Fault-Tolerant Edge Computing in JAMScript

Rossen Vladimirov

School of Computer Science
McGill University
Montreal, Canada

August 2021

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Muthucumaru Maheswaran, for his invaluable guidance, advice, and insights. His deep knowledge and passion for the field of the Internet of Things were instrumental to the success of my research. Without his continuous support, patience, and encouragement, this thesis would not have been possible. I would also like to thank the members of the Advanced Networking Research Lab at McGill University for their contributions to JAMScript.

Abstract

The Internet of Things (IoT) experiences rapid growth in various industries such as healthcare, automotive, manufacturing, and retail. IoT devices generate large volumes of data but lack processing capabilities. In the Cloud of Things (CoT) computing model, the devices are connected to cloud computing systems in data centers which provide these capabilities. Edge computing brings high-powered computing resources closer to the devices to reduce the latency of access. Specialized software frameworks address the challenges of edge environments such as device mobility, disconnections, and network latency variability. JAMScript is a polyglot programming language and middleware for developing edge-oriented IoT applications. It combines C and JavaScript allowing a single program to execute on device, edge, and cloud nodes. JAMScript implements efficient inter-node communication, automatic data propagation, and the ability to run distributed computations. In this thesis, we introduce application zones which group together computing nodes in physical proximity executing the same JAMScript program. Application zones enable performing synchronized distributed computations, generating shared data at the edge, and recovering from edge server failures. We present supercalls which enhance synchronous controller-to-worker computations in JAMScript. A supercall is a collective distributed computation in a zone which synchronizes computation execution and data generation across the participating devices. Supercalls have a built-in fault tolerance mechanism with automatic re-execution. We enrich the data management capabilities of JAMScript with private and shared data generated at the edge. Private data is based on data from a single device, while shared data is based on data from multiple devices participating in a supercall. We incorporate application zones, supercalls, and private and shared data in a robust fault tolerance mechanism for recovering from edge server failures. We demonstrate the new features in an application for distributed multiple-target tracking which efficiently and accurately tracks objects moving in a physical space. We collect performance results from a simulation with real flight data in scenarios with and without edge server failures. The experiments show that the overhead of the fault tolerance scheme is small and the runtime recovers quickly from edge server failures.

Abrégé

L'Internet des Objets (IdO) connaît une croissance rapide dans divers secteurs tels que la santé, l'automobile, la fabrication et le commerce de détail. Les appareils IdO produisent de grandes quantités de données mais manquent de capacités de traitement. Dans le modèle de l'informatique en nuage, les appareils sont connectés à des systèmes informatiques dans des centres de données qui fournissent ces capacités. L'informatique en périphérie rapproche les ressources informatiques des appareils IdO pour réduire les temps d'accès. Des logiciels spécialisés répondent aux défis de l'informatique en périphérie tels que la mobilité des appareils, les déconnexions et la variabilité de la latence du réseau. JAMScript est un langage de programmation polyglotte et un intergiciel pour le développement d'applications IdO. Il combine C et JavaScript permettant à un seul programme de s'exécuter sur les appareils IdO, les nœuds de périphérie et de nuage. JAMScript met en œuvre une communication inter-nœuds efficace, la propagation automatique des données et la possibilité de réaliser des calculs distribués. Dans cette thèse, nous introduisons les zones d'application qui regroupent des nœuds de calcul à proximité physique exécutant le même programme JAMScript. Les zones d'application permettent d'effectuer des calculs distribués et synchronisés, de générer des données partagées et de récupérer des pannes de serveur de périphérie. Nous présentons les supercalls qui améliorent les calculs synchrones contrôleur-travailleur dans JAMScript. Un supercall est un calcul distribué dans une zone qui synchronise l'exécution du calcul et la génération de données entre les appareils participants. Les supercalls ont un mécanisme intégré de tolérance aux pannes avec ré-exécution automatique. Nous enrichissons les capacités de gestion des données de JAMScript avec des données privées et partagées qui sont générées en périphérie du réseau. Les données privées se basent sur les données d'un seul appareil, tandis que les données partagées se basent sur les données de plusieurs appareils participant à un supercall. Nous incorporons les zones d'application, les supercalls et les données privées et partagées dans un mécanisme robuste de tolérance aux pannes de serveur de périphérie. Nous montrons les nouvelles fonctionnalités dans une application distribuée pour le suivi de cibles mobiles qui suit efficacement et précisément les objets se déplaçant dans un espace physique. Nous collectons des résultats de performance à partir d'une simulation avec des données de vol réelles dans des scénarios avec et sans pannes de serveur de périphérie. Les expériences montrent que la surcharge associée au mécanisme de tolérance aux pannes est faible et que l'environnement d'exécution récupère rapidement après une panne de serveur de périphérie.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Edge Computing

The Internet of Things (IoT) [1] consists of smart devices which exchange data with computing systems over the Internet. IoT experiences rapid growth in various industries such as healthcare [2, 3], automotive [4], manufacturing [5], and retail [6]. The IoT ecosystem is heterogeneous and includes a wide range of devices and applications [7]. For example, smart cities leverage IoT technologies for monitoring air and water quality [8, 9] and use the insights to provide more efficient management of resources. Homes are equipped with smart thermostats [10] and security systems [11, 12] to ensure the comfort and safety of the occupants. Networks formed of moving vehicles and roadside units open many possibilities for autonomous vehicles [13].

The Cloud of Things (CoT) computing model [14] brings together cloud computing [15] and IoT. Today many devices with constrained computing resources are connected to the Internet. Typically IoT devices generate large volumes of data at high rates [16] but lack reliable long-term storage and sufficient processing capabilities. The devices are connected to cloud computing systems which provide these capabilities. In cloud computing, the computing resources for data processing are located in data centers [17]. The resource capacities adjust automatically according to the demands of the devices [18].

The high latency of the connection between a device and computing resource on the cloud poses a major challenge [19] and may make cloud computing unsuitable for latency-sensitive applications such as real-time gaming, augmented reality, and real-time streaming [20]. To overcome this, edge computing [21] introduces the edge as an intermediary between the devices and the cloud. The role of the edge is to bring high-powered computing resources closer to the devices to reduce the latency of access.

Reliability is an important concern for distributed applications [22]. IoT applications run on cloud, edge, and device nodes. Device and edge nodes may fail and rejoin the system after restarting. A node may temporarily lose network connectivity. The latency of a network link may vary. Node and network failures may cause the loss of computational state at one or more nodes and lead to the failure of a collective distributed computation in progress. Due to the unique characteristics of edge environments, innovative reliability and fault tolerance solutions designed specifically for edge computing are being developed [23, 24, 25].

1.2 The JAMScript Programming Language

JAMScript [26] is a polyglot programming language and middleware for developing edge-oriented IoT applications. A JAMScript program runs on a hierarchy of device, edge, and cloud nodes. A component of the JAMScript program runs on each node. In a typical scenario with all three types of nodes present, the hierarchical structure of JAMScript program components can be viewed as a tree rooted at a cloud node with leaves corresponding to device nodes.

JAMScript combines the C [27] and JavaScript [28] programming languages allowing developers to write a single program executing on different types of nodes. The C language is used for programming devices, while the JavaScript language is used for programming device, edge, and cloud nodes. Combining C and JavaScript achieves two of the main goals of JAMScript. The first goal is high performance and efficient use of computing resources. C has a proven track record in embedded systems programming [29, 30] because it is fast and memory-efficient [31]. The introduction of Node.js [32] has made JavaScript a top contender for writing high-performance server applications [33, 34]. The second goal of combining C and JavaScript is creating a language that is easy to learn and use. Most developers working in the IoT area are familiar with one or both of C and JavaScript [35] which facilitates the adoption of JAMScript.

JAMScript implements efficient inter-node communication by introducing activities which are a form of remote method invocation (RMI) [36]. Activities are invoked up or down the tree. An activity is initiated at one node and executes at one or more nodes at different levels of the hierarchy. Different types of activities are used for synchronous and asynchronous invocations of remote functions. The JAMScript runtime automatically handles implementation details such as resolving network addresses and exchanging messages between nodes.

JAMScript introduces loggers and broadcasters for exchanging data between nodes without making function calls. Data can be sent upward or downward to one or more nodes in the node hierarchy. Loggers are used for propagating data to parent nodes, while broadcasters are used for propagating data to child nodes. The data propagation is performed automatically by the JAMScript runtime.

JAMScript is designed for high performance and scalability. The inter-node data exchange implementation allows large amounts of data to be transferred concurrently between different nodes. The JAMScript runtime efficiently manages the execution of a JAMScript program on a large number of nodes. An important feature of JAMScript is its ability to orchestrate and synchronize the execution of a distributed computation on many devices. The synchronization ensures that the data collected from the devices is time-aligned. The synchronization also ensures that the devices perform an action on their physical environment at the same time.

JAMScript is specifically designed for addressing many of the challenges of edge environments such as network latency variability and disconnections. The network latency may change due to device mobility when the distance between device and edge changes. The JAMScript runtime addresses this by resending messages and waiting for acknowledgements. By design, a JAMScript program is deployed in such a way that a disconnected node or subtree can continue operating autonomously. This is achieved by ensuring that a component of the JAMScript program which acts as a controller runs on every node. The controller continues to manage the computations when the node is disconnected.

1.3 Thesis Contributions

The main contributions of the thesis are the following:

- We introduce application zones which group together computing nodes in physical proximity executing the same JAMScript program. The partitioning of the physical space into zones enables performing synchronized distributed computations, generating shared data at the edge, and dealing with edge server failures. We describe the roles of the primary and backup edge servers in a zone in recovering from edge server failures.

- We present the design and implementation of supercalls which enhance synchronous controller-to-worker computations in JAMScript. A supercall is a collective distributed computation in a zone which runs on the primary edge server and devices. It synchronizes computation execution and data generation across the participating devices. Supercalls have a built-in fault tolerance mechanism for recovering from edge server failures with automatic re-execution.

- We enrich the data management capabilities of JAMScript with private and shared data generated at the edge from device data. Private data is based on data from a single device, while shared data is based on data from multiple devices participating in a computing context during a supercall. We explain the differences between private and shared data and the reliability guarantees for each one.

- We incorporate application zones, supercalls, and private and shared data in the design and implementation of a robust fault tolerance mechanism in JAMScript for recovering from edge server failures. We discuss node and network failures and techniques for system recovery in different failure scenarios.

- We showcase application zones, supercall programming, shared data, and the fault tolerance implementation in an application for distributed multiple-target tracking which efficiently and accurately tracks objects moving in a physical space.

- We run a simulation with real flight data and measure the performance of the supercall implementation in various scenarios with and without edge server failures. The experimental results show that the JAMScript runtime recovers quickly from edge server failures and the overhead of the implemented fault tolerance scheme is very small.

1.4 Thesis Organization

Chapter 2 provides background information on the controller-worker model and JAMScript language and middleware. Chapter 3 discusses motivating scenarios from edge computing applications, defines the system architecture requirements, and introduces application zones. Chapter 4 covers the design and implementation of JAMScript supercalls and programming with supercalls. Chapter 5 discusses node and network failures and techniques for system recovery in different failure scenarios. Chapter 6 focuses on private and shared data and explains the data reliability guarantees. Chapter 7 describes the distributed multiple-target tracking application and analyzes the experimental results from a simulation with flight data. Chapter 8 presents related research in edge computing and performs a comparative analysis. Chapter 9 concludes the thesis with a discussion of future research for extending the presented work.

Chapter 2

Background Information

2.1 Controller-Worker Model for Edge Computing

A JAMScript program executes on two types of computing nodes: controller and worker. Both controllers and workers can perform computations. Controllers also manage workers and subordinate controllers. Each controller and worker runs as a separate process. Controllers and workers can run on the same or different machines connected by a network. JAMScript supports different deployment configurations on a single or multiple machines.

Workers run on a wide range of devices like autonomous vehicles, smart appliances, and stations for monitoring air quality. Typically, devices generate data at high rates but lack reliable storage and processing capabilities. These capabilities are provided by controllers in data centers, roadside units of vehicular networks, and base stations of 5G networks.

In a device deployment, one or more workers are connected to a device controller. Typically, the device controller and workers run on the same machine (device). A device deployment can also be a multi-node configuration with a local area network (e.g., Wi-Fi) connecting the workers to the device controller. In this configuration, if the connection to the single controller fails, the worker cannot continue operating.

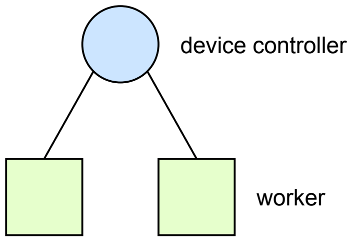


Figure 2.1: Device deployment.

In an edge deployment, one or more device controllers are connected to an edge controller using a wide area network (e.g., a cellular network). In this configuration, computationally intensive tasks can be performed at the edge server instead of the devices which often lack processing capabilities. Each worker is connected to both the edge and device controllers. When the network connecting a worker to the edge controller is unavailable, the worker is still connected to its device controller and can continue operating.

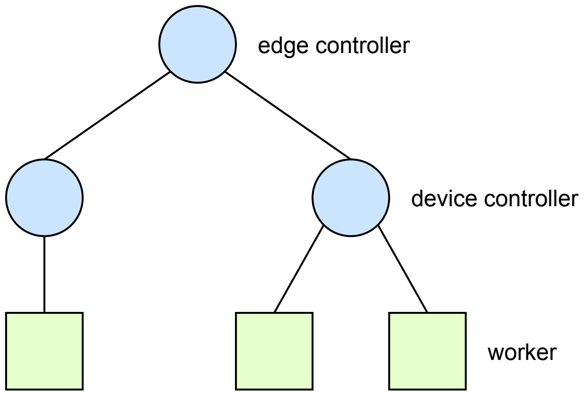


Figure 2.2: Edge deployment.

In a cloud deployment, one or more edge controllers are connected to a cloud controller. One of the main use cases for configurations with multiple edge controllers is dealing with device mobility. Devices attach to the nearest edge controller as they move. In a cloud deployment, each worker is connected to three controllers: device, edge, and cloud. This ensures that the worker maintains functionality in case of network disruptions.

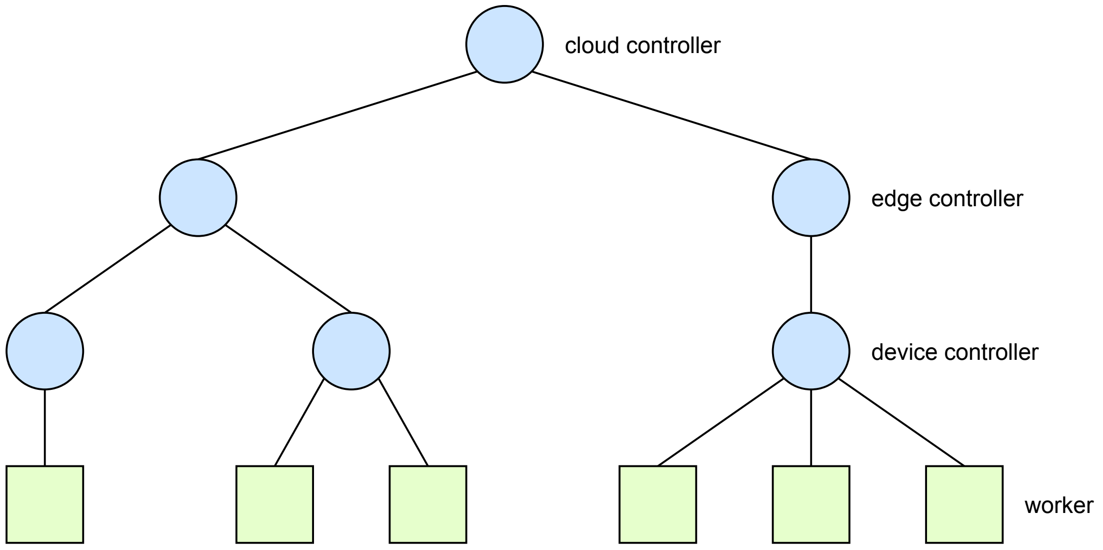


Figure 2.3: Cloud deployment.

The controller-worker model implementation in JAMScript supports function calls and data flows between controllers and workers as well as controllers and parent or subordinate controllers. No interactions are allowed between workers or controllers at the same level. An edge controller cannot initiate the execution of a function on another edge controller. A worker cannot run a function on another worker.

Each node can make two types of function calls: self and remote. During a self call, a node invokes a function which executes at the same node. During a remote call, a node invokes a function which executes at another node. In JAMScript, remote calls can be either upward or downward. An upward function call is performed from a worker to a controller or from a controller to a parent controller. A downward call is made from a controller to a worker or from a controller to a subordinate controller. Function calls can be synchronous (blocking) or asynchronous (non-blocking). Synchronous calls block the execution of the program at the call site until the function completes. Asynchronous calls do not block the program's execution.

In a typical scenario, workers are data producers. For example, a worker can be equipped with a temperature sensor and can capture temperature readings periodically. Workers can also be consumers of data. For instance, a worker in an autonomous vehicle can receive a command to turn or stop. Controllers consume and transform data from workers. Controllers can also send data to workers and subordinate controllers.

JAMScript provides facilities for upward and downward data exchanges without the need to make function calls. The runtime automatically propagates data written to a logger (upward stream) from a worker or controller to the controllers of all of its ancestor nodes. Similarly, the runtime automatically propagates data written to a broadcaster (downward stream) from a controller to its subordinate controllers and workers.

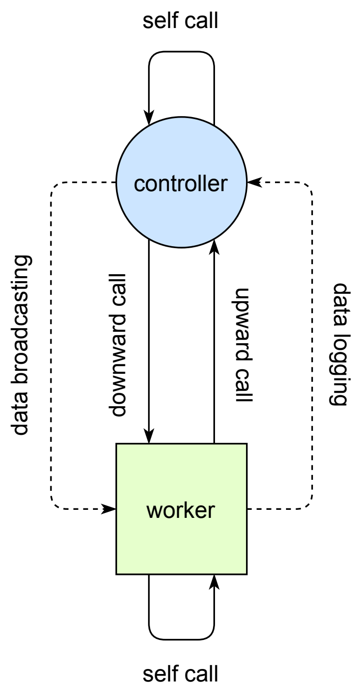


Figure 2.4: Function calls and data flows.

