REC: Resilient Edge Computing in Emergency Scenarios

Authors removed for review

Abstract—**This paper introduces REC, a novel resilient edge computing platform designed for emergency scenarios where communication links to centralized cloud infrastructure may be unreliable or unavailable. REC combines the convenience of serverless computing with resilience and locality, allowing diverse parties to join their computing resources into a distributed, local network to support recovery efforts. REC utilizes WebAssembly for platform-agnostic code execution and implements a distributed key-value store for data persistence. It consists of multiple node types including brokers, executors, datastores, and clients that work together to orchestrate job execution and data management. The key features of REC include self-healing capabilities, job parallelization, and a flexible node discovery mechanism. Our experimental evaluation shows that, while there is some performance overhead compared to native execution, REC enables parallel processing that can provide significant speedups.**

I. Introduction

The increasing global instability, attributed to natural disasters and international conflicts, requires the development of novel, distributed, and resilient alternatives to established centralized computing solutions. Traditionally, computing services have been provided by companies operating large centralized data centers, which efficiently consolidate resources but are susceptible to single-point-of-failure vulnerabilities.

Resilient distributed systems offer a potential mitigation strategy by dispersing resources across a wider geographical area and maintaining operational continuity even when certain components become unavailable. Two prominent examples of such systems are *serverless computing* and *function-as-a-service* (FaaS), which have gained popularity due to their ability to abstract complex server administration and software deployment processes, enabling developers to deploy code rapidly and with minimal effort.

While major service providers such as *Amazon AWS* and *Google Compute Cloud* leverage their extensive data center networks to offer these services at competitive costs, the distribution of data centers globally provides only macro-level redundancy (e.g., deployment options in North America, Europe, Asia). However, this approach does not address local resilience issues. In scenarios where the local communication infrastructure is compromised, these services become inaccessible due to the absence of data centers in all cities.

During major emergencies such as flooding, earthquakes, or wildfires, populations can experience severe disruptions in Internet connectivity, effectively isolating them from larger networks. The absence of uplinks to data centers makes centralized services unavailable, forcing affected populations to rely solely on local resources until external communications can be restored.

To address this critical issue, we propose REC, a novel and resilient edge computing platform that adapts serverless computing paradigms to local ad hoc networks. REC provides an abstraction layer for code deployment and data storage, enabling rapid code deployment similar to traditional FaaS offerings. REC is designed as a distributed architecture, featuring nodes spread throughout the network, and provides resilience by maintaining operational continuity even in the event of node failures or network partitioning. REC operates without a central controlling entity, allowing various parties to manage cooperating nodes. Instead, different parties can operate cooperative nodes that allow the population affected by the crisis to share their computing resources efficiently.

REC enables users to dispatch computational jobs to the network without requiring knowledge of the underlying network topology. The system manages data storage and retrieval, automatically providing necessary data to jobs and storing job outputs for subsequent use. We have developed a proof-of-concept implementation of REC to demonstrate the real-world feasibility of our approach. This prototype runs on actual hardware and is deployable across various computing platforms. We evaluated our prototype in an emulated network using the CORE emulator[1]. Our experimental evaluation shows that while there is some performance overhead compared to native execution, REC enables distributed parallel processing that can provide significant speedups.

The paper is organized as follows. Section II discusses related work. Section III presents the design of REC, while Section IV is devoted to implementation issues. In Section V, we evaluate REC experimentally with respect to different metrics. Section VI concludes the paper and outlines areas of future work.

II. Related Work

Serverless computing has traditionally been focused exclusively on large centralized data centers, but there have been some attempts to move serverless functionality to the network edge.

