of different operators with data flowing through different operators starting from the data source. Based on this application model, we can split up an application into operators that reside on different cloudlets. EdgeSys uses dedicated input/output queues to interface with individual deployed operator as shown in Figure 5. EdgeSys also uses the queueing service to set up the server queue and metadata queue for receiving application data and metadata from neighbouring cloudlets. For this work, RabbitMQ [21], a popular message queueing system, is used as the backend for the queueing service.

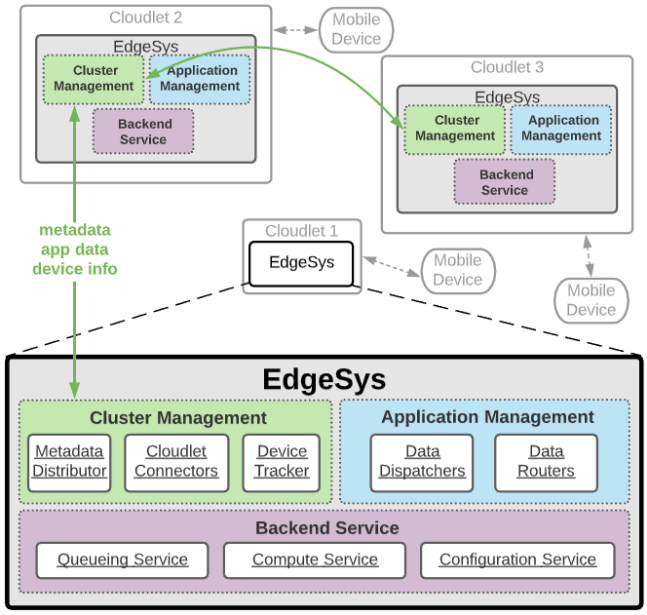

Figure 4: Decentralized EdgeSys Architecture: An EdgeSys cluster is formed by establishing ad-hoc connections with other neighbouring EdgeSys cloudlets. Each cloudlet makes independent routing decision for its application data based on information updated from neighbouring cloudlets.

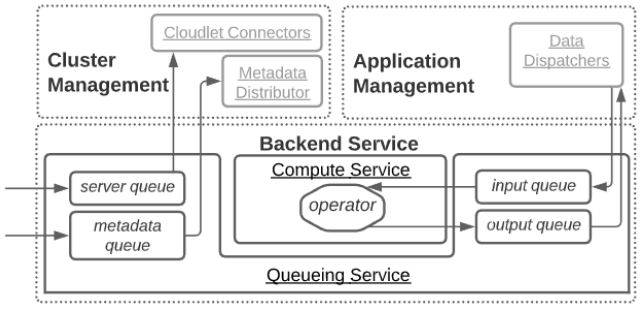

Figure 5: EdgeSys utilizes the queueing service to interface with application operators and other cloudlets

2) Compute Service: The compute service provides the runtime environment for application operators. EdgeSys uses Mesos [22] as the backend of the compute service. Mesos is a framework that organizes server(s) to provide a resource sharing platform for executing binaries and other computing frameworks. Marathon [23] is a management tool that handles application initialization and monitoring. EdgeSys uses Marathon to interface with Mesos.

3) Configuration Service: The configuration service provides the locations of the queueing and compute services for the cluster and application management. In the case where a cloudlet contains more than one physical servers, it is likely that different EdgeSys services are hosted in different servers within the cloudlet. With the configuration service, Any EdgeSys service can acquire the endpoints for other

EdgeSys services. Zookeeper [24], a cluster management tool, is used for running the configuration service.

—Application Management:

Application management is responsible for deploying application operators, routing application data to/from other cloudlets, and dispatching application data in/out of application operators.

1) Data Dispatchers: For each deployed operator in a cloudlet, in addition to the input/output data queues created for the operator, two Erlang processes [25] are created as the dedicated input and output dispatcher for that operator. As shown in Figure 6, the input dispatcher receives data from one of the data routers and send it to the corresponding input queue. As for the output dispatcher, it consumes the output data from the output queue and forwards the output data, depending on the data header, to either a data router for the next operator or the device tracker for returning the data to the client device.

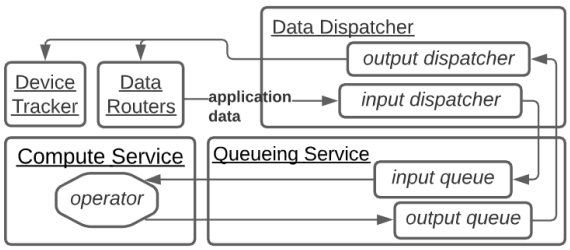

Figure 6: Data Dispatcher manages application data going in and out of the operator's queues

2) Data Routers: A data router of an application operator is responsible for forwarding data to its operator. There are two ways to create of a data router: One is when an application operator is deployed in a cloudlet, an Erlang process is spawned in that cloudlet as the dedicated data router for that operator. Another way is when a neighbouring cloudlet informs a cloudlet about a newly deployed remote operator, the informed cloudlet creates a dedicate data router for that remote operator.