2.2 JAMScript Language Overview

JAMScript is a polyglot programming language which combines C and JavaScript. The C language is used for programming worker nodes, while the JavaScript language is used for programming controller nodes. Worker nodes typically run on embedded devices for which C is a natural choice due to its compact size and high execution speed. JavaScript running in a Node.js environment on controller nodes is suitable for developing high-performance server applications that can handle many connections and large volumes of data.

The worker and controller code are separated into two source files: C and JavaScript. The C file contains the worker code and the JavaScript file contains the controller code. In the JavaScript file, the developer can specify the controller level (device, edge, or cloud) at which different parts of code run. The `jsys` global variable stores read-only properties of the controller node such as its type (level). There can be different branches of execution based on the value of the `jsys.type` property as shown in Listing 2.1.

Listing 2.1: Conditional execution based on the controller level.

```
switch (jsys.type) {
    case 'cloud':
        // cloud controller code
        f1();
        break;
    case 'fog':
        // edge controller code
        f2();
        break;
    case 'device':
        // device controller code
        f3();
        break;
}
```

JAMScript implements inter-node communication by introducing activities which are a form of remote method invocation. An activity is initiated at one node and executes at one or more nodes at different levels. Activities allow a worker to execute a function on one or more controllers, a controller to execute a function on one or more workers, or a controller at one level to execute a function on one or more controllers at other levels. A worker cannot initiate the execution of a function on another worker. A controller cannot initiate the execution of a function on another controller at the same level.

The developer must select the appropriate type for each activity by considering where it should be initiated (e.g., edge controller) and where it should execute (e.g., worker). The developer does not need to know the addresses of the nodes or to manage connections. This is performed automatically by the JAMScript runtime.

2.2.1 Worker Activities

Worker activities are initiated from a controller and execute on all workers in the subtree rooted at that controller. When initiated from the cloud controller, a worker activity runs on all worker nodes. When initiated from an edge controller, a worker activity runs only on the workers connected to that edge controller. When initiated from a device controller, a worker activity runs only on the workers connected to that device controller.

Worker activities are also called J2C activities and are defined in the C source file. There are two kinds of worker activities: synchronous and asynchronous. Similarly to regular functions, synchronous and asynchronous activities differ in whether they can return a value and whether they block the execution of the controller program until the activity completes.

Synchronous

A synchronous worker activity blocks the execution of the program at the call site until the activity completes. The result of the activity is an array of values returned from the participating workers. The worker activity is defined with a regular C function preceded by the `jsync` keyword. It takes an arbitrary number of parameters and must declare a return type. The body of the activity contains regular C code.

Listing 2.2: Synchronous worker activity definition (C file).

```c
jsync int my_j2c_sync_activity(int d, char* s) {
    printf("d=%d, s=%s\n", d, s);
    return d * 2;
}
```

Listing 2.3: Synchronous worker activity invocation at the controller (JavaScript file).

```javascript
if (jsys.type === "fog") { // edge level
    var v = my_j2c_sync_activity(123, "abc");
    console.log(v.device); // array of worker values
}
```

Asynchronous

An asynchronous worker activity does not block the execution of the program at the call site. The participating workers do not return a value to the controller. The worker activity is defined with a regular C function preceded by the `jasync` keyword. It takes an arbitrary number of parameters and must not declare a return type. The body of the activity contains regular C code.

Listing 2.4: Asynchronous worker activity definition (C file).

```c
jasync my_j2c_async_activity(int d, char* s) {
    printf("d=%d, s=%s\n", d, s);
}
```

Listing 2.5: Asynchronous worker activity invocation at the controller (JavaScript file).

```
if (jsys.type === "fog") { // edge level
    my_j2c_async_activity(123, "abc");
}
```

2.2.2 Controller Activities

In JAMScript, controller activities are defined in the JavaScript source file. There are two kinds of controller activities: synchronous and asynchronous. Similarly to regular functions, synchronous and asynchronous activities differ in whether they can return a value and whether they block the execution of the program until the activity completes.

Controller activities are initiated from a worker or controller and execute on one or more controllers. Controller activities initiated from workers are called C2J activities. Controller activities initiated from controllers are called J2J activities.

When initiated from a worker, a controller activity runs on a subset of all controllers to which that worker is connected. Depending on the deployment, the worker may be connected to up to three controllers: device, edge, and cloud. The set of controllers on which the activity runs depends on whether the activity is synchronous or asynchronous. When initiated from a controller, a controller activity runs on that controller as well as on all adjacent controllers.

Table 2.1: Initiation and execution of controller activities.

Type	Initiated From	Executes On		
		Device Controllers	Edge Controllers	Cloud Controller
sync	worker	highest available ancestor		
async	worker	co-located	parent	yes
sync/async	device controller	self	parent	no
sync/async	edge controller	children	self	yes
sync/async	cloud controller	none	all	yes

Synchronous

A synchronous controller activity blocks the execution of the program at the call site until the activity completes. When initiated from a worker, the result of the activity is a single value returned from the controller on which the activity runs. When initiated from a controller, the result of the activity is a collection of values returned from the controllers on which the activity runs as described in Table 2.1.

The controller activity is defined with a regular JavaScript function preceded by the `jsync` keyword. It takes an arbitrary number of parameters. The body of the activity contains regular JavaScript code.

Listing 2.6: Synchronous controller activity definition (JavaScript file).

```
jsync function my_c2j_sync_activity(d, s) {
    console.log("d=" + d + ", s=" + s);
    return d * 2;
}
```

Listing 2.7: Synchronous controller activity invocation at the worker (C file).

```
int my_c2j_sync_activity(int d, char* s); // C prototype

void execute() {
    int v = my_c2j_sync_activity(123, "abc");
    printf("v=%d\n", v);
}
```

Asynchronous

An asynchronous controller activity does not block the execution of the program at the call site. The controllers on which the activity runs do not return a value to the caller. The controller activity is defined with a regular JavaScript function preceded by the `jasync` keyword. It takes an arbitrary number of parameters. The body of the activity contains regular JavaScript code.

Listing 2.8: Asynchronous controller activity definition (JavaScript file).

```
jasync function my_c2j_async_activity(d, s) {
    console.log("d=" + d + ", s=" + s);
}
```

Listing 2.9: Asynchronous controller activity invocation at the worker (C file).

```
void my_c2j_async_activity(int d, char* s); // C prototype

void execute() {
    my_c2j_async_activity(123, "abc");
}
```

2.2.3 Data Flows

In addition to activities, JAMScript introduces loggers and broadcasters for inter-node data exchange without making function calls. Loggers are used for propagating data to parent nodes, while broadcasters are used for propagating data to child nodes. A data store (Redis [37]) is co-located with each controller.

When a value is written to a logger, it is stored in the data store co-located with that node and the data stores of its ancestors along the path to the root. When a value is written to a broadcaster, it is stored in the data store co-located with that node and the data stores of all nodes in the subtree rooted at that node. The data propagation is performed automatically by the JAMScript runtime. The different scenarios are described in Table 2.2.

Table 2.2: Data flow scenarios.

Written From		Stored At		
		Device Data Stores	Edge Data Stores	Cloud Data Store
logger	device	self	parent	yes
	edge	none	self	yes
	cloud	none	none	yes
broadcaster	device	self	none	no
	edge	children	self	no
	cloud	all	all	yes

Loggers

Loggers are used for sending data from a node to its ancestors. In a typical scenario, a device with a sensor periodically writes a measurement to a logger which propagates it to the edge and cloud for processing. A logger is defined in the JavaScript file using the `jdata` construct and the `logger` keyword. Both controllers and workers can write to a logger. Nodes at different levels can write to the same logger. Only controllers can read from a logger.

Data saved to a logger from each controller or worker is stored in a separate data stream. A data stream contains a sequence of values with timestamps. JAMScript supports different types for the values stored in a data stream (e.g., `int`, `float`, `char*`). Values can only be appended to a data stream and can never be removed. The data stream API provides methods for determining the number of values in a data stream, retrieving all values with or without timestamps, retrieving the last N (constant) values and all values logged within a specified time period.

A controller reads values from a logger by iterating over the data streams of all child controllers and workers as shown in Listing 2.10. A controller writes a value to a logger by calling the `log` method of the stream returned from the logger's `getMyDataStream` method as shown in Listing 2.11. A worker writes a value to a logger by assigning a value to the logger variable as shown in Listing 2.12.

Listing 2.10: Reading from a logger.

```
jdata {
    int x as logger;
}

function read_from_logger() {
    var num_streams = x.size();
    for (var i = 0; i < num_streams; i++) {
        var stream = x[i];
        var value = stream.lastValue();
        console.log(value);
    }
}
```

Listing 2.11: Writing to a logger from a controller.

```
function write_to_logger_from_controller() {
    var my_stream = x.getMyDataStream();
    my_stream.log(123);
}
```

Listing 2.12: Writing to a logger from a worker.

```
void write_to_logger_from_worker() {
    x = 123;
}
```

Broadcasters

Broadcasters are used for sending data from a node to all nodes in the subtree rooted at that node. A broadcaster is defined in the JavaScript file using the `jdata` construct and the `broadcaster` keyword. Only controllers can write to a broadcaster. Nodes at different levels can write to the same broadcaster. Both controllers and workers can read from a broadcaster.

A broadcaster contains a single value at a time. JAMScript supports different types for the value stored in a broadcaster (e.g., `int`, `float`, `char*`). The broadcaster API provides the `getLastValue` method for retrieving the most recent broadcaster value.

A controller writes a value to a broadcaster by calling the `broadcast` method as shown in Listing 2.13. A controller reads a value from a broadcaster by calling the `getLastValue` method as shown in Listing 2.14. A worker reads a value from a broadcaster by assigning the broadcaster variable to another variable as shown in Listing 2.15.

Listing 2.13: Writing to a broadcaster.

```
jdata {
    int x as broadcaster;
}

function write_to_broadcaster() {
    x.broadcast(123);
}
```

Listing 2.14: Reading from a broadcaster from a controller.

```
function read_from_broadcaster_from_controller() {
    var value = x.getLastValue();
    console.log(value);
}
```

Listing 2.15: Reading from a broadcaster from a worker.

```
void read_from_broadcaster_from_worker() {
    int temp;
    temp = x;
    printf("x=%d\n", temp);
}
```

2.2.4 Conditional Execution

In JAMScript, conditionals are used for limiting the set of nodes on which an activity executes. In a typical scenario, the set of nodes is filtered based on node type (device, edge, or cloud). For example, an asynchronous C2J activity is called from a worker and executes on its co-located device controller, parent edge controller, and cloud controller. A conditional can be used to limit its execution to the parent edge controller or cloud controller or both.

A conditional is defined in the JavaScript source file using the `jcond` construct as shown in Listing 2.16. A conditional is applied to an activity by adding its name in curly brackets after the `jsync` or `jasync` keyword. Conditionals can be used with both controller and worker activities. Conditionals are evaluated at runtime on all nodes which are candidates for executing the activity.

Listing 2.16: Defining and using a conditional.

```
jcond {
    deviceOnly: jsys.type == "device";
    edgeOnly: jsys.type == "fog";
    cloudOnly: jsys.type == "cloud";
}

jasync {edgeOnly} function my_c2j_async_activity(d, s) {
    console.log("d=" + d + ", s=" + s);
}
```

Chapter 3

System Architecture

3.1 Motivating Scenarios and Design Requirements

We present motivating scenarios from edge computing which illustrate the need for device-specific and shared data, synchronized distributed computations, application zones, and a robust fault tolerance mechanism. We discuss two edge computing applications: a drone fleet controller and an augmented reality game.

The first application controls a fleet of drones from an edge server on the ground. Each drone captures a video feed which is sent back to the edge server, augmented with annotations, and streamed to viewers. The edge server periodically collects information from each drone about its location, altitude, and speed. It also receives data from devices with sensors measuring wind speed and direction. The edge server uses the data collected from the drones and wind sensors to make decisions for issuing navigation commands to the drone formation (e.g., turn, speed up, land).

The second application is an augmented reality game hosted at multiple edge servers. Players interact with both the physical and virtual world as well as with each other as either teammates or opponents. Players control virtual avatars and can improve their armor, acquire equipment, go on expeditions, and take part in battles.

3.1.1 Device-Specific and Shared Data

Typically, edge applications transfer data from devices to edge servers for processing and may then send the results back to the devices. The data at an edge server can be either specific to a device or associated with a common computing context shared between many devices. Collecting device-specific data at the edge server is a classic use case of edge computing [38].

An example of device-specific data is video captured from a drone and sent to an edge server for analysis and processing (e.g., adding annotations) before being streamed to viewers. Temperature readings from a thermostat in a smart home are another example of device-specific data. The edge server can also transform the data collected from each device. For example, it may calculate an average of temperature readings from the thermostat over a period of time. Based on the device data, the edge server can issue a command to the thermostat to adjust the temperature in the home.

An example of shared data associated with a computing context common to many devices involves the augmented reality game application. The edge server maintains a shared state consisting of information about the locations and actions of the players as well as characteristics of the physical and virtual world surrounding them. The shared state is created using data provided by many devices and is used to process the actions of the players interacting with the environment.

3.1.2 Synchronized Distributed Computations

Synchronization of distributed computations across many devices is a major challenge for edge computing applications [39]. In the drone fleet controller application, the drones are connected to the same edge server. The edge server processes the data collected from the drones and wind sensors to build and maintain a shared state used for making navigation decisions. For the shared state to be accurate, it needs to be built from device data which is both recent and time-aligned. This means that the computations which run on each drone and send back to the edge server information about its location, altitude, and speed must be synchronized. The execution of a command sent from the edge server to the devices must be synchronized to ensure that all drones in the fleet interact with the physical world simultaneously (e.g., turn, speed up). The system must also be able to execute in parallel computations at the edge server and devices. The edge server computation builds a map with the locations of all drones in the fleet.

3.1.3 Application Zones

Device mobility is an important consideration for many edge applications [40, 41]. In the augmented reality game application, the devices move when the players move. A device can disconnect from one edge server and connect to another one with lower network latency. Different sets of devices can be connected to different edge servers.

Shared data described in Section 3.1.1 and synchronized distributed computations described in Section 3.1.2 operate with a common computing context. Such a context is associated with a group of edge servers and devices which are typically near each other. Grouping edge servers and devices can be achieved by partitioning the physical space into zones. The partitioning can be different for each application. An application zone groups edge servers and devices in physical proximity running the same application.

Zone constituents change dynamically. Devices can fail or move between application zones. New devices can join a zone. For example, in an online game players join and leave the battlefield as the game progresses. Edge servers can also fail and restart.

In the augmented reality game application, when the players operating the devices interact with each other and the environment, a shared state must be updated periodically at the edge server. The easiest way to maintain a shared state is by having all devices in the zone connected to the same edge server. One server in the zone can be designated as main or primary. When the primary server fails, a backup server takes over. Application zones with designated primary and backup edge servers play a vital role in the design of a fault tolerance mechanism which satisfies the requirements outlined in Section 3.1.4.

When devices move between application zones, device-specific and shared data are affected in different ways. In the augmented reality game application, when a player moves between zones, their gear must be unaffected. This means that the private data associated with a device must be transferred seamlessly between zones. Since shared data is associated with a zone, there is no need to transfer shared data between zones.

3.1.4 Fault Tolerance Requirements

Fault tolerance is a major concern for distributed applications especially in cloud and edge environments [42, 43, 44]. An edge application must be resilient against device and edge server failures, network disconnections, and network latency variability. The main goals of fault-tolerant implementations are fast recovery from failures with minimal service interruptions and low processing overhead.

Impact of Recovery on the System

A well-designed fault tolerance scheme must recover quickly in case of failures with minimal disruptions to the system. For example, if an edge server in the augmented reality game application fails, another one must take over as soon as possible. Players connected to the failed edge server must not experience any downtime and ideally should not be aware of the failure. All players should be able to continue playing almost immediately with the location and features of their characters unaffected by the failure. In the drone fleet controller application, if an edge server failure occurs when the drone formation is in flight, another edge server in the zone must take control. The flight must not be interrupted and the drones must not be left unattended.

Automatic Failure Recovery

The failure recovery process must be triggered automatically by the runtime environment when a failure is detected. In the augmented reality game application, the recovery must be transparent to the players who should not need to perform any specific actions to recover from an edge server failure. If a distributed computation is in progress when the failure occurs, the runtime must automatically re-execute the computation as part of the recovery process. In the augmented reality game application, if an edge server initiated action to teleport all connected players to another virtual location is in progress when a failure occurs, it must be re-executed. In the drone fleet controller application, if collection of drone flight data is in progress when the edge server fails, a backup edge server must re-execute the computation to ensure that the shared state reflects the most up-to-date information about the drones' locations, altitudes, and speeds.

Fault Tolerance Processing Overhead

The overhead of the fault tolerance scheme must be very small during normal operation. Low latency is crucial for both of the edge computing applications we have discussed. In the augmented reality game scenario, players interact with the physical and virtual world and the system must be very responsive for a truly immersive experience. In the drone fleet controller application, the data about a drone's location, altitude, and speed must reach the edge server with minimal delay when requested. This ensures that the edge server has an accurate view of the drone formation's flight parameters at any time and can make the right decisions based on up-to-date information. A command sent from the edge server to the drones must also be executed quickly before the drones' flight parameters have changed significantly.

Application-Specific Control of the Fault Tolerance Scope

A fault tolerance scheme mitigates the effects of failures on an application but incurs processing costs related to data replication and re-execution of failed computations. Reducing these costs is essential. It can be achieved by providing reliability guarantees only for the computations which need them. For example, a command to the drone fleet to land must be re-executed if there is a failure. On the other hand, a computation which periodically calculates a moving average of temperature readings might not need to be re-executed after a failure. The framework must provide the tools for application-specific control of the fault tolerance scope by the application developer.

Data Reliability Guarantees

The fault tolerance mechanism must ensure minimal or no data loss in case of failures with clearly defined reliability guarantees for different types of data. To ensure continuity, the shared state from the failed edge server must be available at a backup edge server. For example, restoring the flight history for the drone formation is crucial if a failure occurs while the fleet is in the air. To achieve this, the system may employ different data replication schemes involving multiple edge servers in the zone. Data replication requires additional processing. While some data must be preserved and restored after a failure, it may be acceptable to lose non-essential data in order to reduce replication costs. The application generates different types of data when different reliability guarantees are needed.

3.1.5 JAMScript Concepts and Features

Table 3.1 lists the new JAMScript concepts and features that we have implemented to address the design requirements. The new features build on and extend JAMScript's controller-worker architecture outlined in Section 2.1.