For example, Aske and Zhao [1] introduced their framework, named *MPSC*, which can schedule jobs to several serverless-computing providers simultaneously. Their work primarily focuses on achieving performance gains by using multiple computing providers in parallel. Thus, it relies on

[1]https://github.com/coreemu/core

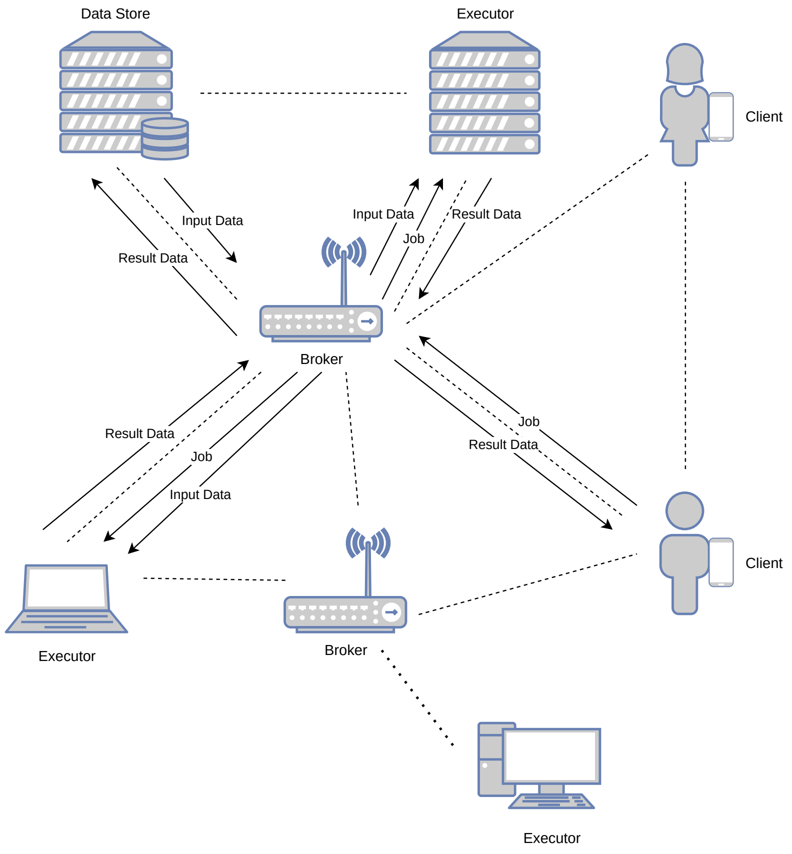

Fig. 1. Possible city network

existing, centralized cloud providers and does not include any notion of resilience.

Baresi and Mendonça [2] presented an approach similar to REC. Their system can deploy functions either to remote FaaS providers or to local fog nodes. While this work focuses on efficient usage of local resources, the primary goals are performance and latency optimizations. In addition, the system relies on a single central controller. This makes it unsuitable for use in an unreliable network, as a single node failure could lead to the collapse of the entire system.

Gadepalli et al. [3] leveraged the advantages of WebAssembly for edge computing. Their system, named *Sledge*, is a WebAssembly runtime optimized for edge computing workloads. Although thematically close to our work, the authors focus exclusively on optimizing local runtime and do not cover networking, deployment, or resilience.

With OPPLOAD, Sterz et al. [4] presented a system similar to REC to provide computation capabilities at the edge of the network during emergencies or events in which a centralized cloud infrastructure is no longer reachable or available. However, the authors rely on citizens' mobile phones as target execution platforms and do not use a platform-agnostic execution platform like WebAssembly but rely on platform-specific implementations of functions. Finally, the functions must be deployed before the event, since OPPLOAD cannot push the function code itself to the executor.

rAFCL is a SaaS middleware designed by Ristov et al. [5]. It attempts to provide resilience and federation to SaaS computing. rAFCL allows users to create *function choreographies* (FCs) that combine several SaaS deployments to build a more complex application. However, in their context, *resilience* means that a FC can withstand the failure of a component function (e.g., due to rate-limiting by the provider), and re-run the function somewhere else. This notion of resilience is related to ours, but it is not exactly the same. Additionally, this work focuses only on improving the usability of commercial SaaS offerings and thus fundamentally depends on the centralized data center model, which we are trying to avoid.

III. Design

In an emergency scenario, the local network hosting REC may be volatile. Various emergencies may damage the existing backbone network or render it unavailable. Floods, earthquakes, or fires may destroy infrastructure, while a mass panic might lead to extended overloading. Nodes may join or leave the network for various reasons, such as mobile nodes moving in and out of range or devices on backup power being forced to shut down. Therefore, REC must be able to deal with changing network topologies.

For this purpose, we have created four distinct *node types* that each contribute distinct abilities to the network. There might be multiple instances of each node type in the network, all hosted on network participants' existing hardware. Figure 1 shows such a network as it might exist in a typical city.