Figure 7 shows the data path for a data router. There are three ways for application data to enter a data router. The first way is from a local output dispatcher, the second way is from locally connected device, and the lastly, from neighbouring cloudlet. Once a data router receives an input data from one of the three sources, the data router first checks if the corresponding operator is deployed in the current cloudlet. If it is, then the router sends the data to the corresponding local input dispatcher. If not, as shown in Figure 8, the data router requests the metadata distributor for the next-hop cloudlets that can reach the cloudlet with the desired operator. The metadata distributor returns a list of qualified next-hop cloudlets along with their hop counts to the target cloudlet. Next, the data router requests the connection status for the cloudlets in the list and removes any with broken connection. After the filtering, the data router selects the cloudlet with the minimal hop count. Finally, the data is sent to the selected next-hop cloudlet through the corresponding cloudlet connector.

—Cluster Management:

Cluster management is responsible for exchanging application operators' information with neighbouring cloudlets, establishing connections with neighbouring cloudlets, and tracking devices' location.

1) Metadata Distributor: Since there is no centralized controller in the EdgeSys framework, every cloudlet relies on its metadata distributor to obtains information about the cloudlet cluster. Metadata distributor's responsibilities are providing application operators' metadata to data routers and updating mobile devices' location to the device tracker. The metadata for application operator includes the hostname of the cloudlet that runs the operator, the application operator ID, the next-hop cloudlets that can reach the cloudlet with the operator, and the hop count to the cloudlet with the operator. The metadata for mobile device location includes the hostname of the cloudlet that is currently connected the device, the device ID, the next-hop cloudlets that can reach the connected cloudlet, and the hop count to reach the connected cloudlet. Since cloudlet cluster is a ad-hoc network, we adopt and modify Destination-Sequenced Distance-Vector Routing (DSDV) [26] as the routing protocol for EdgeSys's metadata exchange. We also utilizes split horizon [27] to reduce the number of data exchange between cloudlets. The goal of the original DSDV is for each node in the network to have a routing table to reach the nodes in the topology. But in EdgeSys, the routing table in each cloudlet serves as a look up table for locating where to reach the application operators or which cloudlet is connected with a particular mobile device. So each cloudlet exchanges information about available operators and mobile devices' location instead of the information about the cloudlet themselves. In a typical DSDV routing table, the entries includes "Destination Host", "Next-hop Host", "Metric", "Sequence Number", and "Install Time". For EdgeSys, we keep the same usage for most of the entries but we use hop count for "Metric", and for mobile device location routing, "Timestamp" is used instead of Sequence Number. We also add new entry, "Residing Operator ID", for application operator metadata routing, and "Connected Mobile Device ID" for mobile device location routing. Each cloudlet advertises information about its resident operators periodically to serves as the purpose of heartbeat. Every metadata distributor periodically deletes entries with expired heartbeat update. The advertisement for mobile device location is fired whenever a device makes new connection to a cloudlet. Metadata distributor uses MQTT [28], as the communication protocol.

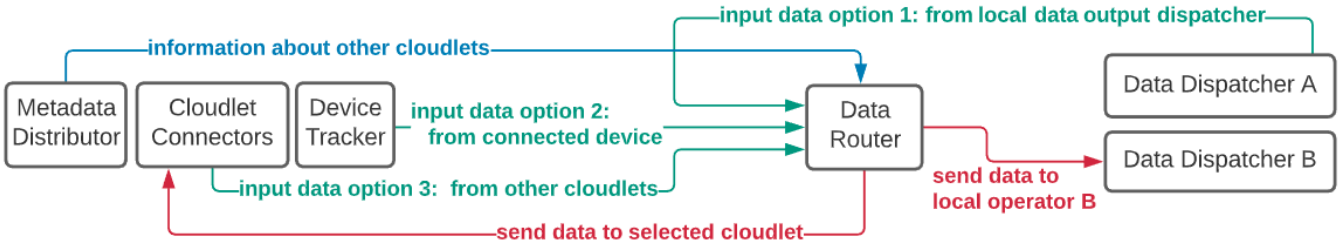

Figure 7: Data Routing in EdgeSys — A Data router can receive data from local output dispatchers, connected devices, or other cloudlets. Then the data router sends data to either selected cloudlet or local input dispatcher for processing