Table 3.1: Mapping of design requirements to JAMScript concepts and features.

Design Requirement	JAMScript Concept/Feature
data at the edge specific to one device	private data
data associated with a common computing context shared between many devices	shared data generated during a supercall from data from participating devices
synchronized distributed computations	supercalls with a fault tolerance mechanism for recovering from edge server failures
partitioning of the physical space into zones	application zones with many edge servers and devices in each zone
quick recovery from edge server failures with minimal disruptions to the system	- primary and backup edge servers in a zone - the backup server takes over the execution of computations when the primary one fails
automatic failure recovery	automatic re-execution of a supercall in progress when a failure occurs
low processing overhead of the fault tolerance scheme during normal operation	less than 1.9% processing overhead of the fault tolerance implementation
application-specific control of the fault tolerance scope	supercalls are used for implementing computations with reliability requirements
minimal or no data loss in case of edge server failures	- shared data is always available (either recovered or regenerated) - private data might be recovered partially
data reliability guarantees	- shared data from completed supercalls is available at the backup edge server - shared and private data from a supercall in progress are regenerated during the re-execution of the supercall after a failure - private data generated after the last completed supercall might be lost and is not regenerated

Application zones are defined in Section 3.2. The primary and backup edge servers introduced in Section 3.3 are an integral part of the fault tolerance mechanism described in Chapter 5. The supercall implementation detailed in Chapter 4 adds new functionality to synchronous controller-to-worker activities in JAMScript and offers automatic re-execution in case of a primary edge server failure. The low processing overhead of the fault tolerance scheme is confirmed by the experimental results in Section 7.4. Private and shared data, their relationship with supercalls, and their reliability guarantees are explained in Chapter 6.

3.2 Application Zones

3.2.1 Zone Overview

A JAMScript application is typically deployed in a physical space which is partitioned into areas (e.g., road segments). Areas are defined per application and do not change while the application is running. Different applications may partition the space differently. There are one or more edge servers in each area. Some edge servers can move between areas (e.g., an edge server on a bus). The size of an area is such that the edge servers in the area have the capacity to handle the expected number of devices. Since each device occupies some physical space, the number of devices in an area is not infinite and can be estimated.

A zone is a grouping of nodes in physical proximity executing the same JAMScript program/application. Nodes belonging to the same zone must execute the same application but need not run on the same physical machine. Nodes which run on the same machine but execute different JAMScript programs belong to different zones.

Each zone is associated with an area. Typically, nodes in a zone are physically located inside or close to that area. For an edge node, the zone to which it belongs is configured on start-up. For a device node, the zone to which it belongs is determined by its physical coordinates and current supercall. (Supercalls are described in Chapter 4.) If a device node is not participating in a supercall, it belongs to the zone whose area contains the device's coordinates. If a device node is participating in a supercall, it belongs to the zone in which that supercall is running.

At any given time, each node executes a single JAMScript application and belongs to a single zone. The set of nodes in a zone changes dynamically as edge and device nodes start or fail. An edge node joins a zone on start-up and leaves the zone only if it fails. An edge node's zone does not change while it is running. A device node may join or leave a zone due to mobility. However, while a device participates in a supercall, its zone does not change even if it moves to another area.

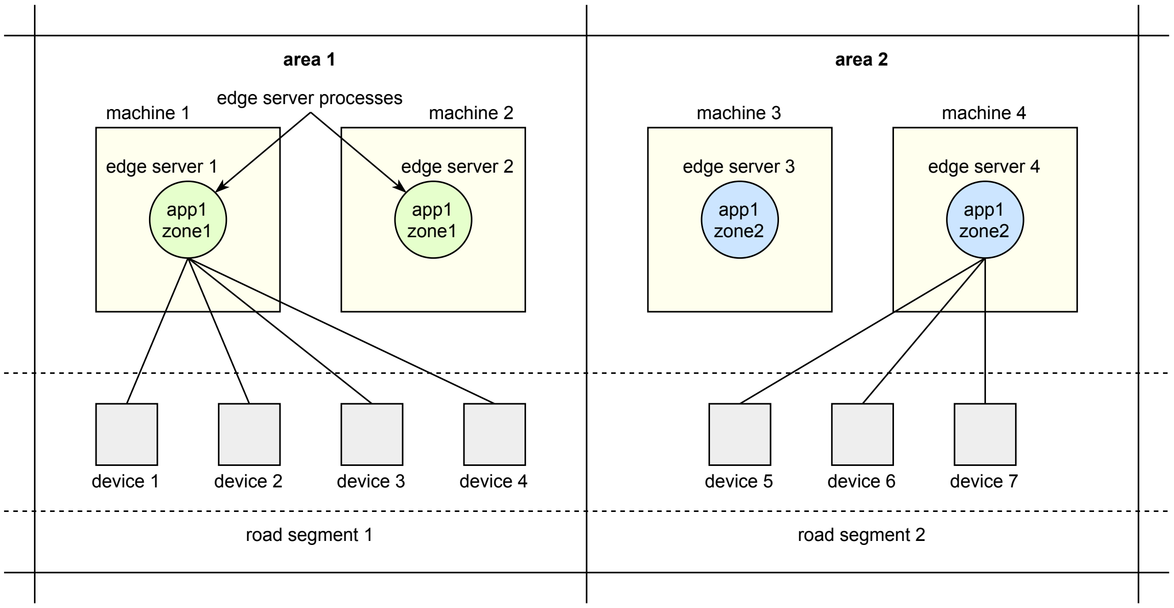


Figure 3.1: Area partitioning.

3.2.2 Zone Definition

For a given application and instant in time, a zone Z_i is defined as the set of nodes satisfying the zone membership function z_i, i.e. $Z_i = \{\text{node} : z_i(\text{node}) = 1\}$. The output of the function is 1 (true) if the node belongs to the zone and 0 (false) otherwise. Here a node is considered as a tuple of properties (xCoord, yCoord, nodeType, zoneConfig, inSupercall, zoneSupercall). The area associated with zone Z_i is denoted by A_i. The zone membership function is defined as follows:

$$z_i(\text{node}) = \begin{cases} 1 & \text{nodeType} = \text{edge} \land \text{zoneConfig} = i \\ 0 & \text{nodeType} = \text{edge} \land \text{zoneConfig} \neq i \\ 1 & \text{nodeType} = \text{device} \land \text{inSupercall} = 0 \land (\text{xCoord}, \text{yCoord}) \in A_i \\ 0 & \text{nodeType} = \text{device} \land \text{inSupercall} = 0 \land (\text{xCoord}, \text{yCoord}) \notin A_i \\ 1 & \text{nodeType} = \text{device} \land \text{inSupercall} = 1 \land \text{zoneSupercall} = i \\ 0 & \text{nodeType} = \text{device} \land \text{inSupercall} = 1 \land \text{zoneSupercall} \neq i \end{cases}$$

The xCoord and yCoord properties refer to the physical two-dimensional coordinates of the node. The nodeType property indicates whether the node is an edge server or device. For an edge node, zoneConfig specifies the zone configured on start-up. For a device node, the inSupercall property indicates whether the device is currently participating in a supercall, and if so, zoneSupercall specifies the zone associated with the supercall.

Each edge server process logically belongs to a single (application, zone) pair regardless of the physical location of its host machine. Typically the host machine is physically located inside or close to the area associated with the zone. An edge server process is started with command-line arguments indicating the (application, zone) pair to which it belongs. It is made aware of the other edge server processes in its zone through the JAMScript runtime's node discovery mechanism. In the following sections, the term *edge server* refers to an *edge server process*.

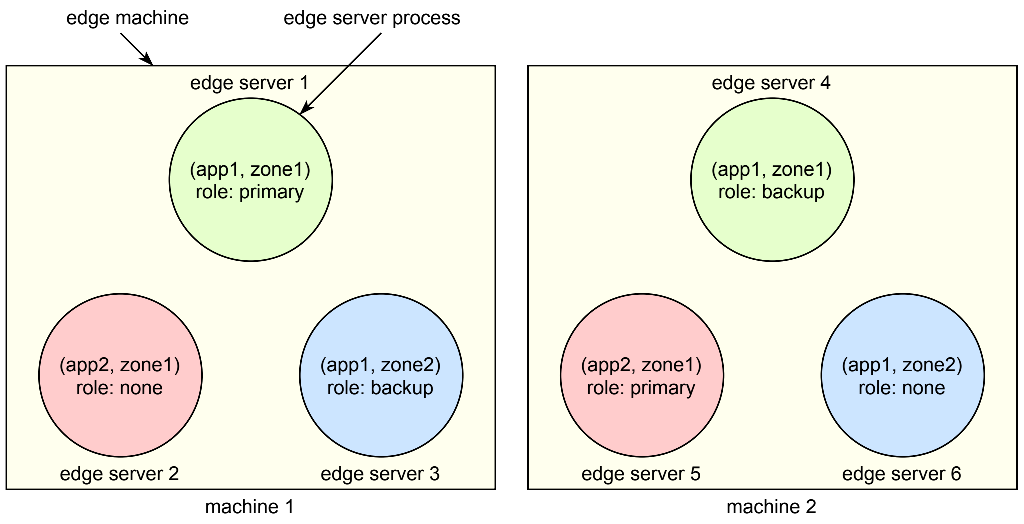


Figure 3.2: Edge machines and processes.

3.3 Zone Edge Servers

Each zone contains one or more edge servers. One of them is designated as primary and another one – as backup. The primary edge server performs computations originating from device nodes and intended to run on an edge node. To save computing resources, these computations are executed only on the primary edge server. The backup takes over when the primary fails. The primary and backup edge servers are chosen using an edge server selection policy which takes into account characteristics (e.g., CPU load, available memory, network latency) of the machines on which the edge servers are running. Each edge server is aware of all other edge servers in the same zone and is notified when the primary or backup edge server changes.

Each edge server in the zone can be either a primary edge server, backup edge server, or edge server with no special role assigned. During the course of the JAMScript program's execution, the selected primary and backup edge servers may change at specific execution points due to edge server failures.

Each device in the zone is made aware of the selected primary and backup edge servers. At any given time, all devices in the zone are connected to the primary edge server. The data that the devices generate using JAMScript's logger construct is propagated by the runtime only to the primary edge server. When the primary edge server changes, all devices connect to the new primary edge server.

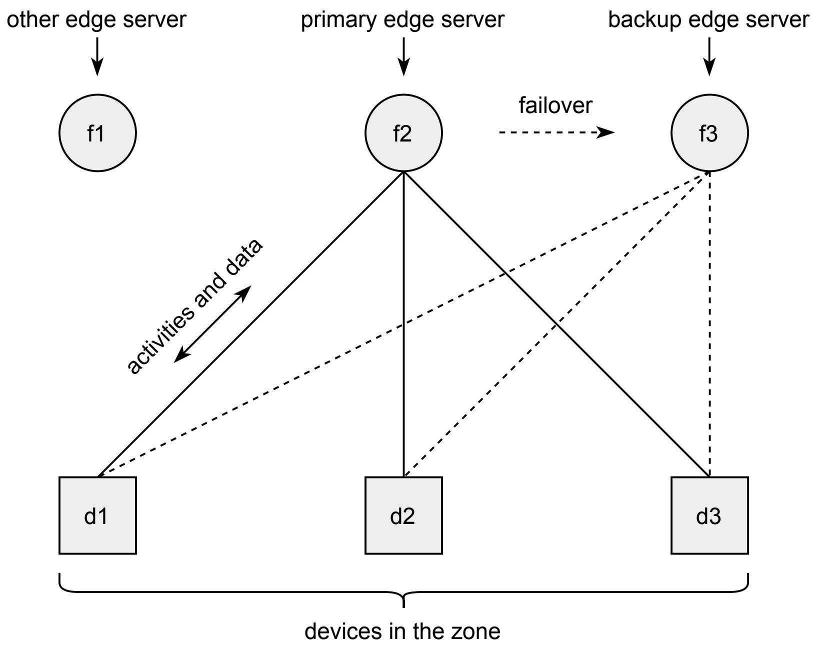


Figure 3.3: Zone edge servers.

3.3.1 Primary Edge Server

At any given time, a single edge server called the primary edge server serves an (application, zone) pair. All devices belonging to the same (application, zone) pair are connected to the primary edge server. The primary edge server performs the zone's computations and stores the zone's data. It sends and receives requests for remote function invocations (activities) and data to and from the devices. It also transforms the data generated by the devices.

The computations performed by the primary edge server are not executed by other edge servers in the zone. Since redundant computations are avoided, the edge resources available to all applications and zones are maximized. To avoid having a single point of failure, the other edge servers in the zone participate in the fault tolerance mechanism employed when the primary edge server fails. The primary edge server is chosen according to the edge server selection policy during application start-up. A new primary edge server is also chosen in case of primary edge server failure.

3.3.2 Backup Edge Server

In addition to having a primary edge server, each (application, zone) pair has a backup edge server. The purpose of the backup edge server is to take over execution when the primary edge server fails. The primary and backup edge servers run on different machines to ensure that their failures are independent.

If there is only one edge server in the zone, it becomes primary and there is no backup. If there are two edge servers in the zone, the edge server which has started first becomes primary and the other one becomes backup. The remaining edge servers have no special role assigned. Each edge server is notified about the edge servers selected as primary and backup. When the primary edge server fails, the backup edge server is selected as the new primary. A new backup is chosen using the edge server selection policy.

3.3.3 Edge Server Selection Policies

The goal of the edge server selection policy is to assist in selecting primary and backup edge servers while aiming for optimal resource utilization. The edge server selection policy ranks active edge servers while taking into account factors like machine load, latency, and bandwidth. Each edge server is aware of all active edge servers in the zone through the JAMScript runtime's node discovery mechanism. Each edge server periodically sends a message to the other edge servers in the zone to share information about its resource utilization.

The edge server selection policy is used when selecting primary and backup edge servers in different scenarios. A new primary is chosen during application bootstrapping. A new backup is selected during application bootstrapping, when the primary fails and the backup takes over as primary, and when the backup fails.

The edge server selection policy is configured before the application starts and does not change during the application's execution. There are three policies. The first policy is based on a machine learning (ML) model which is constantly updated and available on each edge server. The second policy generates a ranking based on a load function calculation with different weights for machine load, latency, bandwidth, etc. The third policy ranks all active edge servers in a random order and is used as a baseline.

Table 3.2: Example edge server ranking produced by an edge server selection policy.

rank	edge server id	load (%)	latency (ms)	bandwidth (Mbps)
1	edge2	12	2	370
2	edge3	10	5	320
3	edge1	28	13	260
4	edge4	65	7	290

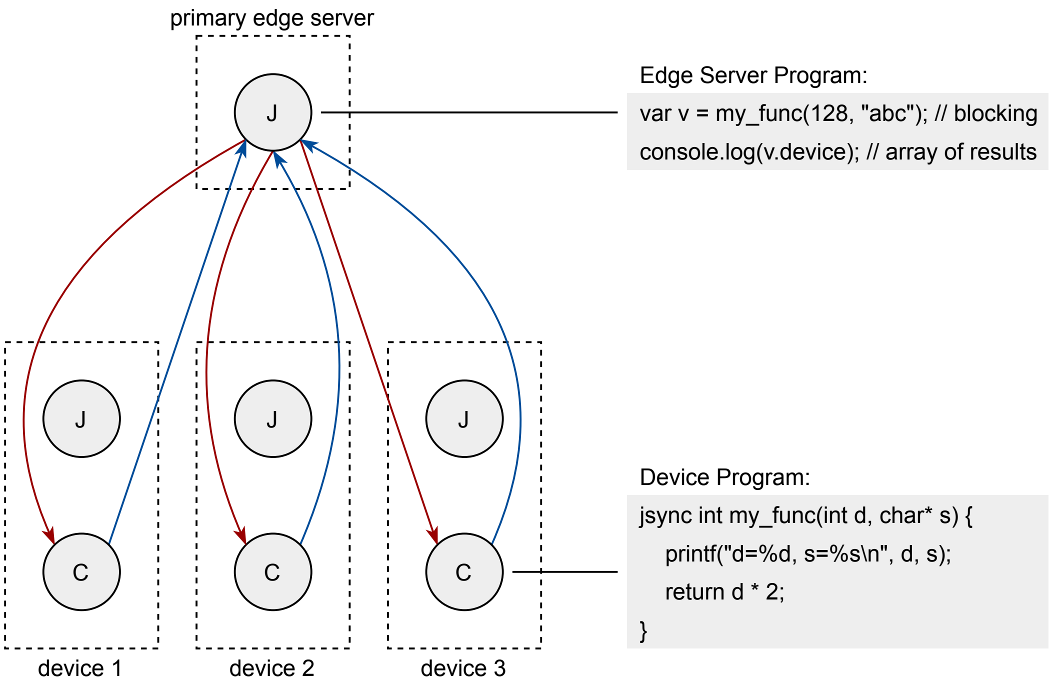
Chapter 4

Supercalls

In vehicular networks, it is often desirable to perform a computation simultaneously on a collection of devices. For example, a roadside unit (controller) may want to inform all vehicles (workers) of the presence of an obstacle on a road segment. A tower (controller) may want to synchronize all drones (workers) so that they fly in a formation. The group of drones flying together must start executing a turn command at exactly the same time.

One motivation for defining zones in JAMScript is the execution of collective synchronized computations. A zone groups together nearby edge servers and devices running the same application. A supercall represents a collective computation in the zone which requires execution synchronization and fault tolerance.

The supercall synchronizes computation execution and data generation across the device nodes in a zone. A typical scenario involves a computation initiated at an edge node and executed at multiple device nodes (e.g., J2C sync activity in JAMScript). The execution of the computation at the devices must be synchronized (i.e. start at the same time). The data generated at the devices is time-aligned and propagated to the edge where it is transformed. The supercall allows the edge server to identify the data generated from devices executing the same computation at the same time.



```
var v = my_func(128, "abc"); // blocking
console.log(v.device); // array of results
```

```
jsync int my_func(int d, char* s) {
    printf("d=%d, s=%s\n", d, s);
    return d * 2;
}
```

Figure 4.1: Synchronous J2C activity in JAMScript.

4.1 Supercall Overview

A supercall is a sequence of executions of a distributed synchronized computation on a set of nodes (primary edge server and devices) producing a result which includes data collected from device nodes as well as data generated at a primary edge node. Performing the computation involves running a distributed function with a list of arguments. The function and arguments are the same for each computation execution.