REC is designed to bring together a variety of heterogeneous parties to aggregate and share computing resources in the city. Parties might be institutions with access to "big iron" hardware (e.g., database servers and compute servers) or individuals who contribute their personal hardware to the network.

A. Nodes

Figure 2 shows the different node types and the messages exchanged during operation. A single device may host multiple node types.

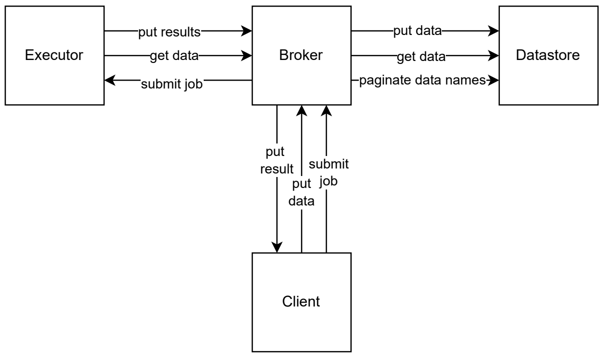

Fig. 2. The architecture of REC

1) Broker: Broker nodes manage all other nodes and serve as connections between them. While this is somewhat similar to a centralized controller, several brokers should be active in the network simultaneously so that a single node failure will not cause catastrophic disruption. Each broker may have several connected datastores and can forward, store, and retrieve requests for the other nodes. Because REC is designed for mesh networks where not all nodes might be directly connected, data transfers have to go through a broker. Brokers also manage executors and are responsible for selecting the executor for a submitted job. Brokers should be placed on highly connected nodes, such as machines in the local network backbone or interconnected mesh routers.

Since broker nodes orchestrate the network, their loss is particularly impactful. Since REC is designed to run with multiple broker nodes, a broker does not represent a single point of failure, but its associated executors and datastores will be cut off from the network. In this case, there are the following options:

- If the broker rejoins before a defined timeout is reached, REC continues as if nothing has happened. In the meantime, client nodes will be unable to access job results.
- If the broker leaves permanently (or at least for longer than the timeout threshold), but at least one broker remains present in the network, the orphaned nodes may join another broker. However, metadata held on the disappeared broker will be lost.
- If no other broker is present in the network, a node will need to be (self-)promoted to become the new broker.
- Should the broker rejoin the network after the timeout period, it will continue operating as a broker but not have any associated nodes.

2) Executor: The executor nodes execute WebAssembly code. Each executor is managed by one broker. When a job is submitted, all relevant data is pulled from the datastores, the job is executed, and the result is sent to either a datastore or a client. If a participant has access to powerful hardware, they should consider running an executor node on it. Possible operators of executor nodes might be local institutions with big-iron servers or individuals with powerful hardware (such as a gaming desktop).

3) Datastore: The datastore nodes store all the data required to execute the jobs and store their results. The datastore allows jobs that are not immediately executable (due to the lack of a capable executor) to be stored and queued later. This way, a client may submit a job and then go offline. Once the client reconnects, it can query for the job's result. Data is accessible through names, e.g., name/example. Data specific to a job starts with this job's UUID, followed by a slash. Since job results are available as named data, they can be used as the input for another job, thus enabling pipelining. Datastores are known by and accessed through brokers. If a participant has access to large amounts of storage, it should consider running a datastore node. Possible operators of datastore nodes might be institutions that operate a local data center or individuals willing to share capacity on their home NAS.

4) Client: The client node is used to interact with REC. For this purpose, the client software runs on the user's hardware, allowing the user to submit jobs and retrieve results. When a new client instance starts, it searches the local network for an existing broker instance. If multiple brokers are found, the client is free to choose any one of them; a randomized choice may be beneficial for load-balancing purposes.

Should the chosen broker disappear, either because the broker node has gone down or the network has partitioned, the client will have to find another broker to connect to. In case of a network partition, both the previous broker and executor may be in the now unreachable network partition, in which case the client will have to resubmit its job.

B. Jobs

Since jobs should be platform-/architecture-agnostic, but still make efficient use of computing resources, we use WebAssembly as our execution environment. WebAssembly programs can be written in several high-level programming languages[2] and are compiled to WebAssembly bytecode.