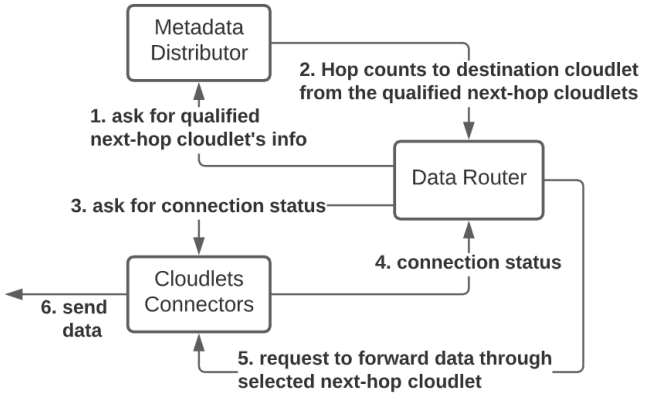

Figure 8: Selecting a neighbouring cloudlet to forward the application data based on connection status and hop count

2) Cloudlet Connectors: Cloudlet connectors are responsible for sending and receiving application data between two cloudlets. We use Figure 9 as an example to show how an EdgeSys connection is formed between two cloudlets. When Cloudlet Y wants to establish an EdgeSys connection with neighbouring Cloudlet X, Cloudlet Y sends a initiation request to Cloudlet X. Cloudlet X receives the request and creates a *Server-for-Y* queue to only serve application data coming from Cloudlet Y. In addition, a *Client-to-Y* and *Server-for-Y* connector, both Erlang processes, are spawned to handle application data transmission to/from Cloudlet Y. The *Client-to-Y* connector is responsible for sending data to Cloudlet Y and monitoring the connection status. The *Server-for-Y* connector is responsible for handling all the data delivered to the *Server-for-Y* queue.

After setting up the server queue and client/server connectors for Cloudlet Y, Cloudlet X informs metadata distributor on the new addition of the next-hop cloudlet Y. After that Cloudlet X sends the same initiation request to Cloudlet Y, which triggers the same steps described above to establish connection with Cloudlet X. The two connected cloudlets now start sending heartbeat to each other. If there is a timeout in heartbeat update from either side, it triggers the cloudlet to remove the timed-out cloudlet's server queue, server/client connectors, and related entries in the metadata distributor. A connection failure detected in a cloudlet client connector also trigger the same clean up process. It is crucial to remove failed connections and corresponding entries in the metadata distributor since we do not want to send application data through broken links. To further avoid data lost, every to be forwarded data first checks if the connection is still alive before sending it off to another cloudlet. If failure is detected, that data is sent back to the original data router to find the next best connected cloudlet.

Figure 9 also illustrates how a server connector handles data coming from other cloudlet. The data arrives at a server connector is either, based on the data header, forwarded to the device tracker for returning result data to the device, or sent to a data router. In this example, Cloudlet X is running application operator A and is connected to Cloudlet Y and Cloudlet Z. Application operator B is hosted in Cloudlet Z, and through the metadata distributor, Cloudlet X is informed about operator B's location and creates a data router for operator B. Now Cloudlet X receives three data from Cloudlet Y, indicated by three different colors, the blue data with destination to the device is sent to the device tracker, the red data with destination to operator B is forwarded to Cloudlet Z, and the green data with destination to operator A is delivered to local data input dispatcher A.

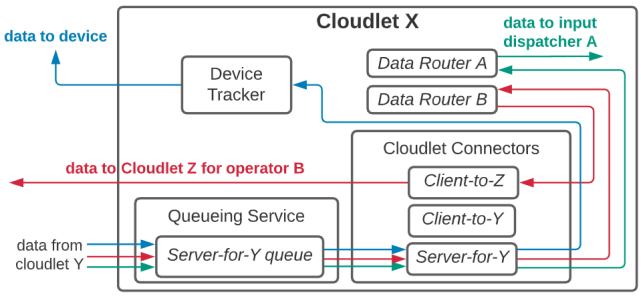

Figure 9: Cloudlets Connection through Cloudlet Connectors

3) Device Tracker: Device tracker is responsible for forwarding offloaded device data to data routers, tracking the location of the mobile devices, and delivering the result data back to devices. Inside device tracker, a UDP server is hosted to communicate with the devices. We assume each device has an unique ID, so whenever a device offloaded a data, device tracker records the device ID for the metadata dis-

tributor to advertise. Figure 10 illustrates a working device tracker. When the output dispatcher receives an output data and its header indicates to return to a device ID, the output dispatcher send the data to the device tracker. With the information from the metadata distributor, the device tracker either send the data to the locally connected device (blue data path), or forward the data to a neighbouring cloudlet that can reach the device (red data path).