Each supercall consists of one or more computation executions. The first execution is called the initial execution. The remaining executions are re-executions. If the initial execution succeeds, there are no re-executions. All executions but the last one are unsuccessful. The last execution succeeds and produces the result.

Each execution runs on a primary edge node and a set of device nodes. The set of devices participating in an execution is a subset of all devices in the zone at a particular moment. A device node in the zone may choose not to participate in the computation and thus will not be a part of the execution.

Each computation execution may run on a different set of nodes. This is due to the following reasons. During the program's execution, different edge nodes might be selected as a primary edge server. Device nodes join and leave the zone due to device mobility. Each device node can only be a part of a single computation execution at a time but may belong to many different ones as the program's execution progresses.

When a computation succeeds, the associated supercall ends. The primary and backup edge servers remain unchanged for the initial execution of the next supercall. New primary and backup edge servers are selected in case of edge server failures.

A supercall may have various preconditions. For example, a supercall may require a minimum number or proportion of devices to confirm participation in order to proceed with the computation. If the supercall's preconditions are satisfied, the primary edge server runs the controller side of the computation. The devices which have confirmed participation run the worker side of the computation. The primary edge server collects the results from the devices. If the supercall's preconditions are not satisfied, no computations are performed and the supercall returns an error. The supercall also returns an error if a device which has confirmed participation fails.

Supercalls allow the developer to control the scope of fault tolerance in the JAMScript application. A computation which requires execution synchronization among the participating devices and fault tolerance is implemented with a supercall. It is automatically re-executed after an edge server failure. A computation which requires only execution synchronization without reliability guarantees is implemented with a synchronous controller-to-worker activity to avoid the overhead of the supercall fault tolerance scheme.

4.2 Supercall Definition

A supercall c_i is defined as a triple $(u_i,\ e_i,\ r_i)$ where u_i is a computation, e_i are executions of the computation, and r_i is the result of the computation. The computation u_i is a pair $(p_i,\ a_i)$ where p_i is a distributed function and a_i are its arguments. The executions e_i are represented by a tuple $(e_{i,0},\ e_{i,1},\ \ldots,\ e_{i,k})$. If there is no primary edge server failure, there is only one execution. If there is a primary edge server failure, there are two or more executions. All executions except the last one are unsuccessful. Only the last execution succeeds. The result r_i is a pair $(s_i,\ g_i)$ where s_i are the results collected from the devices and g_i is the data generated at the primary edge server during the last execution.

Each execution $e_{i,j}$ of computation u_i of supercall c_i is represented by a pair $(f_{i,j},\ d_{i,j})$ where $f_{i,j}$ is the zone edge server pair and $d_{i,j}$ is the set of devices participating in the execution. The primary and backup edge servers are denoted by $f^P_{i,j}$ and $f^S_{i,j}$ respectively. The runtime provides the following guarantees between executions. The primary $f^P_{i,j+1}$ of the next execution is the backup $f^S_{i,j}$ of the previous execution. The backup $f^S_{i,j+1}$ of the next execution is chosen using the edge server selection policy. For execution $e_{i,0}$, the device set $d_{i,0}$ is a subset of all devices in the zone. For execution $e_{i,j+1}$, the device set $d_{i,j+1}$ is a subset of the device set $d_{i,j}$ of the previous execution. The notation is summarized as follows:

$$\underbrace{c_i}_{\text{supercall}} = (\underbrace{u_i}_{\text{computation}},\ \underbrace{e_i}_{\text{executions}},\ \underbrace{r_i}_{\text{result}})$$

$$\underbrace{u_i}_{\text{computation}} = (\underbrace{p_i}_{\text{function}},\ \underbrace{a_i}_{\text{arguments}})$$

$$\underbrace{e_i}_{\text{executions}} = (\underbrace{e_{i,0}, e_{i,1}, \ldots, e_{i,k-1}}_{\text{unsuccessful executions}},\ \underbrace{e_{i,k}}_{\text{successful execution}})$$

$$\underbrace{r_i}_{\text{result}} = (\underbrace{s_i}_{\text{results from devices}},\ \underbrace{g_i}_{\text{data generated at primary edge server}})$$

$$\underbrace{e_{i,j}}_{\text{execution}} = (\underbrace{f_{i,j}}_{\text{zone edge servers}} , \underbrace{d_{i,j}}_{\text{set of devices}})$$

$$\underbrace{f_{i,j}}_{\text{zone edge servers}} = (\underbrace{f_{i,j}^{P}}_{\text{primary edge server}} , \underbrace{f_{i,j}^{S}}_{\text{backup edge server}})$$

$$f_{i,j+1}^{P} = f_{i,j}^{S}$$

$$d_{i,0} \subseteq \text{all devices in the zone}$$

$$d_{i,j+1} \subseteq d_{i,j}$$

4.3 Bulk Synchronous Parallel versus Supercalls

The bulk synchronous parallel (BSP) model [45] is a bridging model for parallel computations. Bridging models are used for designing algorithms and making reliable predictions about their performance. The BSP model consists of the following components: nodes (processors) that can perform local computations, a network that allows the nodes to exchanges messages, and a mechanism for synchronizing all nodes.

A BSP program is divided into supersteps. As shown in Figure 4.2, each superstep has three phases: computation, communication, and barrier synchronization. During the computation phase, each node performs operations on local data as well as data generated during previous supersteps. During the communication phase, each node sends and receives messages to and from other nodes. The barrier synchronization phase ensures that all nodes have sent and received their intended messages before starting the next superstep. The execution time of a superstep is determined by the execution time of the computation and communication phases of the slowest node.

A BSP program is a sequence of supersteps. In contrast, a JAMScript program consists of supercalls and othercalls as illustrated in Figure 4.3. BSP supersteps cannot overlap. Supercalls are also non-overlapping and two supercalls cannot execute concurrently. However, in JAMScript other computations called othercalls can run while a supercall is in progress as well as between supercalls.

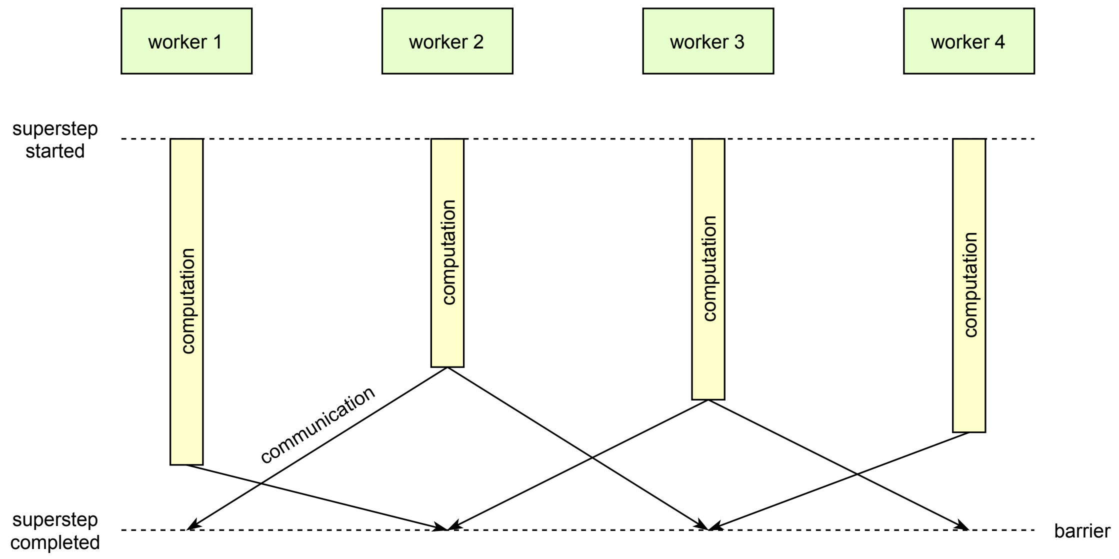


Figure 4.2: Bulk synchronous parallel model.

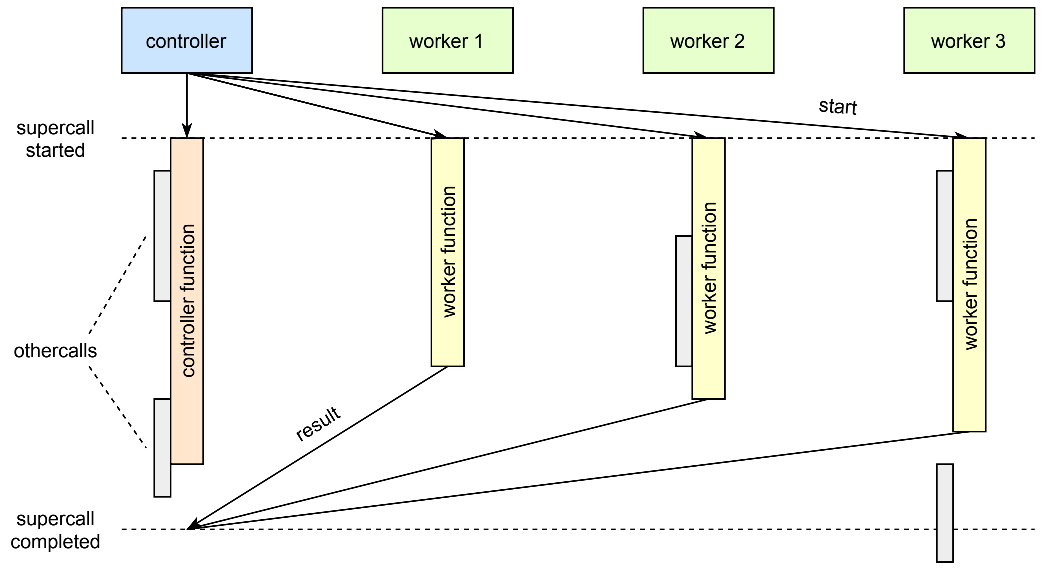


Figure 4.3: Supercall model.

In the BSP model, all nodes performing superstep computations are of the same type. In contrast, JAMScript differentiates between controller nodes (edge servers) and worker nodes (devices). A supercall is initiated at an edge server and triggers computations at both the edge server (controller function) and the devices (worker function). Local computations during a superstep are performed in parallel on all nodes. Similarly, the worker function executes on all devices at the same time and runs concurrently with the controller function executing on the edge server.

In BSP, each superstep begins with a local computation phase followed by a communication phase. Local computations cannot use data generated at other nodes during the same superstep. In contrast, while a JAMScript supercall is executing, computations and data exchanges happen simultaneously using mechanisms like loggers and broadcasters. Data generated at the devices during a supercall can be used during the same supercall in the controller function running at the edge server.

The BSP model ensures that the local computations performed during a particular superstep can access the data from all previous supersteps. The start of a superstep can be delayed until the data from the previous one is available. In JAMScript, each supercall also has access to the data generated by all previously executed supercalls. A supercall cannot start until the data layer has finished replicating the data to the backup edge server.

The barrier synchronization performed at the end of a superstep is similar to the mechanism which determines when a JAMScript supercall completes. In BSP, when a node reaches the barrier, it waits for all other nodes to also reach the barrier. In a similar fashion, a supercall completes when the edge server and all devices have finished their computations. Another difference between supersteps and supercalls is that, unlike a BSP superstep which always executes, a supercall runs only if its preconditions (e.g., quorum of participating devices) are satisfied.

4.4 Programming with Supercalls

In JAMScript, the programmer launches a supercall by calling a special J2C sync activity (`jsync_ctx` activity) from the edge level. The `jsync_ctx` activity consists of two functions: a worker function and a controller function. The worker function runs on all devices that agree to participate in the supercall. Similarly to a regular J2C sync activity, all devices start executing the worker function at the same time. The controller function runs on the primary edge server. The other edge servers in the zone do not perform any computation. When the supercall completes, the primary edge server shares the result of the supercall with the other edge servers in the zone.

The worker and controller functions run in parallel and can call local and remote functions synchronously or asynchronously. The worker function can call a function executing at the edge level (e.g., C2J sync/async activity) and send data to the edge using JAMScript's logger construct. The controller function can call a function executing at the device level (e.g., J2C async activity) and send data to the devices using JAMScript's broadcaster construct. Calls to synchronous activities (e.g., J2C sync activity) from the controller function are not allowed. JAMScript does not support nested synchronization.

The `jsync_ctx` activity returns a JavaScript promise. The promise is settled in two cases: (a) when both the worker and controller functions complete at all nodes participating in the supercall; (b) if the computation cannot be started or there is an error during its execution. In case (a), the promise is fulfilled and the result is an array containing the values returned by the worker function running on the devices. The controller function does not return a value. In case (b), the promise is rejected and the error contains an error message. An error is generated if the supercall's preconditions are not met (e.g., there are not enough workers to reach a quorum) or devices fail during the computation (see Section 5.1.5).

Listing 4.1 shows an example of an edge server program. The condition in the `if` statement on line 1 ensures that the supercall is launched from the edge level. The supercall is launched by calling the `my_func` function on line 5 after a delay of 10 seconds (line 17). The `my_func` function can have an arbitrary number of parameters as needed. The `callback` argument supplied last is the controller function defined on lines 21–23. The `context` parameter of the `callback` function on line 21 represents the computing context created by the supercall. Its value is automatically supplied by the JAMScript runtime. On line 7, the program handles the two possible outcomes of the supercall (success and failure) by calling the `then` method of the returned promise.

Listing 4.1: Edge server program with controller function and supercall invocation.

```
1  if (jsys.type === "fog") {
2      // run only at the edge level
3      setTimeout(() => {
4          // launch a supercall
5          var c = my_func(128, "abc", callback);
6          // handle the two possible outcomes
7          c.then(
8              res => {
9                  // res is an array of device results
10                 console.log(res);
11             },
12             err => {
13                 // err is an error message
14                 console.log(err);
15             }
16         );
17     }, 10000);
18 }
19
20 // controller function
21 function callback(context) {
22     console.log("context:", context);
23 }
```

Listing 4.2 shows the corresponding device program. The device program contains the definition of the worker function on lines 4–7. All devices participating in the supercall start executing it at the same time. The values for parameters `d` and `s` on line 4 are determined at the primary edge server (line 5 of Listing 4.1) and are propagated by the JAMScript runtime to the devices. In this example, each device returns the value $128 \times 2 = 256$ (line 6). As a result, the `res` array on line 10 of Listing 4.1 contains the value 256 repeated as many times as there are devices participating in the supercall.

Listing 4.2: Device program containing the worker function of the supercall.

```
1  #include <stdio.h>
2  
3  // worker function
4  jsync_ctx int my_func(int d, char* s) {
5      printf("d=%d, s=%s\n", d, s);
6      return d * 2;
7  }
8  
9  int main(int argc, char* argv[]) {
10     return 0;
11 }
```

4.5 Supercall Implementation

All edge servers execute the server program but only the primary edge server executes supercalls. When the primary edge server reaches a supercall statement, it starts executing the supercall. All non-primary edge servers wait while the primary edge server executes the supercall. When the primary edge server finishes executing the supercall, it shares the result with the other edge servers in the zone. The execution of the program then continues at all edge servers in the zone.

Before executing the supercall, the primary edge server first queries all devices in the zone to determine the participants of the supercall by sending an `EXEC` message. A device confirms or rejects its participation by replying with an acknowledgment (`ACK`) or negative acknowledgment (`NAK`). The primary edge server determines whether a quorum has been reached by comparing the number of `ACKs` with the number of `NAKs` it has received from the devices. The execution of the supercall proceeds if and only if the proportion of `ACKs` is greater than a specified threshold and the other supercall preconditions are satisfied.

The primary edge server then waits for all participants to become ready to start computing. A device which has confirmed participation by sending an `ACK` message indicates its readiness to begin the computation by sending a `READY` message. A participating device may not be ready to start computing if it is currently performing another computation. When all participating devices are ready, the primary edge server sends a `START` message to begin executing the worker function. The primary edge server also begins executing the controller function. When a device finishes executing the worker function, it sends the result back to the primary edge server. When the primary edge server has finished executing the controller function and has collected results from all participants, the supercall completes. The primary edge server shares the result of the supercall (array of device results) with the other edge servers in the zone through the supercall log described in the next section.

Data generated at device nodes and propagated by the JAMScript runtime to the primary edge server during a supercall is replicated by the underlying data layer to the backup edge server. Two copies of the data generated at the primary edge server during the current and previous supercalls are always kept: one at the primary edge server and another one at the backup edge server. This is necessary to ensure supercall recovery in case of primary or backup edge server failure.

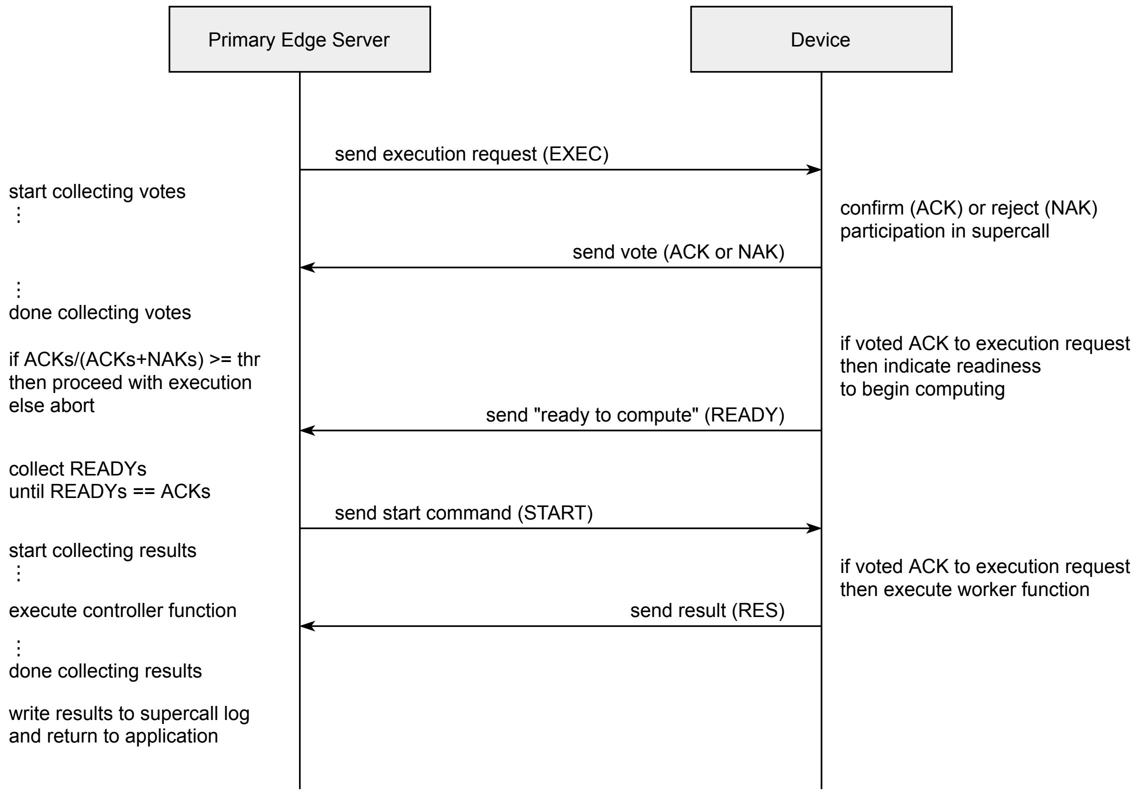

Primary Edge Server
Device
send execution request (EXEC)
start collecting votes
confirm (ACK) or reject (NAK) participation in supercall
send vote (ACK or NAK)
done collecting votes
if ACKs/(ACKs+NAKs) >= thr then proceed with execution else abort
if voted ACK to execution request then indicate readiness to begin computing
send "ready to compute" (READY)
collect READYs until READYs == ACKs
send start command (START)
start collecting results
if voted ACK to execution request then execute worker function
execute controller function
send result (RES)
done collecting results
write results to supercall log and return to application

Figure 4.4: Supercall execution.