[2]Compilation is handled by LLVM, so in theory, any language with an LLVM front end can be used.

The execution platform must run a *WebAssembly runtime* to execute the bytecode. In this sense, WebAssembly is not dissimilar to Java or Scala, which are compiled to bytecode and rely on the *jvm* for execution. The main difference between WebAssembly and previous "write once - run anywhere" approaches is WebAssembly's focus on performance [6] and security, as well as its language independence. Since WebAssembly was originally designed to replace client-side JavaScript executed in a browser, it is supported by all major web browsers that include their own WebAssembly runtimes. However, there exist plenty of standalone runtimes written in various languages with varying feature sets. While WebAssembly applications are generally sandboxed for security reasons, there is the *WebAssembly System Interface (WASI)*[3] with which a program can interact with system resources in a well-defined manner. Each WASI-API provides code running within the VM with a limited set of actions to interact with the outside world, similar to *system calls*, while maintaining the sandbox's security posture. Jobs may require an executor to have additional *capabilities* to execute the job, such as, for example, the presence of a GPU. In these cases, the job may access additional hardware through a WASI-API. It should be noted, however, that many WASI-APIs are still in the early stages of standardization.

C. Data

When designing the data model, we had to make a choice between several approaches. The most straightforward approach would have been for the client to supply all data necessary for the job as part of its deployment request. The broker would forward the data to the executor, and then the result would be returned to the client. While this might be the obvious approach, it does have some inefficiencies where multiple jobs might depend on the same data. Instead, we designed REC based on the principles of *named data networking* [7]. For this reason, we added the *datastore* node type; datastore nodes collectively provide data persistence inside the network. This data is organized in a distributed key-value store, where each piece of data has a unique name. Names are organized hierarchically, and each key may have an arbitrary number of subkeys. Queuing a key returns its content and the content of all subkeys. For example, data from temperature sensors could be stored as `/temperature/<city quarter>/<street name>/<bulding no>/<timestamp>`, allowing data to be queried at different aggregation levels. Each job can specify an arbitrary list of arguments in the form of keys. We then retrieve the data for the specified names and provide it to the job for execution. Importantly, the job creator does not need to know where data is located, instead the job creator just needs to specify the name, and REC will take care of discovery and transmission. The result of a job is stored as a key-value pair in the network. Not only does this allow for multiple jobs to use the same data, the data may also be pre-submitted for later use, or one can build job *pipelines*, where several jobs may depend on the output of a previous job.

[3]https://github.com/WebAssembly/WASI

D. Job Submission

To get executed, a job needs to run through the following steps (see Figure 2 for an illustration of the data flows):

- First, all the necessary data is uploaded into the datastores.
- The JobInfo is submitted to the connected broker.
- The broker then chooses an executor based on the capabilities needed.
- The JobInfo is submitted to the executor that starts populating the job's directory.
- As soon as the job's directory is set up, the job is executed.
- If the broker currently does not know any executor capable of executing the job, the broker can store the job until an executor becomes available.

E. Node Discovery

Since REC is inherently a peer-to-peer system, finding peers to connect to is a major challenge for setup. When a new REC node starts, it has no knowledge about the locations/addresses of REC nodes it could connect to or if there even is a compute network present in its local area network. To allow a network to form and new nodes to join it, we need some form of peer discovery. When a new node enters the local area network (LAN), it needs to check whether at least one broker is present. If this is the case, the node *may* connect to this broker as a client, executor, or datastore, respectively. To increase the network's resilience, it might be helpful for a node to (also) start a broker instance since we want to always have multiple brokers to avoid a single-point-of-failure, even after network partitions.

IV. Implementation

Our proof-of-concept implementation is written in Python. Since the WebAssembly runtime *wasmtime*[4] runs only on x86-64 machines, the implementation is currently limited to this architecture. While other WebAssembly runtimes target other processor architectures, we use wasmtime for our prototype since it is one of the most mature implementations. There is no reason why REC should not work with other WebAssembly implementations. All our code is published under a permissive open-source license.[5]

A. Code Execution

The WebAssembly code is executed using the wasmtime runtime. The wasmtime runtime is written in Rust and is accessed through Python bindings. *wasmtime* was chosen because it compiles the WebAssembly code to native code before execution, leading to fast execution. It also allows for easily setting stdin and capturing stdout/stderr. The jobs are run in different processes, thus they do not interfere with one another or the communication part of the code.