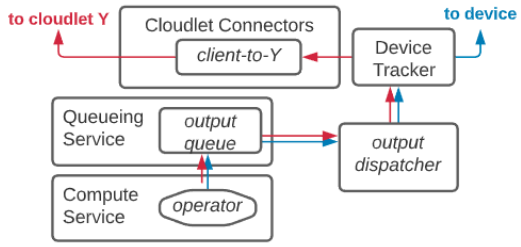

Figure 10: Device Tracker either returns the processed data to the locally connected device (blue) or forward the data to neighbouring cloudlet that can reach the device (red)

B. Requirements for Creating EdgeSys Application

For an application to be deployed under EdgeSys, there are a few requirements for the operators. First, each application operator should have a unique operator ID, which is used for identifying the operator in data routing, dispatching and metadata advertisement. Secondly, each application operator should consume input data from a RabbitMQ exchange with the operator ID as the routing key. As for the output data of the operator, it should publish to a RabbitMQ exchange with the operator ID as the routing key. If the output data is supposed to be delivered to another or multiple operators, that output data must include a header indicating the next operator ID(s). If the output data is supposed to be returned to the device, that output data must include a header indicating the device ID(s). Based on the header provided, the output data dispatcher either forward the data to the target data router(s) and/or to device tracker. As for the data offloaded by the mobile devices, the data should include header that indicates the target operator ID, so device tracker can forward the data to the target data router.

C. Heron API Support

As discussed previously, we adopt the Heron API as an alternative way to develop for EdgeSys. This allows one to gain the advantages of a decentralized, edge-focused processing framework while using a familiar interface. Furthermore, this allows the deployment of existing applications with minor modifications. Our compatibility layer works at two phases: application definition (*EdgeSysTopologyBuilder*) and application deployment (*EdgeSysExecutor*).

In the application definition phase, the changes for a developer are minimal. As seen in Figure 11, we overload the *TopologyBuilder* class. Our *EdgeSysTopologyBuilder* then receives the calls to *setSpout()* and *setBolt* operators. We then intercept the various stream definition calls as well. Finally, when the *createTopology()* method is called, the *EdgeSysTopologyBuilder* writes the information on the various stream processing instances and the connectivity graph to a file for the *EdgeSysExecutor* to use at runtime. It also generates serializations of the instances and a list of commands that the *EdgeSysExecutor* will use to launch the proper instances.

```
import com.twitter.heron.api.topology.TopologyBuilder;
import edgesys.util.EdgeSysTopologyBuilder;

...

public static void main(String[] args) throws Exception {

  // Old Heron topology builder
  // TopologyBuilder builder = new TopologyBuilder();
  // New EdgeSys topology builder
  TopologyBuilder builder = new EdgeSysTopologyBuilder();
```

Figure 11: Adapting a Heron topology to EdgeSys is a single-line change

In a normal Heron environment, the application (topology) would be launched through the Heron executable. However in EdgeSys, the instances are distributed across the cloudlets. EdgeSys uses the information generated by the *EdgeSysTopologyBuilder* to launch instances and handle the communication between those instances (Figure 12). Each *EdgeSysExecutor* runs one instance and manages the communication to and form EdgeSys' operator queues. The executor waits for data in the input queues, parses that data into a tuple format that is compatible with the operator, then passes it to the bolt instances through the *execute()* method. The executor also overloads the *OutputCollectors*. Whenever the bolt instances emit a data tuple, EdgeSys packages that into an EdgeSys data type and sends it to the output queue.

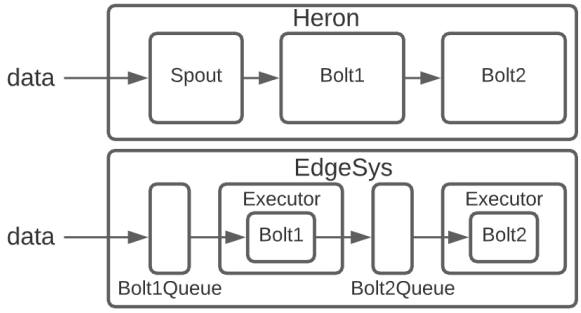

Figure 12: EdgeSys encapsulates Heron operators and manages communication between instances

Overall, EdgeSys is compatible with Heron applications with minimal changes. Developers have a single-line change to their topology definitions while deployments simply use the EdgeSys interface instead of the Heron executables. While EdgeSys is not restricted to Heron stream processing

applications, this level of compatibility can help speed up the deployment of existing applications on the edge.

IV. Results and Discussion

In this section, we first go over the specifications of the servers used in the experiments. After that, we present EdgeSys' performance results and analysis using two different applications.