4.5.1 Supercall Log

All edge servers in a zone exchange information about the zone edge servers (primary and backup) as well as results of supercall execution through a supercall log. Only the primary edge server writes records to the log. All other edge servers in the zone read from the log. The log is persistent and accessible upon server restart.

The zone edge servers are selected on start-up and in case of primary or backup edge server failure. When the primary edge server fails, the backup becomes primary. When the backup edge server fails, the primary selects a new backup according to the edge server selection policy. When the primary or backup edge server changes, the primary writes a log record with the new edge server pair. All non-primary edge servers in the zone retrieve the new edge server pair from the log record. An edge server learns about being selected as backup through this mechanism.

Upon reaching a supercall statement in the program, all non-primary edge servers wait for the primary edge server to execute the supercall and share the result. When the primary edge server finishes executing the supercall, it writes a log record with the result. All non-primary edge servers read the result of the supercall from the supercall log and continue the execution of the program.

The supercall log can be either centralized or distributed. In a centralized implementation, the supercall log uses a data store located at the cloud (e.g., Redis). All edge servers must be connected to the cloud at all times. The read/write speed depends on the latency of accessing the cloud. In a distributed implementation, the supercall log is replicated at each edge server using a consensus algorithm (e.g., Paxos, Raft). The cloud is neither necessary nor involved. The read/write speed depends on the number of messages exchanged for consensus and the network bandwidth.

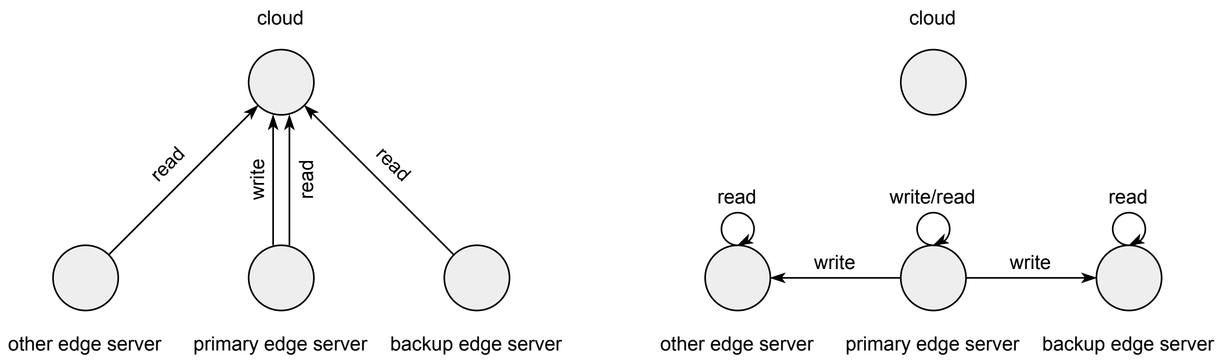


Figure 4.5: Centralized versus distributed supercall log.

4.5.2 Log Record Types

The supercall log consists of a sequence of records. Records can only be appended to the log. Records cannot be removed or updated. Each record has different fields. Some fields are common to all records and others are relevant only to records of a specific type.

The following fields are common to all records: record type, supercall id, primary edge server, and backup edge server. The supercall id is a unique id associated with the current supercall (`jsync_ctx` activity). The primary and backup edge servers are the zone edge servers that are currently selected.

There are three record types: `START`, `RECONFIG`, and `DONE`. In addition to the fields that are common to all records, the `DONE` record has two extra fields: result and error. The result field contains an array with the results collected from the devices after a successful execution of the `jsync_ctx` activity. The error field contains an error message if the activity has failed.

Table 4.1: Supercall log record types.

record type	supercall id	primary edge server	backup edge server	result	error
`START`	$\surd$	$\surd$	$\surd$		
`RECONFIG`	$\surd$	$\surd$	$\surd$		
`DONE`	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

4.5.3 Log Record Sequences

The primary edge server writes a `START` record when the system is ready to start executing the next supercall or re-executing the current supercall after recovering from a failure. A `RECONFIG` record is written whenever the zone edge server pair changes after a primary or backup edge server failure. The primary edge server writes a `DONE` record with the results from the devices after finishing the execution of the `jsync_ctx` activity.

If there is no primary edge server failure during supercall execution, two records are added to the supercall log: `START` and `DONE`. If the `jsync_ctx` activity has succeeded, the result field of the `DONE` record contains the results collected from the devices and the error field is empty (see Table 4.2). If the `jsync_ctx` activity has failed, the result field of the `DONE` record is empty and the error field contains the error message (see Table 4.3).

Table 4.2: Record sequence for supercall execution without failure.

record type	supercall id	primary	backup	result	error
`START`	1	edge1	edge2		
`DONE`	1	edge1	edge2	$[v_1, v_2, \ldots, v_n]$	

Table 4.3: Record sequence for supercall execution with device failure.

record type	supercall id	primary	backup	result	error
`START`	2	edge1	edge2		
`DONE`	2	edge1	edge2		"timeout"

If there is a primary edge server failure during supercall execution, four records are added to the supercall log: `START`, `RECONFIG`, `START`, and `DONE` (see Table 4.4). The `RECONFIG` record stores the new zone edge server pair. The second `START` record is written by the new primary edge server when the system has recovered from the failure and is ready to start re-executing the supercall.

Table 4.4: Record sequence for supercall execution with primary edge server failure.

record type	supercall id	primary	backup	result	error
`START`	3	edge1	edge2		
`RECONFIG`	3	edge2	edge3		
`START`	3	edge2	edge3		
`DONE`	3	edge2	edge3	$[v_1, v_2, \ldots, v_n]$	

Chapter 5

Fault Tolerance

Running a JAMScript program involves computation and data layers. The computation layer consists of synchronous and asynchronous activities. The data layer implements loggers and broadcasters for data exchange between nodes at different levels (device, edge, and cloud). In this discussion, we describe the fault tolerance mechanism associated with the computation layer. Data layer failures are handled with a separate fault tolerance mechanism. Failures in the computation layer include node and network failures as shown in Table 5.1.

Node failures refer to fail-stop (non-Byzantine) edge server and device process failures which may be caused by failing hardware or software. An edge server process may stop running for various reasons even though the machine on which it is deployed might still be running. We distinguish between permanent and temporary node failures. If a node is not restarted after failing, the failure is treated as permanent. If a node fails and is restarted after a short period of time, the failure is temporary.

Network failures can be temporary or intermittent. Temporary network failures make the network unavailable for a relatively long period of time. In such cases, a device node is unable to connect to any edge node in the zone. Intermittent network failures last for a very short period of time and cause a device node to temporarily lose connectivity to the primary edge server.

The JAMScript runtime performs various actions when recovering from computation layer failures. These actions may include selecting a new primary or backup edge server, devices connecting to a new primary edge server, and re-execution of a computation in progress (if there is any). The recovery actions depend on the type of failure, the type and role of the failed node, as well as the execution context at the time of failure. For example, the actions to recover from a primary edge server failure differ from the actions to recover from a backup edge server failure. Device failures are handled differently than edge server failures. The re-execution of a computation is necessary only if there was a computation in progress when the failure occurred.

Table 5.1: Computation layer failures.

Component	Failure Type	Description
Node	Permanent	Node is not restarted after failing
	Temporary	Node is restarted after failing and recovers
Network	Temporary	Network is unavailable for a significant period of time
	Intermittent	Network is unavailable for a very short period of time

5.1 Node Failures

5.1.1 Primary Edge Server Failure

When the backup edge server detects that the primary edge server is down, it immediately selects itself as primary. It then chooses a new backup among all active edge servers in the zone according to the edge server selection policy. The new primary edge server informs all edge servers in the zone about the new edge server pair by writing a `RECONFIG` record in the supercall log.

When a device detects that the primary edge server is down, it connects to the backup edge server which becomes the new primary of the zone. If a device is unable to connect to the backup, it connects to the nearest edge server in the zone. The device retrieves from the nearest edge server information about the new edge server pair. It then connects to the new primary edge server.

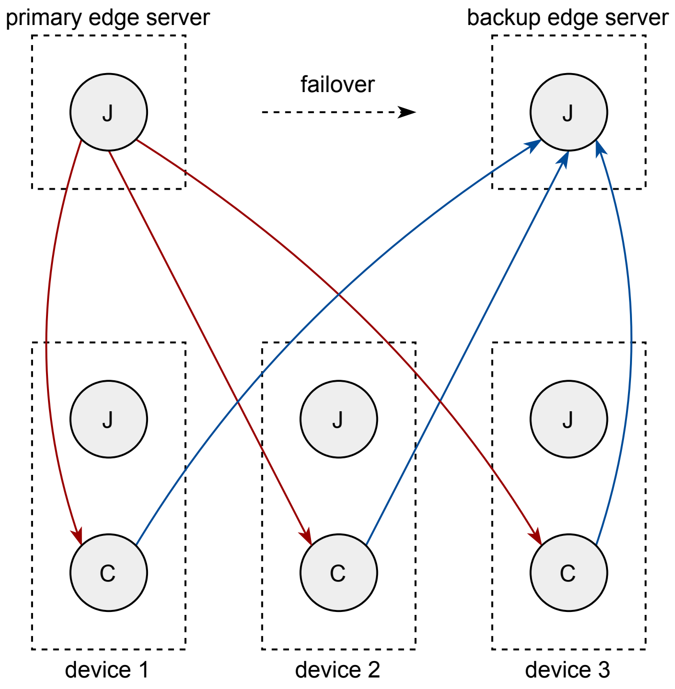


Figure 5.1: Device reconnection after primary edge server failure.

If there is a supercall in progress, the new primary edge server re-executes the supercall. Devices which have joined the zone after the initial execution do not participate in the re-execution. Only devices which were part of the supercall during the initial execution may participate in the re-execution. A device may choose not to participate even if it has been part of the initial execution. Before starting the re-execution, the new primary edge server queries all devices. A device confirms or rejects its participation in the re-execution. During the re-execution of the supercall, the new primary edge server re-runs the controller function and the devices re-run the worker function. The result of the supercall is the result of the re-execution.

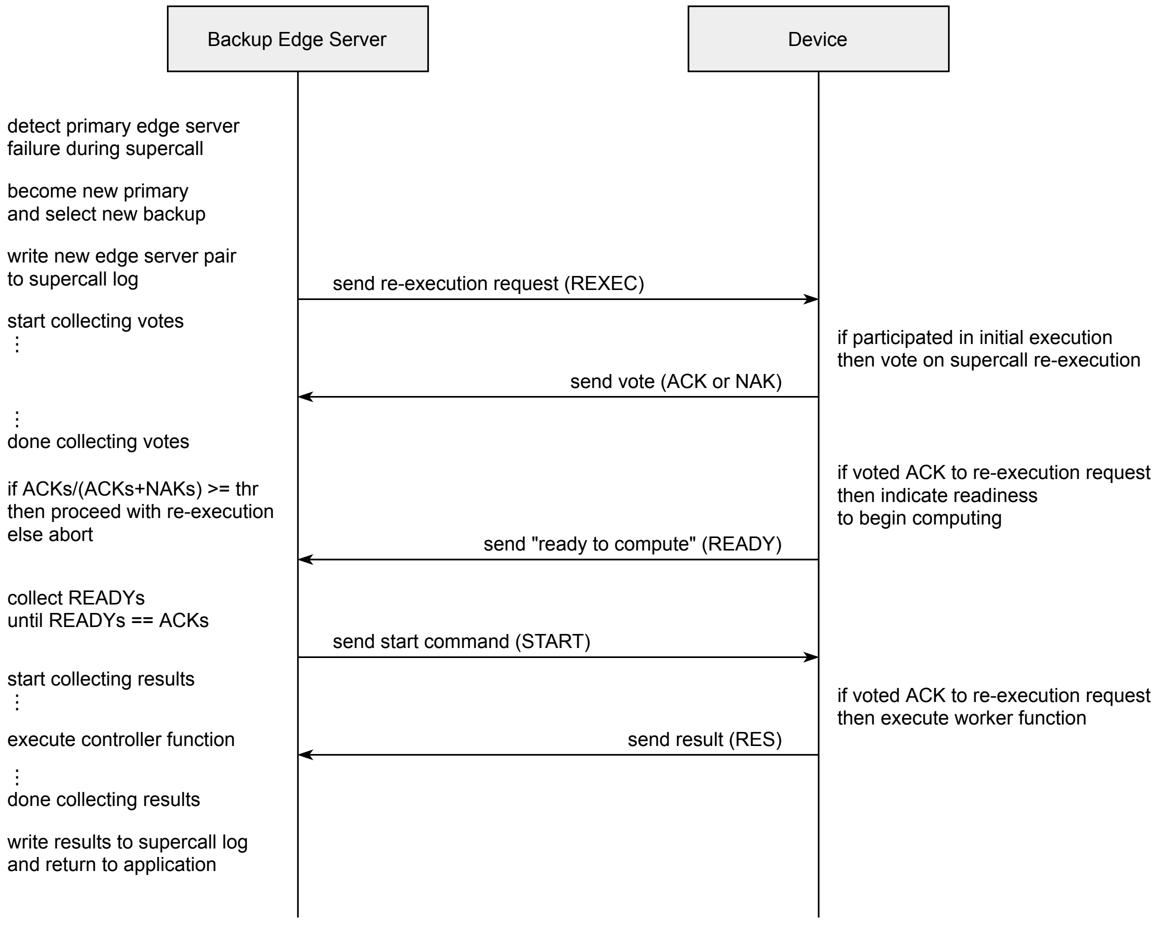

Backup Edge Server
Device
detect primary edge server failure during supercall
become new primary and select new backup
write new edge server pair to supercall log
send re-execution request (REXEC)
start collecting votes
if participated in initial execution then vote on supercall re-execution
send vote (ACK or NAK)
done collecting votes
if voted ACK to re-execution request then indicate readiness to begin computing
if ACKs/(ACKs+NAKs) >= thr then proceed with re-execution else abort
send "ready to compute" (READY)
collect READYs until READYs == ACKs
send start command (START)
start collecting results
if voted ACK to re-execution request then execute worker function
execute controller function
send result (RES)
done collecting results
write results to supercall log and return to application

Figure 5.2: Supercall re-execution after primary edge server failure.

5.1.2 Backup Edge Server Failure

When the primary edge server is not currently executing a supercall and detects that the backup edge server is down, the primary immediately chooses another backup according to the edge server selection policy. It writes a `RECONFIG` record with the new edge server pair to the supercall log. All other edge servers in the zone learn about the newly chosen backup by reading the `RECONFIG` record.

When the primary edge server is executing a supercall and detects that the backup edge server is down, the primary does not take any action until the supercall completes. When the supercall ends, the primary edge server remains as primary for the next supercall and selects a new backup. The new edge server pair is part of the `DONE` record written to the supercall log at the end of the current supercall.

5.1.3 Non-Primary/Non-Backup Edge Server Failure

When an edge server stops running, each edge server in the zone is notified through the JAMScript runtime's node discovery mechanism. No action is taken or necessary when an edge server which is neither primary nor backup goes down. Each edge server updates its set of active edge servers. When a primary needs to select a new backup, the set of currently active edge servers is used as an input to the edge server selection policy.

5.1.4 Edge Server Restart

Upon restarting, a failed edge server has no special role assigned (i.e. it is neither primary nor backup). On start-up, it retrieves the current edge server pair from the supercall log common to all edge servers in the zone. The edge server starts executing the JAMScript program from the beginning. When a previously executed supercall is encountered, its result is read from the supercall log. The result of a completed supercall is always available in the supercall log regardless of whether the execution of the corresponding `jsync_ctx` activity has succeeded or failed. When the program's execution reaches a supercall in progress, it waits for the result to appear in the supercall log. The result is written to the supercall log by the primary edge server.

5.1.5 Device Failure

The effect of a device failure on the system depends on whether the device is part of a supercall when the failure occurs. When a device which is not currently participating in a supercall stops running, the failure does not affect the program's execution on the primary edge server. When a device which is currently participating in a supercall fails, the device failure may cause supercall failure. Whether the supercall succeeds or fails depends on the timing of the failure with respect to the sequence of messages exchanged between the primary edge server and the devices during supercall execution.

If a device fails before confirming its participation in the supercall, it does not take part in the execution of the supercall. When the primary edge server checks whether a quorum has been reached, the device is not considered in the total count of devices. Therefore, the device failure has no effect on whether the supercall succeeds or fails.

If a device fails after confirming its participation in the supercall but before indicating it is ready to start computing, the primary edge server fails the supercall. In order for a supercall computation to start, all devices which have acknowledged participation must report being ready to start computing. Due to the device failure, no device executes the computation associated with the supercall. The supercall completes unsuccessfully without a result.

If a device fails after indicating it is ready to start computing but before returning a result to the primary edge server, the primary edge server sends a message to all devices to start computing (assuming a quorum has been reached). The other devices execute the computation associated with the supercall. Due to the device failure, the primary edge server cannot collect results from all participants which have confirmed being ready to start executing the computation. The supercall completes unsuccessfully without a result.