[4]https://wasmtime.dev/

[5]Link to be inserted after the end of review process.

B. Jobs

Each job is described by a *JobInfo* structure that contains all the information needed to orchestrate the execution of the job. This includes data names (e.g., the executable, an image or text file used as input), arguments and environment variables, instructions on how to handle the results and the capabilities needed to run the job, and a hint that can be used to help in executor selection. This definition is packed with associated data in a zip-archive (called a *bundle*) and uploaded to a broker.

```
class JobInfo:
    wasm_bin:           Union[str,
     ↪  tuple[str, str]]
    stdin:              Union[str,
     ↪  tuple[str, str]]
    job_data:           dict[str, str]
    named_data:         dict[str, str]
    directories:        list[str]
    args:               list[str]
    env:                dict[str, str]
    zip_results:        dict[str, str]
    named_results:      dict[str, str]
    capabilities:       Capabilities
    result_node_id:     str
```

Listing 1: Job metadata

The (slightly abbreviated) metadata definition can be found in Listing 1.

- `wasm_bin`: WebAssembly executable. This may be the name of an existing executable in a datastore. Otherwise, the job can also provide its executable as part of the job bundle.
- `stdin`: File whose contents will be supplied to the program via `stdin`.
- `job_data`: Additional data uploaded as part of the bundle.
- `named_data`: Additional data to be fetched by the network before execution.
- `directories`: If the program needs a directory structure to be present inside the sandbox to run, these can be specified here.
- `args`: Command-line argument supplied to the program at startup.
- `env`: Environment variables.
- `zip_results`: Results from the program, packed in a zip archive; will be sent back to the client.
- `named_results`: Results of program; will be stored in a datastore. This may allow other jobs to use this data as input.
- `capabilities`: Capabilities necessary for program execution.
- `result_node_id`: Client's node ID where the results archive will be sent. The client must remain available at the given ID to receive the results. If it is not available, the return will fail. If *named_result* was set, then the client can get a copy from a datastore later.

```
class Capabilities:
    memory:       int
    disk:         int
    cpu_load:     float
    cpu_cores:    int
    cpu_freq:     float
    has_battery: bool
    power:        float
```

Listing 2: Job capabilities

The (slightly abbreviated) capability definition can be found in Listing 2. Most of the requirements should be pretty self-explanatory. However, this is merely a set of sample capabilities for our proof-of-concept implementation. It is possible (and intended) to further extend these capabilities.

C. Data

In REC, data is made available to executors and clients through the datastores. Each piece of data must be given a unique name. When submitting a new job, a client may specify the necessary input data by its name. In addition, if the job needs data that is not yet present in the network, or if the submitter prefers it to not be made available, it can be included directly in the job bundle.

1) Results: The result of a job is a zip file containing files and directories that the client can specify on job submission. Both `stderr` and `stdout` are captured into files and included in the results archive. The zip file can either be sent to a datastore for later retrieval or directly to the requesting client. Besides the result zip, a job may also publish files datastores under a name. The files included in the zip and the published files do not need to be identical. This is primarily done so that jobs can use the results of other jobs as input.

2) Job Directory: To run multiple jobs in parallel, each job has its own directory where all the necessary data is stored and all results will be written. This means that a path to the storage location of each piece of data used in the job must be specified. The jobs cannot access data outside this directory since the WebAssembly runtime is set up to allow only the WebAssembly code to access this directory (and its sub-directories). The stdin stream is just a file in the directory, which is piped into the WebAssembly runtime.

D. Node Communication

Since REC is meant to run on ad-hoc local networks, we have to account for highly variable networkt topologies. In such emergency ad-hoc networks, links may disappear and reappear at unpredictable times due to outside circumstances. To increase the reliability of inter-node communication, we use the *Bundle Protocol* [8]. In particular, we chose to use the bundle-protocol implementation of the *dtn7* software-project [9], [10].