A. System Setup

The servers used for the experiments are the HP ProLiant SL250s Gen8 Series. Each server has 16 Intel Xeon E5-2650L v2 processors and 64GB memory.

B. EdgeSys Heron — PageViewCount

In this experiment, we use the PageViewCount application from Intel's storm benchmark [12] to evaluate EdgeSys' support for Heron API. We set the PageViewCount application to have a total of 30 operators, as shown in Figure 13, the topology has one spout (data source) that emits data tuples to 14 pageViewBolt operators using shuffle grouping. In each pageViewBolt, the data tuples are filtered and sent to one of the 14 countBolt operators using field grouping. Finally, all count results are sent to printBolt operator which acts as a data sink. For each trial, we input 2000 data with 15 ms interval and collect each data's end-to-end latency which is defined as the time it takes for a data to reach the printBolt operator from the spout operator.

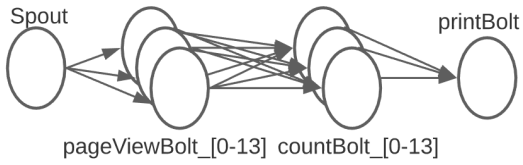

Figure 13: PageViewCount Application Topology

1) Methodology: Four different deployment scenarios are implemented in this experiment. In Scenario 1, we deploy the PageViewCount application using the standard Heron framework in a cluster of four fully connected servers. In Scenario 2, we reuse the same cluster in Scenario 1 but deploy the application using EdgeSys. In Scenario 3, the application is deployed in a 4-cloudlet cluster with grid topology connection as shown in Figure 14. Each cloudlet in the cluster contains one physical server. Input data arrives at Cloudlet 1. For Scenario 4, we deploy the application in the same 4-cloudlet cluster as in Scenario 3. But instead of having static data source, we simulate mobile data sources by constructing the following mobility pattern: at t = 0, the first mobile device enters the cloudlet network, starting from Cloudlet 1, the device offloads one data tuple, moves one hop clockwise and offloads another data tuple at t = 15 ms. Once the device makes 3 hops and offloaded 3 data tuples, the device leaves the cloudlet cluster and the next device enters the cluster and repeats the same steps. Figure 14 illustrates the mobility pattern for Scenario 4. In addition, we modify the application such that the output data of printBolt operator is delivered back to the source device. So for Scenario 4, the end-to-end latency is redefined as the time between the data arrives at a cloudlet to the result data received by the mobile device. Since the standard Heron framework requires fully connected cluster, running Scenario 3 and 4 are not possible using standard Heron deployment.

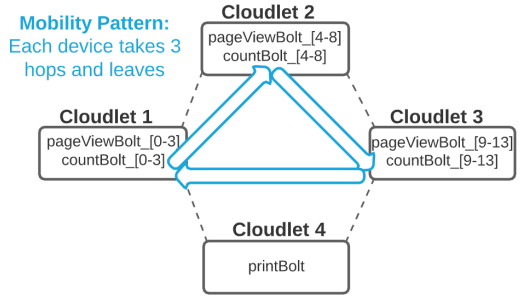

Figure 14: PageViewCount Application Deployment on Multi-cloudlet Cluster in Scenario 3 (with mobile devices offload data to different cloudlets in Scenario 4)

2) Result: Table I shows the end-to-end latency performance and Figure 15 shows the latency distribution for all four deployment scenarios. Since Scenario 1 and Scenario 2 have the same cluster setup. The performance difference between the two scenarios shows the baseline comparison between the standard Heron framework and EdgeSys Heron. The results show that the overhead added by EdgeSys is only 1.49 ms in average latency and 2.17 ms in tail latency. In Scenario 3, where the application is deployed in a 4-cloudlet cluster with grid topology, there is a increase of 4.74 ms in average latency compared to single cloudlet setup in Scenario 2. The increase of latency for Scenario 3 is expected because the operators are scattered across the cluster, requiring intermediate data to travel between cloudlets. In Scenario 4, there is another average latency increase of 2.52 ms compared with Scenario 3 even though both scenarios have the same cloudlet cluster setup. The increase is also expected because Scenario 4 has additional data path of delivering result data back to the mobile data source.

Table I: PageViewApp Timing Performance

	Min	Max	Average Latency
Scenario 1	5.70 ms	11.60 ms	7.97 ms
Scenario 2	7.76 ms	13.77 ms	9.46 ms
Scenario 3	9.00 ms	21.84 ms	14.20 ms
Scenario 4	10.93 ms	25.07 ms	16.72 ms

In this experiment, we successfully show that Heron application deployment using EdgeSys does not require

a fully connected cluster and can return result data to the mobile data source, which are not possible using the standard Heron framework. We also show that under cloud-like setting with fully connected cluster, Heron application deployed using EdgeSys on average only incurs 1.49 ms overhead comparing to using the standard Heron framework.