If a device fails after returning a result to the primary edge server, the supercall completes successfully (assuming results were collected from all participants). The result from the device is part of the result of the supercall. In this case, the device failure has no effect on whether the supercall succeeds or fails.

5.1.6 Device Restart

When a device restarts after a failure, it connects to the nearest edge server in the zone. The device retrieves from it the current edge server pair. After connecting to the primary edge server, the device starts executing the JAMScript program. As described in Section 5.1.5, the device failure before the restart may or may not have caused a supercall failure. There is no supercall failure if there was no supercall in progress when the failure occurred. There is also no supercall failure if the device failed before confirming its participation in a supercall or after returning a result. In case of supercall failure due to device failure, there is no re-execution after the device restarts.

5.2 Network Failures

The effect of a network failure on the system depends on whether the failure is temporary or intermittent. A failure is temporary when the network is unavailable for a relatively long period of time. A failure is intermittent when the network is unavailable for a very short period of time.

5.2.1 Temporary Network Failure

The primary edge server cannot distinguish between a device node failure and a network failure at a device. The behavior of the primary edge server is the same in both cases as described in Section 5.1.5. The device receives a notification about the network failure from the JAMScript runtime's node discovery mechanism. The device does not attempt to connect to any edge server until the network connection is restored. If the device is currently executing a computation as part of a supercall, it may or may not complete the execution (depending on the program). The device then operates autonomously driven by its local controller. When the network connection is restored, the device connects to the nearest edge server in the zone and retrieves the current edge server pair. After connecting to the primary edge server, the device can participate in another supercall.

5.2.2 Intermittent Network Failure

The JAMScript runtime's node discovery mechanism does not notify the device of the network failure. The failure is handled by the underlying protocol used for communicating between the device and the primary edge server. Messages which are not received are retransmitted at specified intervals up to a maximum number of retries. Since the network failure lasts for a very short period of time, eventually the message delivery succeeds.

Chapter 6

Private and Shared Data

Devices at the edge of the network can generate a large volume of data. However, they typically have limited storage and computing resources. Instead of processing the data locally, devices push the data to the edge and cloud. In JAMScript, this is achieved by writing the device data to a logger. The runtime automatically propagates the logger data to the edge and cloud. The data generated at a device is also called raw or original data. The raw data arriving at the edge can then be transformed into derived data.

In JAMScript, there are two kinds of derived data generated at the edge from device data: private data and shared data. Private data is based on data from a single device and is relevant only to that device. Such data can be produced when a device offloads a computation to the edge. Shared data is based on data from multiple devices and is relevant to that collection of devices. Shared data is generated when devices participate in a collective computation. JAMScript provides facilities for creating both private and shared data.

6.1 Private Data

Private data is generated at the primary edge server based only on data from one device. For example, a device may initiate a computation (e.g., C2J sync activity) that executes at the edge level and transforms the device data (e.g., by calculating a running average). Private data is specific to a device and other devices cannot access it. Private data is not associated with a supercall. It can be generated both inside and outside of a supercall. The private data for each device is always available at the primary edge server. When a new primary edge server in the zone is selected, JAMScript's data layer ensures that the new primary edge server has the private data for all devices.

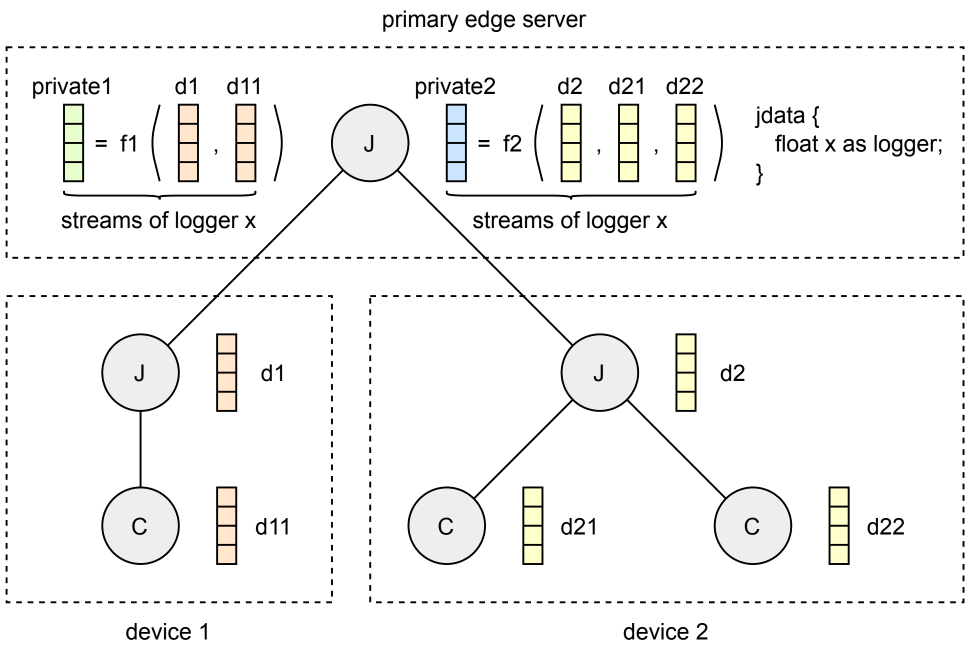


Figure 6.1: Private data generation with two devices.

Figure 6.1 shows private data generation with two devices. At device 1, logger `x` has two data streams: one for the J component (`d1`) and another one for the C component (`d11`). At device 2, logger `x` has three data streams: one for the J component (`d2`) and two for the C components (`d21` and `d22`). At the edge server, logger `x` has all of the device streams plus two private streams (`private1` and `private2`), one for each connected device. The functions `f1` and `f2` represent transformations which produce private streams from the device streams.

6.1.1 Programming with Private Data

To generate private data in JAMScript, the programmer calls the `getPrivateStream` method of a logger at the edge level. The `getPrivateStream` method can be called either from the supercall controller function or from an activity initiated from a device and executing at the edge server (e.g., C2J sync or async activity). The way private data is generated (inside or outside of a supercall) determines the data guarantees provided by the runtime in case of primary edge server failure. The guarantees with regards to private data in different scenarios are discussed in Section 6.3.

Data is written to the private stream by calling its `log` method. The `log` method has one required parameter which is the transformation function and zero or more optional parameters which are passed as arguments to the transformation function. The transformation function takes one required parameter (`device_streams`) and zero or more optional parameters if needed for the computation. The value for the `device_streams` parameter is supplied by the runtime and contains the raw data streams of the device (C and J components). The transformation function must return a value which is written to the private stream.

6.1.2 Example Application

To illustrate the use of private data in JAMScript, we develop a program which builds a historical record of the 10-value moving average of wind speeds at a geographic location. The program runs on a device equipped with a wind speed sensor. The program running at the edge server periodically receives the wind speed reading from the device, calculates the moving average of wind speeds and writes it to the private stream associated with the device. The historical record of the moving average of wind speeds stored in the private stream can then be visualized and analyzed.

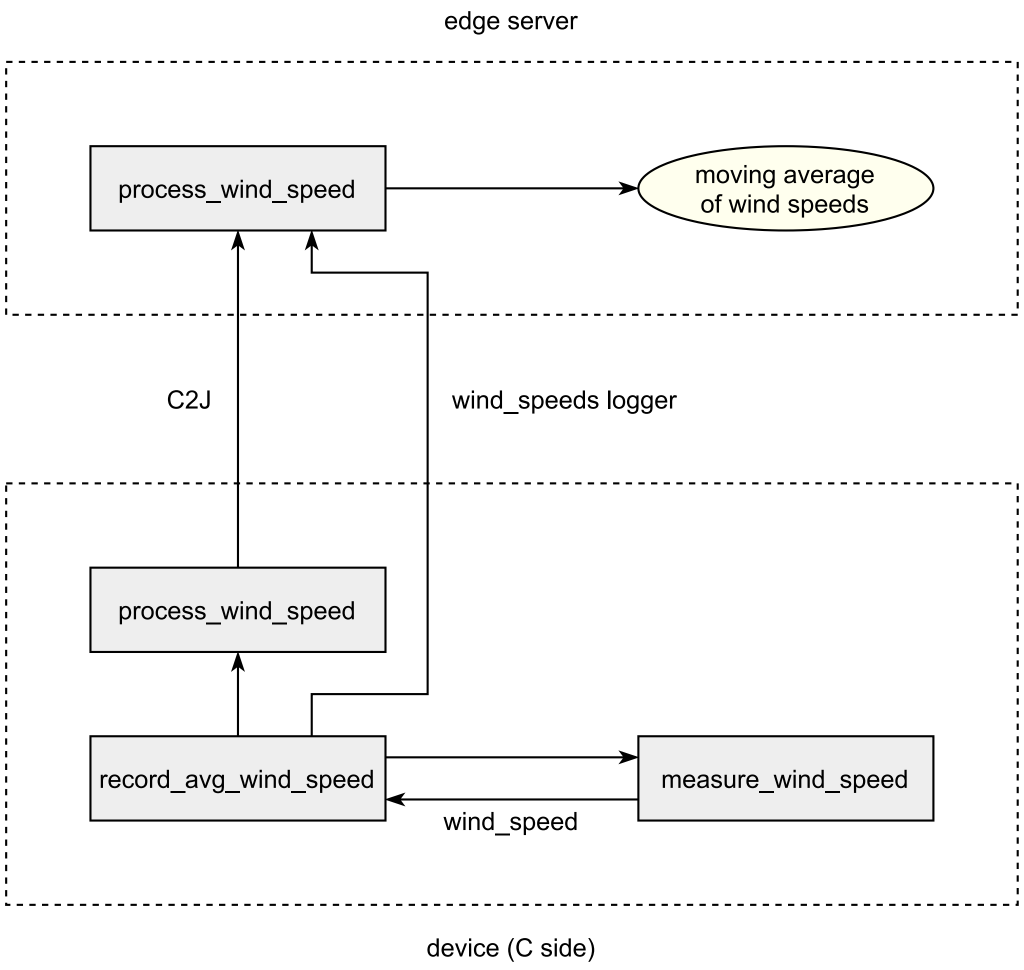


Figure 6.2: Recording the moving average of wind speeds.

Figure 6.2 illustrates the relationships between the main functions involved in building a historical record of the 10-value moving average of wind speeds at a geographic location. The `record_avg_wind_speed` function runs on the device and triggers the calculation of a new moving average value. It first obtains a wind speed reading from the device's sensor by calling the `measure_wind_speed` function. The new reading is written to the `wind_speeds` logger. The record_avg_wind_speed function then launches the process_wind_speed C2J activity which executes on the edge server. The process_wind_speed function extracts the 10 most recent values from the wind_speeds logger and calculates a new moving average value of wind speeds.

Listing 6.1 shows a fragment of the device program responsible for building the historical record of the moving average of wind speeds. It runs on the C component of the device. The while loop on line 3 ensures that a new moving average value is calculated and stored periodically. The current wind speed reading is retrieved from the sensor by calling the measure_wind_speed function on line 5. On line 7, the new wind speed value is written to the wind_speeds logger. The device triggers the calculation of the moving average at the edge server by launching the process_wind_speed C2J activity on line 9. The process_wind_speed function is called from the device but executes at the edge server. Finally, after a pause of 5 seconds (line 11) the process repeats.

Listing 6.1: Recording the moving average of wind speeds.

```
1  // called on device program start-up
2  jasync record_avg_wind_speed() {
3      while (1) {
4          // measure the current wind speed
5          float wind_speed = measure_wind_speed();
6          // write to the wind_speeds logger defined on the J side
7          wind_speeds = wind_speed;
8          // use wind speed reading to calculate moving average
9          process_wind_speed(deviceId);
10         // wait 5 seconds before the next measurement
11         jsleep(5000);
12     }
13 }
```

Listing 6.2 shows the implementation of the process_wind_speed C2J activity which obtains the wind speed reading from the device and calculates the moving average. The `process_wind_speed` function first waits 500 ms to receive logger data from the device (line 12). It then retrieves the private stream associated with the `wind_speeds` logger and the device and stores it in the `private_stream` variable (line 15). The `dev_streams` variable on line 16 contains the streams from the J and C components of the device. The stream from the J component is empty. Lines 17–19 retrieve the stream corresponding to the C component. Lines 20–22 extract the last 10 values from the stream and calculate the average. Finally, the average wind speed returned on line 23 is stored in the private stream by calling the `log` method on line 16.

Listing 6.2: Calculating the moving average of wind speeds.

```
1  jdata {
2      // logger used by device to send wind speed to edge server
3      float wind_speeds as logger;
4  }
5
6  jcond {
7      fogOnly: jsys.type == "fog";
8  }
9
10 jasync {fogOnly} function process_wind_speed(deviceId) {
11     // wait for edge server to receive logger data from device
12     jsys.sleep(500);
13
14     // compute moving average and write it to the private stream
15     var private_stream = wind_speeds.getPrivateStream(deviceId);
16     private_stream.log(dev_streams => {
17         var dev_stream = dev_streams.find(
18             stream => stream.lastValue() !== null
19         );
20         var values = dev_stream.n_values(10);
21         var sum = values.reduce((total, val) => total + val, 0);
22         var average = sum / values.length;
23         return average;
24     });
25 }
```

6.2 Shared Data

Shared data is generated at the primary edge server based only on data from devices participating in a supercall. For example, the primary edge server may initiate a computation (e.g., `jsync_ctx` activity) that generates data at the devices and then transforms it at the edge level (e.g., by calculating the average temperature from multiple sensors). The supercall ensures that the device data arriving at the edge is time-aligned. Shared data is relevant to all devices participating in the supercall and is accessible by all of them. JAMScript's data layer guarantees that the shared data generated during the current and previous supercalls is always available at the primary edge server in the zone.

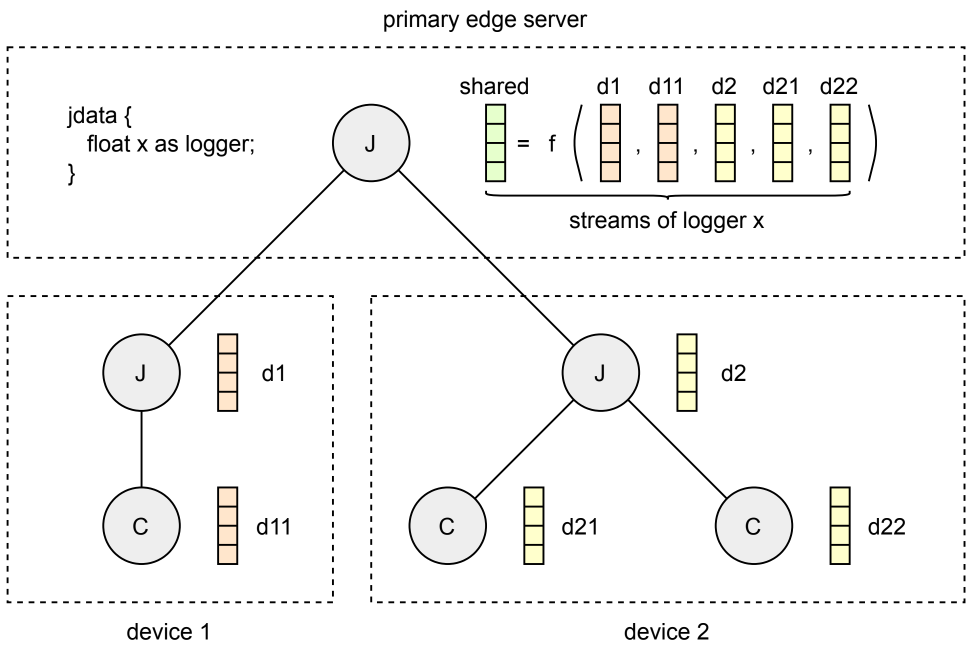


Figure 6.3: Shared data generation with two devices.

Figure 6.3 shows shared data generation from two devices participating in a supercall. At device 1, logger `x` has two data streams: one for the J component (`d1`) and another one for the C component (`d11`). At device 2, logger `x` has three data streams: one for the J component (`d2`) and two for the C components (`d21` and `d22`). At the edge server, logger `x` has all of the device streams plus one shared stream (`shared`) for the supercall. The function `f` represents a transformation which produces a shared stream from the device streams.

6.2.1 Programming with Shared Data

To generate shared data in JAMScript, the programmer calls the `getSharedStream` method of a logger at the edge level. This method can be called either from the supercall controller function or from an activity initiated from a device and executing at the edge server (e.g., C2J sync or async activity). Calling `getSharedStream` from a C2J activity allows devices to read shared data generated during the current and previous supercalls. When `getSharedStream` is called from the supercall controller function, the shared stream it returns is both readable and writable. When `getSharedStream` is called from an activity initiated from a device and executing at the edge server, the shared stream is read-only.

Data is written to the shared stream by calling its `log` method. The `log` method has one required parameter which is the transformation function and zero or more optional parameters which are passed as arguments to the transformation function. The transformation function takes one required parameter (`device_streams`) and zero or more optional parameters if needed for the computation. The value for the `device_streams` parameter is supplied by the runtime and contains the raw data streams of all devices participating in the supercall. The transformation function must return a value which is written to the shared stream.

6.2.2 Example Application

To illustrate how shared data can be used in JAMScript, we create a program which builds a historical record of the average temperature in a smart home. The program runs on several devices which are equipped with temperature sensors and are deployed in different rooms. The program running at the edge server periodically collects temperature readings from the devices, calculates the average temperature in the house and writes it to a shared stream. The historical record of the average temperature in the house stored in the shared stream can then be visualized and analyzed.