If job data is included in the job bundle, it is simply forwarded to the executor during deployment. For additional named data required for the job, the broker knows which associated datastore (if any) the data is located in. The broker will then query the data from the datastore and forward it to the executor. Once the job has finished, the results are sent back to the broker. If the `named_results` variable was set, the broker will forward the result data to a datastore. If the `result_node_id` variable was set, the broker will try to forward the result to the client.

To simplify the extension of the REC platform, we have implemented a REST inspired abstraction on top of the DTN daemon. Custom endpoints can be registered with a DTN server to listen for requests from a DTN client, and where appropriate, a DTN client will automatically wait for a response to a particular request.

```
class DtnRequest:
    method:          DtnRequestMethod
    path:            str
    request_id:      str
    content:         Any
```

Listing 3: DTN request from a DTN client to a DTN server

```
class DtnResponse:
    request_id:      str
    content:         Any
    status_code:     int
```

Listing 4: DTN response from a DTN server to a DTN client

The request and response structures are shown in Listing 3 and Listing 4.

- `method`: Request method similar to HTTP. `GET`, `PUT` and `DELETE` are currently supported.
- `path`: Endpoint. Describes the purpose of a request, e.g. `/job/submit`.
- `request_id`: UUID of a request. Used to match responses to a request.
- `content`: The job's executable.
- `status_code`: Similar to HTTP, describes whether a request was successful, and if not, what kind of problem occurred.

E. Broker

The broker remembers the job's id and result address to send results directly back to a client. When the executor finishes execution, it sends the job's results to the broker, who can forward it to the client. Should the client be unreachable, the result is stored in the datastores instead.

All data is accessed through a broker. Since brokers can know multiple datastores, each broker has to ask its datastores whether they have the needed data until it is found or all have been asked. To minimize communication, the brokers cache which datastore they got which data from.

F. Datastores

The datastores store the data in files with the path specified by name in their root directory. Data can be retrieved directly, throwing an error if it is not available. Otherwise, datastores can also return a list of all their data. It is also possible to paginate the data. For pagination, it is also possible to specify a prefix or a job ID to filter for the correct name. This is, for example, used in the broker's cache to get all data a datastore holds for a specific job.

G. Executor

The executors have capabilities associated with them. They may include free storage, free memory, CPU load, whether the device has a battery, and the battery's charge percentage. When trying to deploy a job to an executor, the broker attempts to match the job's capabilities to the capabilities of available executors. The brokers receive the capabilities periodically from the executors.

H. Client

Jobs are specified to the client through an execution plan. The execution plan allows the specification of multiple jobs and the order in which they are to be executed. It is possible to have jobs run in parallel or have them wait for each other. The execution plan also allows the uploading of non-job named data to the network. This could, for example, be an executable that multiple jobs use, just with different data. Jobs are given by name and a JobInfo structure. Each time a job is submitted, it gets a UUID that is randomly generated by the client.

V. Experimental Evaluation

In this section, we present the results of our experimental evaluation. As an example workload for our evaluation, we selected a machine learning-based audio classification algorithm that has been trained to recognize more than 70 ”urban” sounds, ranging from dogs barking to music and sirens. Both the algorithm and the dataset were provided by Bellafkir et al. [11] The algorithm needs frequency-domain values to function. To obtain these, a preparation job is run first, which handles the transformation from compressed audio to frequency values. These are then passed to the audio classification program.

This workload was chosen to give a rough approximation of a useful task in an urban emergency scenario, as recognizing and classifying urban sound profiles might be helpful to first responders. For example, the audio classification algorithm can differentiate between human speech, laughing, and screaming, thus a location with many screaming voices may warrant investigation. However, this only acts as an example for other workloads, since REC has been specifically designed to run arbitrary code.

The dataset that we ran the algorithm on is a library of 13,812 short audio samples. For sequential runs, all samples were concatenated into one long audio stream. For parallel runs, this was split into 12 files, each containing 1,151 audio samples. The classifier was originally implemented in C and designed for embedded systems. For our evaluation, we ran the classifier in three configurations.

In the first scenario, the code is compiled to an x86-64 binary, which runs natively on our evaluation hardware. This is likely the most efficient way to run the code and will serve as the performance baseline.