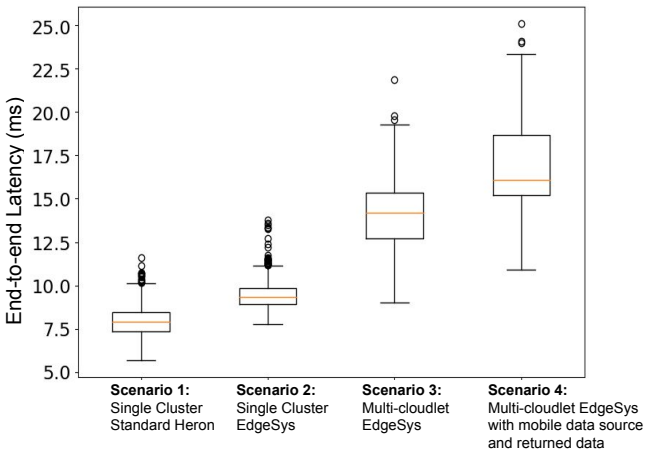

Figure 15: End-to-end Latency Performance under Different Deployment Scenarios

C. Example Edge Application — Multi-agent Path Finding

In this experiment we implement Multi-agent Path Finding (MAPF) application using EdgeSys and demonstrate its ability to maintain application operation under dynamic cloudlet connection.

A MAPF problem can be described as follows: Given a map and a set of agents with starting and goal positions, a MAPF solver outputs a MAPF plan containing collision-free paths for all agents with the objective of minimizing total travel time. A path for an agent contains an ordered list of intermediate destinations. To execute a given MAPF plan, MAPF-POST [29], an online plan-execution scheduler, is used to ensures the plan execution does not deviate from the MAPF plan by periodically gathers information about the agents' location and recalculates the proper arrival times to the next intermediate positions for all agents. Figure 16 shows a high level view on MAPF plan execution with MAPF-POST. During initialization, a MAPF plan is given to the MAPF-POST operator, it then generates the first plan-execution schedule which includes lists of intermediate destinations with arrival times for the agents. The schedule is delivered to the monitor and director operator. During execution, as shown in the blue data path, each agent requests the director for the speed and direction the agent should travel during the next time interval. The director uses the plan-execution schedule to calculate the response. Concurrently, as shown in the green data path, the monitor waits to receive all the position update from the agents and reports the timestamp and accumulated position update to the MAPF-POST operator to recalculate the plan-execution schedule.

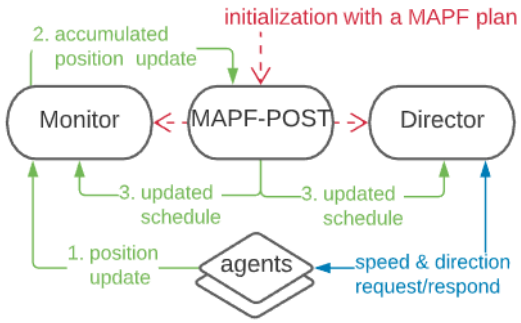

Figure 16: Data flow for MAPF Application Execution

Figure 16 shows that the MAPF application contains three operators with two data paths. Based on this topology, we develop a MAPF implementation and run it on EdgeSys. For the map and agents setup, we use one of the grid-based maps and 50 randomly generated agents from the MAPF Benchmark Sets [30]. We place 16 single-server cloudlets in the map as shown in Figure 17. In Figure 17, the blue dots denote the agents, the green labels mark the starting positions, the red labels mark the goal positions, and the black solid squares indicate the road blocks. The cloudlets are represented by the yellow solid squares and the yellow lines mark the communication boundaries for each cloudlet. The blue lines coming out of the agents indicate their current connected cloudlet. As an agent moves across the yellow lines, the agent establishes new connection with the nearest cloudlet. We use ECBS [31], a MAPF solver, to generate a MAPF plan for the MAPF-POST operator.