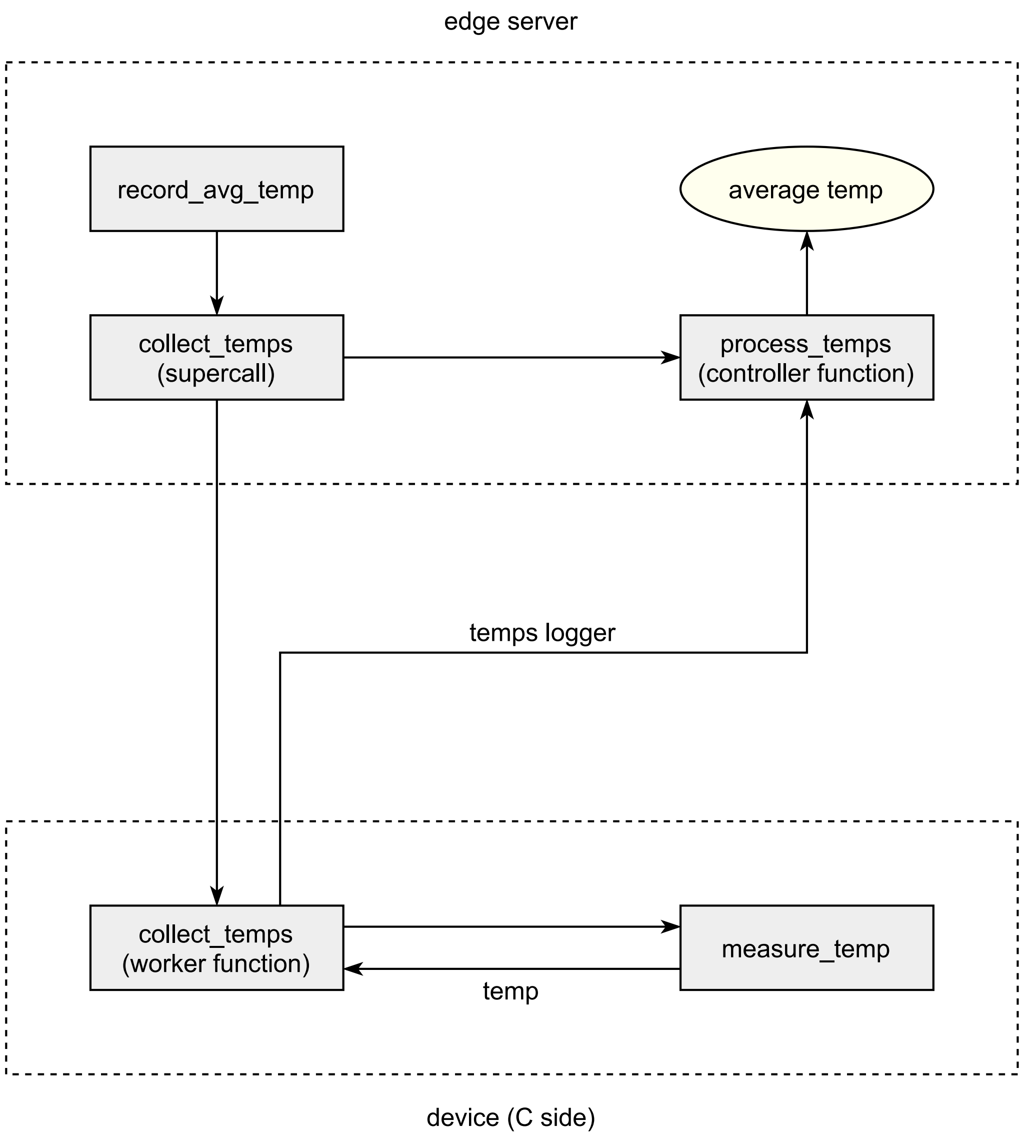


Figure 6.4: Recording the average temperature in the smart home.

Figure 6.4 illustrates the relationships between the main functions involved in building a historical record of the average temperature in the smart home. The `record_avg_temp` function is called initially on edge server program start-up and runs periodically. It launches the `collect_temps` supercall with `process_temps` as a controller function. The supercall triggers the execution of the `collect_temps` worker function on the C side of each device. The worker function calls the `measure_temp` function which obtains the current temperature from the device's sensor. The temperature reading is then written to the `temps` logger. The `process_temps` controller function executing on the edge server collects temperature readings from all devices and calculates the average temperature.

Listing 6.3 shows a fragment of the edge server program responsible for adding a new entry to the historical record of the average temperature. A supercall is launched by calling the `collect_temps` function on line 4. The supercall invocation triggers the execution of a function with the same name (worker function) at the devices. The last argument of `collect_temps` is the controller function `process_temps` defined in Listing 6.5. It executes on the edge server and runs concurrently with the `collect_temps` worker function. On line 6, the program handles the two possible outcomes of the supercall (success or failure) by calling the `then` method of the returned promise. In both cases, another supercall is launched after a delay of 500 ms to record a new average temperature (lines 10 and 15).

Listing 6.3: Recording the average temperature at the edge server.

```
1  // initially called on edge server program start-up
2  function record_avg_temp() {
3      // launch a supercall
4      var c = collect_temps(jsys.id, process_temps);
5      // handle the two possible outcomes
6      c.then(
7          res => {
8              // res is an array of device results
9              console.log(res);
10             setTimeout(record_avg_temp, 500);
11         },
12         err => {
13             // err is an error message
```

```
            console.log(err);
            setTimeout(record_avg_temp, 500);
        }
    );
}
```

Listing 6.4 shows the C side code of the device program which contains the definition of the worker function of the supercall. All devices participating in the supercall start executing it at the same time. The call to the `measure_temp` function on line 4 obtains the current temperature from the device's sensor. The temperature reading is written to the `temps` logger on line 6. The JAMScript runtime automatically propagates the logger data to the edge server.

Listing 6.4: Temperature reading collection at the devices.

```
1 // runs when the supercall is launched at the edge server
2 jsync_ctx int collect_temps(char* id) {
3     // measure the current temperature
4     float temp = measure_temp();
5     // write to the temps logger defined on the J side
6     temps = temp;
7     return 0;
8 }
```

Listing 6.5 shows the implementation of the `process_temps` controller function which collects temperature readings from the devices and calculates the average temperature. First, the controller function waits 500 ms to receive data from the devices (line 10). It then retrieves the shared stream associated with the `temps` logger and the supercall in progress and stores it in the `shared_stream` variable (line 13). The value for the `context` parameter of the `getSharedStream` method is automatically provided by the JAMScript runtime. The `dev_streams` variable on line 14 is a collection of the `temps` logger's data streams from all devices participating in the supercall. Lines 15–17 retrieve the last value from each device stream and calculate the average. Finally, the average temperature returned on line 18 is stored in the shared stream by calling the `log` method on line 14.

Listing 6.5: Calculating the average temperature from the device readings.

```
1  jdata {
2      // logger used by devices to send temps to the edge server
3      float temps as logger;
4  }
5
6  // supercall controller function (edge server computation)
7  // collect temps from devices and calculate average temp
8  function process_temps(context) {
9      // wait for edge server to receive logger data from devices
10     jsys.sleep(500);
11
12     // compute average temp and write it to the shared stream
13     var shared_stream = temps.getSharedStream(context);
14     shared_stream.log(dev_streams => {
15         var values = dev_streams.map(stream => stream.lastValue());
16         var sum = values.reduce((total, val) => total + val, 0);
17         var average = sum / values.length;
18         return average;
19     });
20 }
```

6.3 Data Reliability Guarantees

A JAMScript program stores data by writing to loggers and broadcasters. Each edge server has a separate data store. The data generated by the program and stored using loggers and broadcasters is persisted in the data store of the primary edge server. The data layer asynchronously replicates the data to the backup edge server. Typically, two copies of the data are available: one in the data store of the primary and another one in the data store of the backup. Since the replication is asynchronous, if the primary edge server fails before a data value is replicated to the backup edge server, the data value is lost. To provide data guarantees in a system with asynchronous replication, supercalls are used as checkpoints. A supercall can start executing only when the replication of the data generated during the previous supercalls has completed in which case two copies of the data are available.

The JAMScript runtime provides various data reliability guarantees in case of primary edge server failure. The data guarantees depend on whether the data is generated inside or outside of a supercall. Shared data is always generated inside a supercall because it can be created only from the supercall controller function.

Private data created from the supercall controller function is generated inside a supercall. Private data whose creation is triggered from the supercall worker function is also generated inside a supercall. In this scenario, the supercall worker function launches a C2J sync activity executing at the edge server which creates the private data. Private data created in all other cases is generated outside of a supercall.

Figure 6.5 illustrates the two scenarios for private data generation inside a supercall. Launching the `generate` supercall triggers the execution of the `create_data_ctrl` controller function which generates some private data. The supercall worker function `generate` calls the C2J sync activity `create_data_c2j` which runs at the edge server and also generates some private data.

In case of primary edge server failure, shared data generated from the last N (constant) completed supercalls is recovered from the backup edge server. Private data generated from and between the last N completed supercalls is also recovered from the backup edge server. Shared and private data created from a supercall in progress are regenerated during the re-execution of the supercall. If there is no supercall in progress, private data generated after the end of the last supercall may be lost and cannot be regenerated.

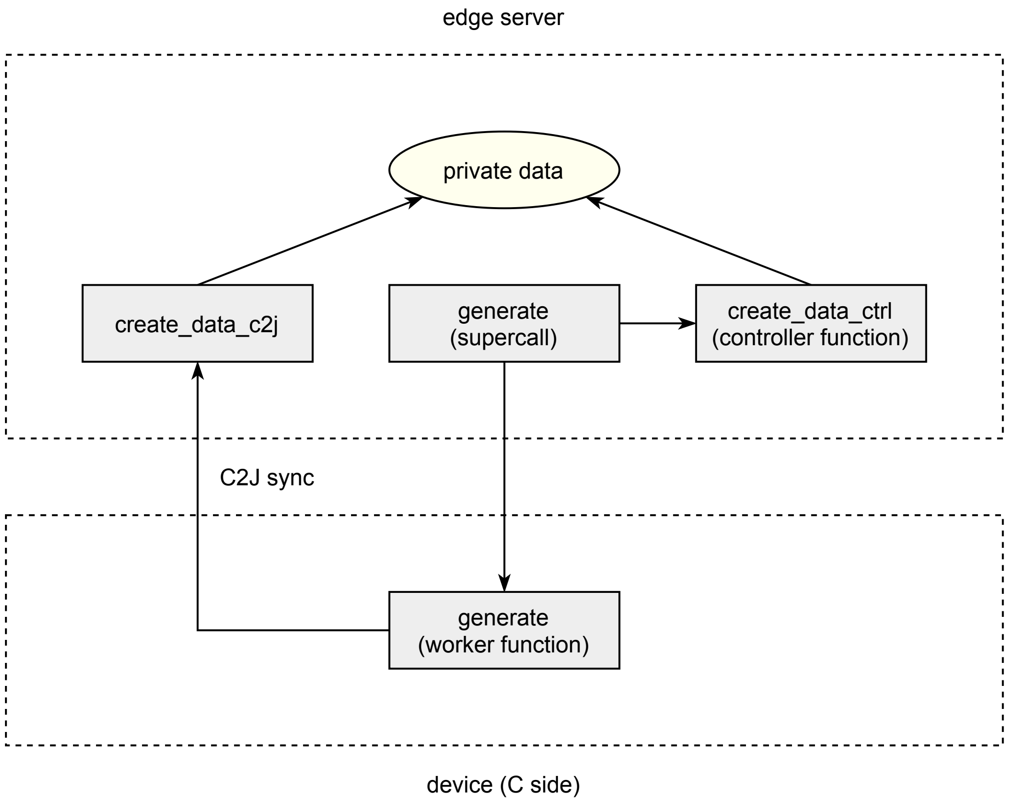


Figure 6.5: Private data generation inside a supercall.

6.4 Shared Data and Machine Learning

The data generated at the edge can be used to produce insights and make predictions using machine learning tools and techniques. To use the generated data as input to machine learning algorithms, it must be stored in a suitable format. Typically, machine learning algorithms operate on feature vectors of fixed size. JAMScript's supercalls and shared streams provide facilities for storing data as feature vectors in a convenient way. Since different devices may participate in each supercall, the developer needs to take extra care to ensure that feature vectors have a fixed size and correct order of values.

We consider a JAMScript program with supercalls. In the supercall worker function, each device writes a value to a logger. In the supercall controller function, the edge server first waits for the runtime to propagate the values from the devices to the edge server. The edge server then reads the last value from the device stream of each participating device and builds a feature vector which is saved to the shared stream. To ensure a consistent order of values in the feature vector, the device streams are processed in ascending order of device id. To ensure that the feature vector has a fixed size, null values are set in positions corresponding to devices which do not participate in the supercall.

For example, we consider a configuration with four devices. For simplicity, we assume that each device has only one raw data stream. During a supercall, each device writes a value to logger x. The edge server creates a feature vector using these values and writes it to the shared stream associated with logger x and the supercall in progress.

All four devices participate in the first supercall. The supercall controller function reads four values from the device streams ordered by device id: v_{11}, v_{21}, v_{31}, and v_{41}. It then writes the feature vector (v_{11}, v_{21}, v_{31}, v_{41}) to the shared stream.

Devices 1 and 3 participate in the second supercall. The supercall controller function reads two values from the device streams ordered by device id: v_{12} and v_{32}. It then writes the feature vector (v_{12}, *null*, v_{32}, *null*) to the shared stream. The null values correspond to devices 2 and 4 which do not participate in the second supercall.

Devices 1, 2, and 4 participate in the third supercall. The supercall controller function reads three values from the device streams ordered by device id: v_{13}, v_{23}, and v_{43}. It then writes the feature vector (v_{13}, v_{23}, *null*, v_{43}) to the shared stream. The null value corresponds to device 3 which does not participate in the third supercall.

Chapter 7

Distributed Multiple-Target Tracking Application

Multiple-target tracking has numerous military and civilian applications in areas such as air defense systems, air traffic control, and drones management. Typically, the monitored area is large and several radar stations are deployed. Multiple targets may be present in the area at any moment. A target might be detected by more than one radar. A distributed target tracking algorithm processes the information collected at the stations and generates a unified view of all tracked objects. The main implementation challenges are reliability, achieving high accuracy, and synchronizing the data collected by the different radars. Edge computing and JAMScript's supercalls are well suited for meeting these challenges.

7.1 Design Requirements and Architecture

We build a distributed multiple-target tracking (DMTT) application which efficiently and accurately tracks objects moving in a physical space. The application runs on several edge servers and devices. One of the edge servers is designated as primary and another one –

as backup. The backup edge server takes over processing when the primary one fails. All devices are connected to the primary edge server. The devices are deployed in the physical space. Each device has a radar which covers a circular area with a particular radius. The areas covered by different devices may overlap.

The edge server builds and maintains a model with information about all moving objects known to the system. Each object in the model has a unique id, coordinates, and velocity vector. The edge server periodically updates the model with information collected from the devices. The model is saved in a persistent data store. The updated model is also periodically broadcasted to the devices.

Devices perform tracking and identification functions. Each device uses the model received from the server as well as data about objects detected by its radar to identify and match these objects to the ones in the model. Objects which have just entered the physical space monitored by the devices are not yet part of the model and are labelled as unknown.

Each device sends to the edge server the ids, coordinates, and velocities of the objects it is tracking. In order to achieve high tracking accuracy independent of the frequency of model updates from the server, each device maintains a local copy of the model and periodically updates the coordinates and velocities of the objects in it based on its radar data.

Due to the fact that the areas covered by different devices overlap, one object might be tracked by multiple devices. This may cause information about the same object to be sent to the server from different devices. The edge server resolves duplicate information. It also assigns ids to all unknown objects and adds them to the model.

The model maintained and updated at the edge server represents synchronized shared state. Synchronized shared state is built at the edge server from data generated at multiple devices at the same point in time. The age of the shared state is determined by the oldest information from any device used to build it. In order for the tracking system to be accurate, the model must have the minimum possible age. The accuracy of the tracking system increases as the age of the model decreases. In the application, the model is built based on tracking information generated simultaneously at the devices.

Another important requirement is support for fault-tolerant computing at the edge. In case of edge server failure, the system must provide a mechanism for automatically generating a new version of the model with the most up-to-date tracking information from the devices. JAMScript's supercall construct satisfies these requirements. It supports shared data and synchronized distributed computations and implements a robust fault tolerance scheme.

Figure 7.1 illustrates a sample deployment with a primary edge server, a backup edge server, and three devices. The areas monitored by the radars of the devices overlap. Device 0 is detecting objects A and B. Device 1 is detecting objects B, C, and D. Device 2 is detecting objects D and E. Objects B and D are each tracked by two devices. The model built at the primary edge server contains five objects: A, B, C, D, and E.

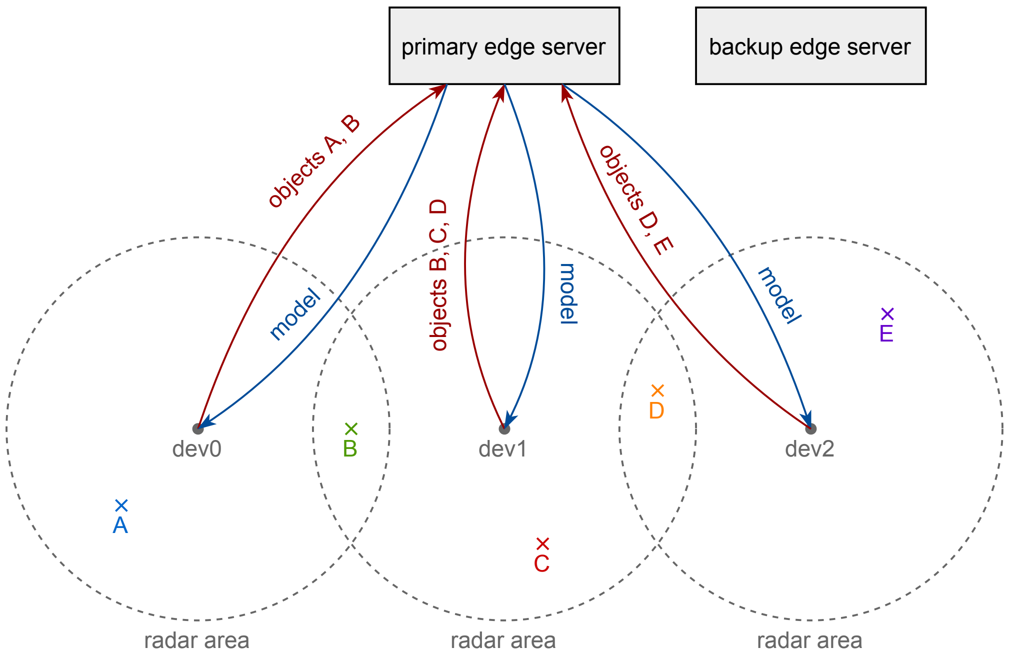


Figure 7.1: Sample deployment of the DMTT application.

7.2 Implementation

The implementation uses supercalls to perform synchronized distributed computations for collecting the targets detected by each device's radar. Devices send the targets to the edge server using a logger. The edge server broadcasts the model asynchronously to the devices.