Next, the code is compiled to WebAssembly, yielding a file containing the WASM bytecode. This file is then run via the WebAssembly virtual machine. While WebAssembly has been designed for speed and efficiency, some overhead remains, which will necessarily reduce performance. This will serve to quantify the overhead resulting purely from running non-native code.

Lastly, we run the job through REC. Since the actual execution uses the same WebAssembly runtime as before, running the code should take approximately the same time. However, this time, the job will have to run through the entire REC pipeline, which includes multiple network transfers. This will be the slowest run since it has the largest overhead. Using this approach, we can quantify the performance tradeoff to benefit from the flexibility of REC.

All components of our evaluation and all data are released under open source/open data licenses[6].

A. *Job Runtimes*

Figure 3 shows the time needed to complete the example workload in each execution mode.

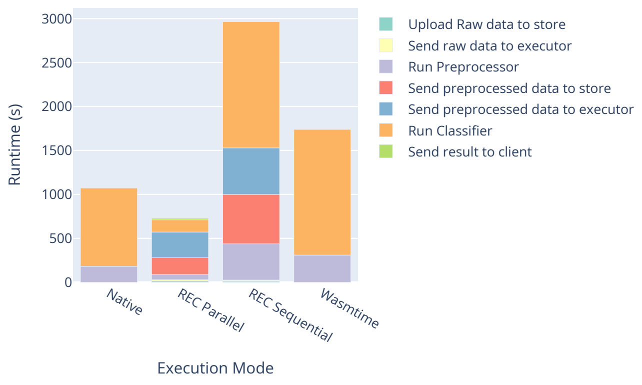

Fig. 3. Job runtimes

Native refers to the natively compiled C-executable, *Wasmtime* to the same program compiled to WebAssembly and executed in the wasmtime VM, while *REC Sequential/Parallel* is the WebAssembly program running through REC.

The native program took, on average, a little over 1,074 seconds to complete; 183 seconds (17%) were spent for the preprocessing step, and 891 seconds (83%) on running the actual classifier.

[6]Links to be added after the end of the review process.

For the *Wasmtime* runs, we see the overhead introduced by relying on WebAssembly, which has several layers of abstraction that necessarily result in lower performance. In particular, we see the *Wasmtime* runs taking 1,741 seconds, on average, which is a 62% slowdown. If we compare each step, then we see a 69% slowdown for the preprocessor step (310 seconds), and a 60% slowdown for the classifier (1,430 seconds).

REC Sequential is the wasmtime program, executed via REC. Since the actual execution is identical, the times for the preprocessor and classifier step are largely the same. We do, however, have the addition of network transfers, which take additional time. In particular, sending the preprocessed data in raw PCM form between nodes takes about as much time as running the preprocessor step.

Lastly, we see some performance gain in the *REC Parallel* run. In a distributed network with several executors, a parallelizable job may be executed on several nodes. To simulate this, we chunked the input file into 12 equally-sized pieces and created 12 jobs, which were started simultaneously. We can see the effect in the far shorter completion times for both the preprocessor and classifier jobs. The parallelized preprocessor completed in an average of 57 seconds (31% of the native run), while the classifier took 139 seconds (16% of the native run).

In total, running a job through REC incurs a performance penalty due to both inefficiencies with WebAssembly and the need to send programs and data over the network. These are, however, fundamental properties of any distributed execution framework and cannot be avoided. Through clever partitioning of work, we can however, achieve improvements due to parallel processing.

B. *System Load*

Figure 4 shows the system load (in terms of CPU utilization) generated by each job. Note that ”100%” is the equivalent of a single CPU core being used to capacity. Therefore, utilizations above 100% are plausible with more than one CPU core available.

Both the native execution and Wasmtime run in a single thread and generate 100% load to run the computation. While the compute server was not running any additional jobs during these experiments, a modern operating system still represents a very dynamic environment, so there may be dips and peaks in system load.

For the *REC Sequential* run, the actual computation is also limited to 100% load; however, since REC components continuously run in the background, we get more states with higher load. In addition, the computation does not start immediately since the job and data have to be first deployed to the executor, which is why the lower load states exist.

Lastly, during *REC Parallel*, 12 jobs are running in parallel; thus, the load reaches well over 1,000%. Since not all the jobs will start/finish at precisely the same time, we do not get a precise load of 1,200%, but rather a continuum since the overleap of computations generates a varying load regime.