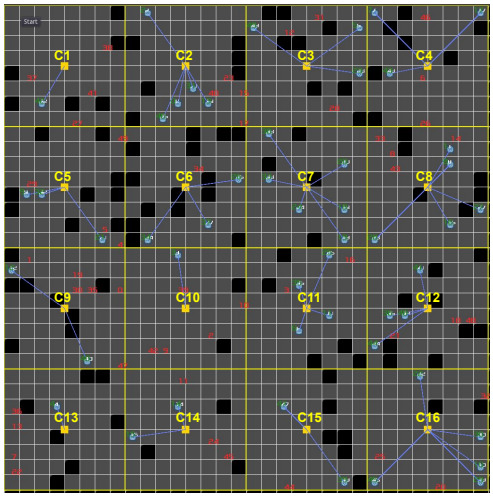

Figure 17: Graphical Interface of MAPF Application Implementation on 16-cloudlet EdgeSys Cluster

1) Methodology: The 16 cloudlets shown in Figure 17 are connected in a grid topology. The monitor, MAPF-POST,

and director operator are deployed in Cloudlet 6, 7, and 10 respectively. Each agent sends position update to the monitor operator every 1000 ms. Each agent concurrently also sends speed and direction request to the director operator every 100 ms. If it takes more than 100ms for the response of the director operator to reach a agent, the agent is stalled and can potentially break the entire operation. To test EdgeSys' ability in handling dynamic cloudlet connection, 4 different network scenarios, shown in Figure 18, are implemented during the MAPF application execution. The application starts with a healthy grid topology, and for every 10 seconds, the network topology changes in the following pattern: healthy grid $\rightarrow$ Scenario A $\rightarrow$ healthy grid $\rightarrow$ Scenario B $\rightarrow$ healthy grid $\rightarrow$ Scenario C $\rightarrow$ healthy grid $\rightarrow$ Scenario D $\rightarrow$ healthy grid. Toxiproxy [32] is used to simulate cloudlet connection failure and recovery. During the MAPF execution operation, the round trip times (RTT) and the round trip hop count for every speed and direction request/response are recorded for analysis.

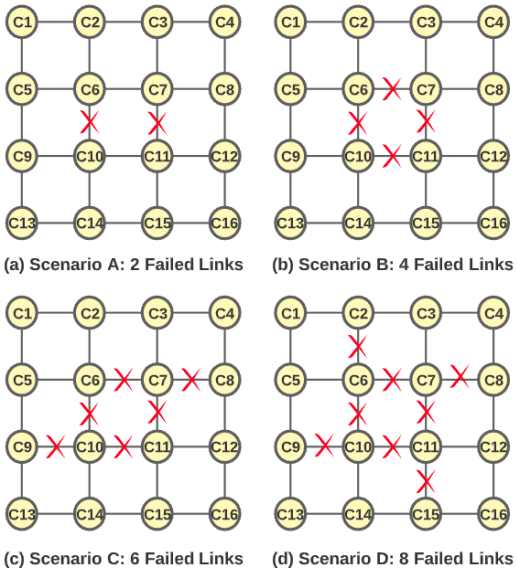

Figure 18: Different Network Scenarios for Simulating Dynamic Cloudlet Connection

2) Result: Figure 19 shows the RTTs (blue) and round trip hop counts (red) for the entire MAPF plan execution and Table II shows the average RTT under each network scenario. Every reported RTT is under 100 ms, which indicates that no deadline is missed and no agent was stalled during the MAPF plan execution. Figure 19 shows that scenario with more failed links results in higher hop count. For example, in a healthy grid network topology, the most number of hop to reach Cloudlet 10 (where the director operator resides) from another cloudlet is 4 hops, so the maximum round trip hop count is 8. But with Scenario C or D, the most number of hop to reach Cloudlet 10 from another cloudlet increases to 8, which results in round trip hop count to 16, as shown in Figure 19. Table II shows that scenario with more failed links results in higher average RTT which is expected as higher number of failed connections implies higher chance for a data to take a longer route to reach the director operator in Cloudlet 10. This experiment shows that EdgeSys is able to continue application execution despite having unstable network conditions.

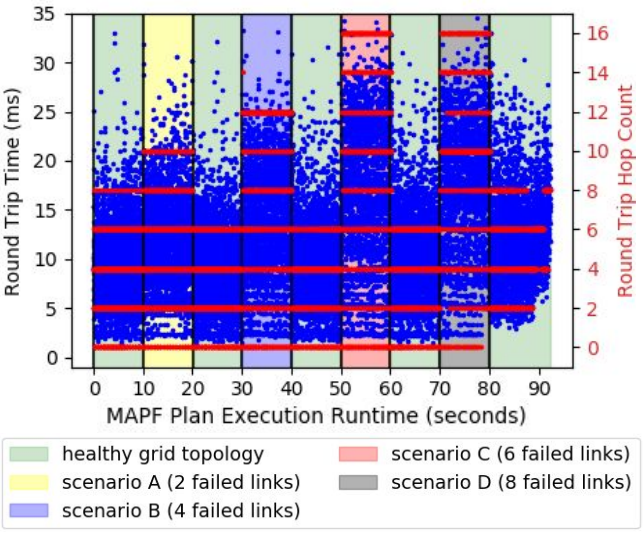

Figure 19: MAPF Application Timing Performance under Dynamic Network Connection

Table II: Average RTT for Different Network Scenarios

	Average RTT (ms)
Healthy grid netowrk topology	8.18
Scenario A (2 failed links)	9.77
Scenario B (4 failed links)	11.25
Scenario C (6 failed links)	14.49
Scenario D (8 failed links)	15.72

In this section, through two applications, we show EdgeSys' ability to run Heron application, support mobile devices, return processed data back to the devices, and maintain application execution in dynamic network scenarios.

V. Conclusion and Future Work

In this work, we develop EdgeSys, a decentralized edge computing framework for running application with mobile devices in a dynamic cloudlet cluster. We plan to further optimize the algorithm for selecting the best next-hop cloudlet in data routing by considering more parameters such as connections' bandwidth and latency. In addition, in the current EdgeSys, we only consider the case when each cloudlet still has at least one connection to the cluster. In the future, We plan to utilize redundancy technique [33] on application operators and data to achieve better network and server fault tolerance.

REFERENCES

[1] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," *SIGPLAN Not.*, vol. 53, no. 2, p. 751–766, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173191

[2] CloudHarmony. [Online]. Available: http://cloudharmony.com/

[3] Ripe atlas. [Online]. Available: https://atlas.ripe.net/

[4] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, p. 30–39, Jan. 2017. [Online]. Available: https://doi.org/10.1109/MC.2017.9

[5] Open Edge Computing Initiative. [Online]. Available: https://www.openedgecomputing.org/about

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, N. Tatbul, S. Zdonik, and M. Stonebraker, "Chapter 20 - monitoring streams — a new class of data management applications," in *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, P. A. Bernstein, Y. E. Ioannidis, R. Ramakrishnan, and D. Papadias, Eds. San Francisco: Morgan Kaufmann, 2002, pp. 215 – 226. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781558608696500275

[7] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," vol. 34, no. 4, 2005. [Online]. Available: https://doi.org/10.1145/1107499.1107504

[8] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788

[9] A. Katsifodimos and S. Schelter, "Apache flink: Stream analytics at scale," in *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, April 2016, pp. 193–193.

[10] Samza. [Online]. Available: http://samza.apache.org/

[11] Heron topology concepts. [Online]. Available: http://heron.incubator.apache.org/docs/heron-topology-concepts

[12] Intel-hadoop storm-benchmark. [Online]. Available: https://github.com/intel-hadoop/storm-benchmark

[13] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 929–946. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fu

[14] M. Chao and R. Stoleru, "R-mstorm: A resilient mobile stream processing system for dynamic edge networks," in *2020 IEEE International Conference on Fog Computing (ICFC)*, 2020, pp. 64–72.

[15] B. Amento, B. Balasubramanian, R. J. Hall, K. Joshi, G. Jung, and K. H. Purdy, "Focusstack: Orchestrating edge clouds using location-based focus of attention," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2016, pp. 179–191.

[16] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, "Enorm: A framework for edge node resource management," *IEEE Transactions on Services Computing*, pp. 1–1, 2017.

[17] A. Javed, J. Robert, K. Heljanko, and K. Främling, "Iotef: A federated edge-cloud architecture for fault-tolerant iot applications," *Journal of Grid Computing*, vol. 18, pp. 57–80, 2020.

[18] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 813–818.

[19] Kubernetes.io. [Online]. Available: https://kubernetes.io/

[20] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," 2011.

[21] Rabbitmq. [Online]. Available: http://www.rabbitmq.com/

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[23] Marathon. [Online]. Available: https://mesosphere.github.io/marathon/

[24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851

[25] Erlang process. [Online]. Available: https://erlang.org/doc/reference_manual/processes.html

[26] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, p. 234–244, Oct. 1994. [Online]. Available: https://doi.org/10.1145/190809.190336

[27] A. Schmid, O. Kandel, and C. Steigner, "Avoiding counting to infinity in distance vector routing," in *Proceedings of the First International Conference on Networking-Part 1*, ser. ICN '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 657–672.

[28] MQTT. [Online]. Available: http://mqtt.org/

[29] W. Hönig, T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig, "Multi-agent path finding with kinematic contraints," in *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, 2016, pp. 477–485, outstanding paper award in the robotics track. [Online]. Available: https://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13183/12711

[30] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *Symposium on Combinatorial Search (SoCS)*, pp. 151–158, 2019.

[31] M. Barer, G. Sharon, R. Stern, and A. Felner, "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem," in *Seventh Annual Symposium on Combinatorial Search*. Citeseer, 2014.

[32] Toxiproxy. [Online]. Available: https://github.com/Shopify/toxiproxyl

[33] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, no. 1, p. 1–26, Mar. 1999. [Online]. Available: https://doi.org/10.1145/311531.311532