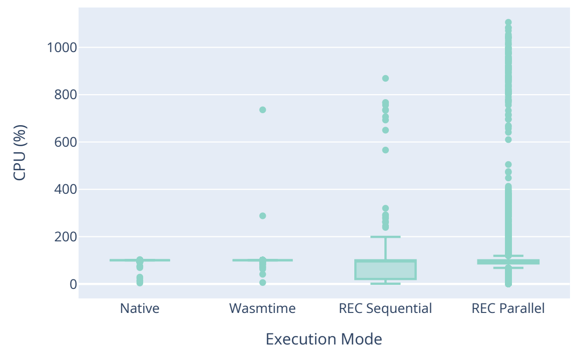

Fig. 4. System load

A single-threaded, non-I/O-bounded program will run at exactly 100% load most of the time, while parallel computation generates additional load in exchange for a shorter runtime. Since many of the REC processes, such as the broker or data store, are idle while the executor processes the job, their overhead appears negligible.

VI. Conclusion

We presented REC, a novel, resilient edge computing platform designed to address the critical need for robust computing capabilities in emergency scenarios where traditional centralized cloud infrastructure may be inaccessible or unreliable. REC successfully integrates the convenience of serverless computing with the principles of resilience and locality, offering a distributed solution that enables diverse parties to pool their computing resources into a local network, thereby supporting recovery efforts in crisis situations. REC leverages WebAssembly for platform-agnostic code execution, ensuring broad compatibility across various devices and systems. A key component of REC is its implementation of a distributed key-value store, which provides reliable data persistence in a decentralized environment. REC's modular design incorporates multiple node types, including brokers, executors, datastores, and clients, working in concert to efficiently manage job execution and data handling. Notable features of REC include a distributed key-value store, self-healing capabilities, job parallelization, and a flexible node discovery mechanism. The implementation of REC is available as free software on GitHub. Our experimental evaluation reveals that while REC introduces some performance overhead compared to native execution, it compensates by enabling parallel processing. This capability can yield significant speedups in computational tasks, particularly beneficial in resource-constrained emergency scenarios.

There are several areas for future work. One primary area for improvement would be a DTN-based publish-subscribe mechanism that allows clients to express interest in specific types of job results, rather than individual jobs, to improve resource utilization and information dissemination. Moreover, future work could focus on developing smarter decision-making processes to optimize the choice of suitable executors, considering factors such as current workload distribution, network conditions, specific job requirements, and resource availability. Finally, implementing advanced replication schemes for both regular data and metadata would significantly enhance REC's resilience. This approach would ensure data availability in the face of node failures or network partitions.

References

[1] A. Aske and X. Zhao, "Supporting multi-provider serverless computing on the edge," in *Workshop Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP Workshops '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229710.3229742

[2] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 1–10.

[3] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[4] A. Sterz, L. Baumgärtner, J. Höchst, P. Lampe, and B. Freisleben, "Opload: Offloading computational workflows in opportunistic networks," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, 2019, pp. 381–388.

[5] S. Ristov, D. Kimovski, and T. Fahringer, "Faascinating resilience for serverless function choreographies in federated clouds," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2440–2452, 2022.

[6] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[7] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 66–73, jul 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656887

[8] S. Burleigh, K. Fall, and E. J. Birrane, "Bundle Protocol Version 7," RFC 9171, Jan 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9171

[9] A. Penning, L. Baumgärtner, J. Höchst, A. Sterz, M. Mezini, and B. Freisleben, "Dtn7: An open-source disruption-tolerant networking implementation of bundle protocol 7," in *International Conference on Ad-Hoc Networks and Wireless (AdHoc-Now 2019)*. Luxembourg, Luxembourg: Springer, 2019, pp. 196–209.

[10] L. Baumgärtner, J. Höchst, and T. Meuser, "B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7," in *2019 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, Dec 2019, pp. 1–8.

[11] H. Bellafkir, M. Vogelbacher, and B. Freisleben, "Urban sound classification on resource-constrained edge devices," in *Proceedings of the Service-Oriented Computing – ICSOC 2024 Workshop on Lightweight AI-based Services (LAIS 2024)*, 2024.