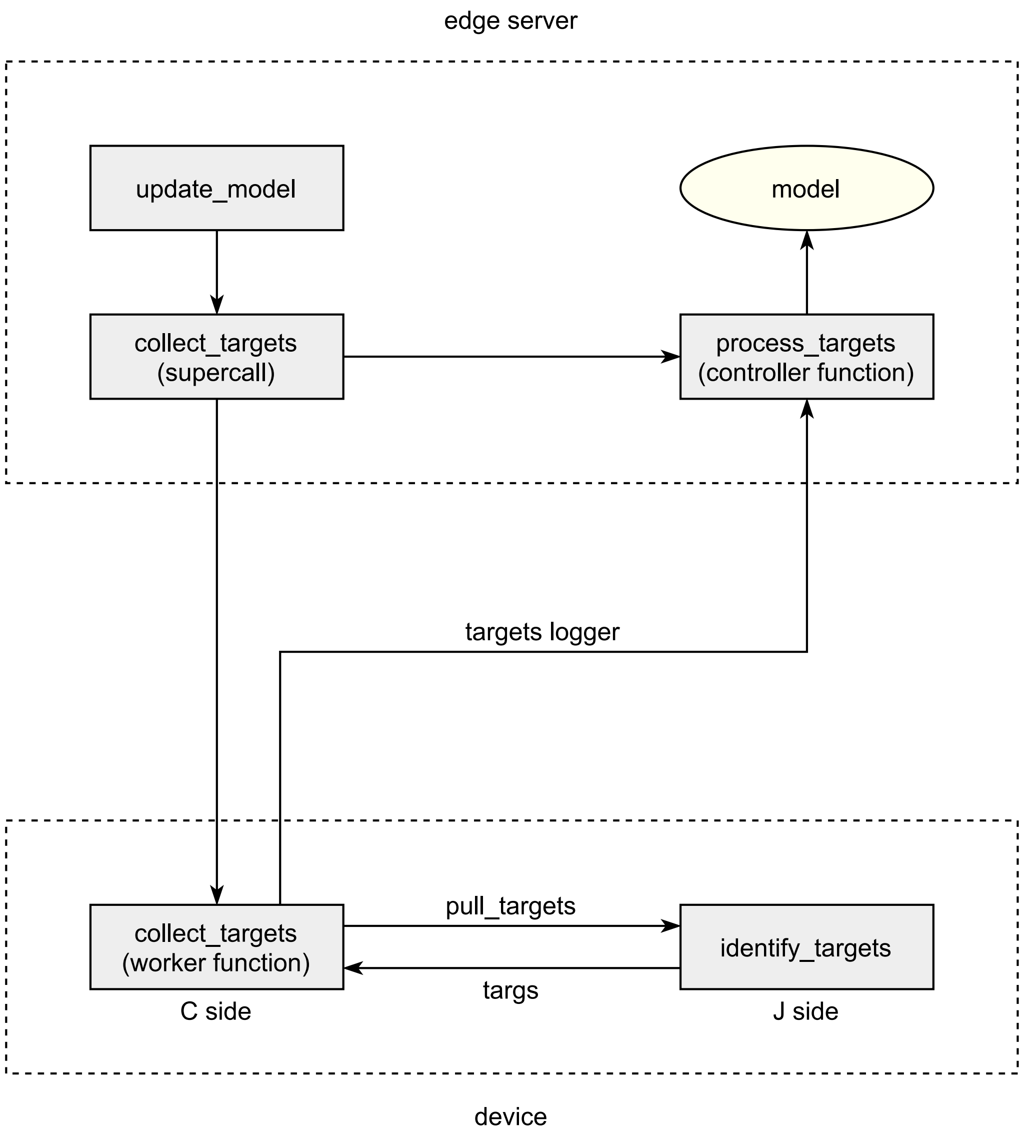


Figure 7.2: Building the model of tracked targets in the DMTT application.

Figure 7.2 illustrates the relationships between the main functions involved in building the model of tracked targets. The `update_model` function is called initially on edge server program start-up and runs periodically. It launches the `collect_targets` supercall with `process_targets` as a controller function. The supercall triggers the execution of the `collect_targets` worker function on the C side of each device. The worker function launches the `pull_targets` C2J sync activity which is called from the C side but executes on the J side. The `pull_targets` function calls `identify_targets` which runs on the J side of the device and matches objects detected by the device's radar to targets in the model. The `pull_targets` function returns the retrieved targets to the worker function on the C side of the device which writes them to the `targets` logger. The `process_targets` controller function executing on the edge server collects targets from all devices and rebuilds the model.

Listing 7.1 shows a fragment of the edge server program responsible for updating the model. A supercall is launched by calling the `collect_targets` function on line 4. The supercall invocation triggers the execution of a function with the same name (worker function) at the devices. The last argument of `collect_targets` is the controller function `process_targets` defined in Listing 7.4. It executes on the edge server and runs concurrently with the `collect_targets` worker function. On line 6, the program handles the two possible outcomes of the supercall (success or failure) by calling the `then` method of the returned promise. In both cases, another supercall is launched after a delay of 500 ms to refresh the model (lines 10 and 15).

Listing 7.1: Model update at the edge server.

```
1 // initially called on edge server program start-up
2 function update_model() {
3     // launch a supercall
4     var c = collect_targets(jsys.id, process_targets);
5     // handle the two possible outcomes
6     c.then(
7         res => {
```

```
            // res is an array of device results
            console.log(res);
            setTimeout(update_model , 500);
        },
        err => {
            // err is an error message
            console.log(err);
            setTimeout(update_model , 500);
        }
    );
}
```

Listing 7.2 shows the C side code of the device program which contains the definition of the worker function of the supercall. All devices participating in the supercall start executing it at the same time. The call to the `pull_targets` function on line 4 triggers the generation of up-to-date tracking information at each device. The `pull_targets` function is a synchronous C2J activity that executes on the J side of the device and calls the `identify_targets` function defined in Listing 7.3. The newly generated tracking information is written to the `targets` logger on line 6. The JAMScript runtime automatically propagates the logger data to the edge server.

Listing 7.2: Target collection at the devices.

```
1 // runs when the supercall is launched at the edge server
2 jsync_ctx int collect_targets(char* id) {
3     // generate up-to-date tracking information
4     char* targs = pull_targets();
5     // write to the targets logger defined on the J side
6     targets = targs;
7     free(targs);
8     return 0;
9 }
```

Listing 7.3 shows the J side code of the device program which performs tracking and identification. The `identify_targets` function runs periodically as well as on demand when called by the `pull_targets` function (line 4 of Listing 7.2). On line 7, the device obtains the list of objects detected by its radar. It then tries to match each detected object to a target in the local model (line 8). The `identify_target` function called on line 11 examines the coordinates and velocity of each target in the model looking for a match. If the condition of the `if` statement on line 12 is true, the object has been matched to a target in the model. The coordinates of that target are updated (line 14) and its velocity is recalculated based on its current and previous location (lines 16–18). Finally the local model is updated on line 22. In the list of targets returned on line 23, each target has an id (or `'UNKNOWN'`), coordinates, and velocity (optional).

Listing 7.3: Target identification at the devices.

```
1  // match objects detected by device radar to targets in model
2  // runs periodically as well as on demand during supercall
3  function identify_targets() {
4      // list of targets tracked by device
5      var targets = [];
6      // objects detected by device radar
7      var radar_data = get_radar_data();
8      radar_data.forEach(object => {
9          // match object to target in model
10         // based on location and velocity
11         var target = identify_target(object.coords);
12         if (target.id !== 'UNKNOWN') {
13             // update target coordinates
14             target.coords = object.coords;
15             // calculate velocity
16             var dx = object.coords.x - target.coords.x;
17             var dy = object.coords.y - target.coords.y;
18             target.velocity = {x: dx, y: dy};
19         }
20         targets.push(target);
21     });
22     update_local_model(targets);
23     return targets;
24 }
```

Listing 7.4 shows the implementation of the `process_targets` controller function which collects targets from the devices and rebuilds the model. First, the controller function waits 500 ms to receive data from the devices (line 10). It then retrieves the shared stream associated with the `targets` logger and the supercall in progress and stores it in the `shared_stream` variable (line 12). The value for the `context` parameter of the `getSharedStream` method is automatically provided by the JAMScript runtime. The `device_streams` variable on line 13 is a collection of the `targets` logger's data streams from all devices participating in the supercall. The `model` variable on line 14 contains the model being built by the controller function.

The `forEach` function call on line 17 iterates over all device streams and the `device_stream` arrow function parameter represents one data stream from one device. The last value in the `device_stream` represents the targets collected by the device during the current supercall. These targets are retrieved and stored in the `device_targets` variable on line 19. The `forEach` function call on line 21 iterates over all device targets and adds each one to the model. The collection of targets stored in the `model` variable may contain multiple instances of the same target tracked by different devices. The call to the `filter_targets` function on line 26 removes the duplicate instances from the model. Lines 29–33 assign identifiers to targets which were previously unknown to the system and appear for the first time in the model.

The call to the `push_model` function on line 36 broadcasts the newly built model to the devices using an asynchronous othercall. An othercall (`visualize_model`) is also used for visualizing the model (line 39). JAMScript's othercalls are suitable for broadcasting the model to the devices as well as for visualizing it because these operations do not require the fault tolerance and synchronization associated with supercalls. Each device periodically updates its local copy of the model based on its radar data and always has an up-to-date view of the area covered by its radar. The model received from the edge server provides additional information about objects in areas covered by other devices. This additional information is not critical for the operation of the device and it is acceptable for the device to occasionally not receive it. Synchronization is also not required when applying the model from the edge server at each device. Devices can update their local copies of the model at slightly different times. The visualization of the model does not affect the functioning of the system.

Finally, the newly built model returned on line 43 is stored in the shared data stream associated with the `targets` logger and the supercall in progress by calling the `log` method on line 13. This saves the model to the persistent data store. The shared data stream containing all models is automatically replicated to the data store of the backup edge server by the underlying data layer. This functionality is essential for a successful recovery in case of a primary edge server failure.

Listing 7.4: Building the model from the targets collected by the devices.

```
 1 jdata {
 2     // logger used by devices to send targets to the edge server
 3     char* targets as logger;
 4 }
 5 
 6 // supercall controller function (edge server computation)
 7 // collect targets from devices and rebuild model
 8 function process_targets(context) {
 9     // wait for edge server to receive logger data from devices
10     jsys.sleep(500);
11 
12     var shared_stream = targets.getSharedStream(context);
13     shared_stream.log(device_streams => {
14         var model = [];
15 
16         // collect targets from all devices
17         device_streams.forEach(device_stream => {
18             // collect targets from one device
19             var device_targets = device_stream.lastValue();
20             // add device targets to model
21             device_targets.forEach(target => model.push(target));
22         });
23 
```

```
        // remove multiple instances of the same target
        // tracked by different devices
        model = filter_targets(model);

        // assign ids to unknown targets
        model.forEach(target => {
            if (target.id === 'UNKNOWN') {
                target.id = make_id();
            }
        });

        // broadcast model to devices (asynchronous othercall)
        push_model(model);

        // visualize model (asynchronous othercall)
        visualize_model(model);

        // save model to persistent data store
        // by writing it to the shared stream
        return model;
    });
}
```

7.3 Simulation with Flight Data

The multiple-target tracking application implemented with JAMScript supercalls runs in a distributed deployment of multiple edge servers and tracking stations (devices). One possible configuration for testing the application is a network of Raspberry Pi base stations emulating a distributed drone tracking system. Here we present results from testing the application using a flight simulation. We create a distributed system which consists of processes running on the same machine. Each process is either an edge server or device process.

In the simulation, there are three devices deployed at the same latitude but different longitudes 200 km apart. Each device covers a circular area with a radius of 125 km. The areas covered by the first and second device as well as the second and third device overlap.

In the simulation, one device is deployed at an airport, another device is located 200 km to the west and the third device – 200 km to the east.

In place of radar data, the simulation uses real flight data provided by flightradar24.com. The flight data is in a comma-separated values (CSV) format. Each record is comprised of the following fields: Timestamp, UTC, Callsign, Position, Altitude, Speed, Direction. The distributed multiple-target tracking application uses the Timestamp, Position and Altitude fields. The Position field contains latitude and longitude coordinates separated by a comma.

The flight data used in the experiments covers departures and arrivals at the airport in Frankfurt, Germany in December 2020. The departures include flights to London (LH900), Prague (LH1392) and Zurich (LH1186). The arrivals include flights from Paris (AF1018), Munich (LH93) and Dresden (LH215). Typically flight data is reported at time intervals between 20 and 90 seconds.

The visualization component of the application displays the trajectories of all tracked flights as they progress over time. A model is presented as a set of points where each point corresponds to a plane location at a particular point in time. Each model is a snapshot in time of the positions of all tracked flights. The visualization is comprised of all models (snapshots) generated since application start-up.

Figure 7.3 presents the trajectories of all tracked flights up to a particular point in time. The blue, brown, and red trajectories correspond to flights departing from the Frankfurt airport. The orange, green, and purple trajectories represent arriving flights. Device 0 is detecting two planes (LH900 and AF1018). Device 1 is detecting three planes (LH1186, LH215, and LH93). Device 2 is detecting two planes (LH93 and LH1392). The plane for flight LH93 is being tracked by both Device 1 and Device 2 at this particular moment due to the overlapping areas monitored by the devices.

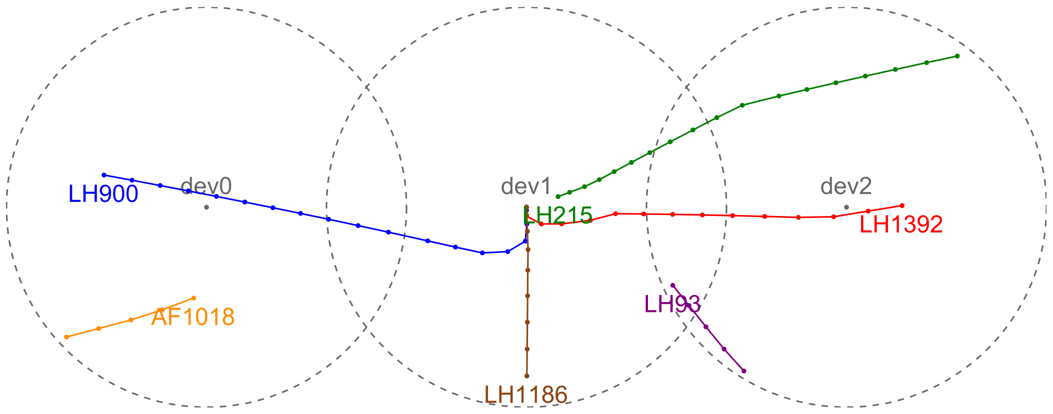


Figure 7.3: Flight tracking simulation.

7.4 Experimental Results

We have measured the performance of the JAMScript supercall implementation in various scenarios with and without edge server failures. Device failures are not simulated because they cause the supercall to complete with an error message and do not trigger a re-execution. To discount application-specific overhead, the performance test application involves an edge server computation which takes 500 ms and a no-op device computation. We run the tests on a laptop with 32 GB of RAM and an Intel Core i7-7820HQ CPU running Ubuntu Linux 18.04. All edge servers share the same persistent data store (Redis 4.0.9).

The recovery time in case of edge server failure during a supercall is defined as the time elapsed between the moment the primary edge server fails and the moment the backup edge server attempts re-execution of the failed supercall. In our tests, the recovery time averaged 25 ms with standard deviation 1.9 ms over 30 test runs. The short recovery time shows the ability of the JAMScript runtime to recover quickly from edge server failures with minimal disruption to the system.

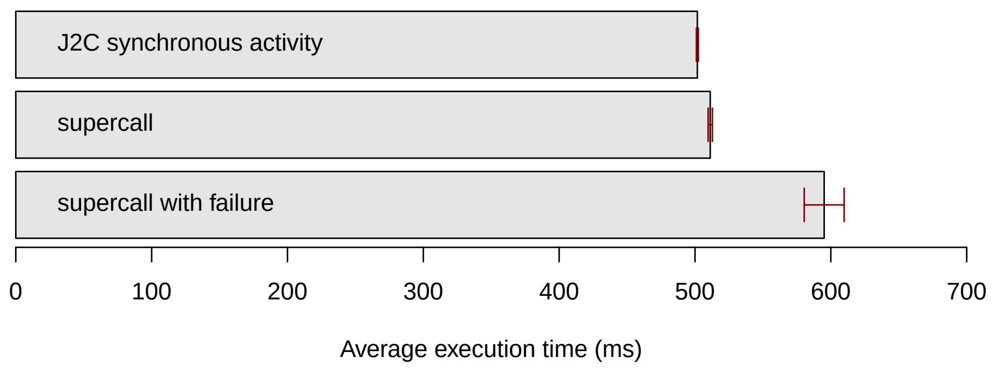


Figure 7.4: Comparison of execution times of supercalls and J2C sync activities.

The supercall implementation in JAMScript is based on the implementation of the J2C sync activity. Compared to a J2C sync activity, a supercall provides the ability to run simultaneously computations at the edge server and devices. A supercall also implements a fault tolerance mechanism for recovering from edge server failures with automatic re-execution. The additional functionality of supercalls is provided with a small performance overhead. The average execution time of a supercall over 30 test runs is 511.2 ms with standard deviation 1.6 ms (middle bar of Figure 7.4). A supercall is only 1.9% slower than a J2C sync activity which has an average execution time of 501.7 ms with standard deviation 0.7 ms over 30 test runs (top bar of Figure 7.4).

We also compare the supercall execution time without failures and the supercall execution time when there is a primary edge server failure and re-execution after recovery. The failure is triggered programmatically at the start of the controller function. The average execution time without failure is 511.2 ms with standard deviation 1.6 ms over 30 test runs (middle bar of Figure 7.4). 500 ms of the execution time are spent in the edge server computation (application dependent). The JAMScript runtime processing takes only about 10 ms.

The average execution time when there is a failure is 595.1 ms with standard deviation 14.7 ms over 30 test runs (bottom bar of Figure 7.4). The execution time with failure consists of the following components: 35 ms runtime processing during the initial execution;

25 ms recovery time; 500 ms edge server computation during the re-execution (application dependent); 35 ms runtime processing during the re-execution. The time to recover from failure during a supercall and successfully re-execute the supercall is 16% larger than the execution time without failure.

Figure 7.5 shows the execution times of the first 30 supercalls during one run of the distributed multiple-target tracking application with programmatically simulated failures during supercalls 10 and 20. The performance results are similar to the ones obtained with the performance test application.

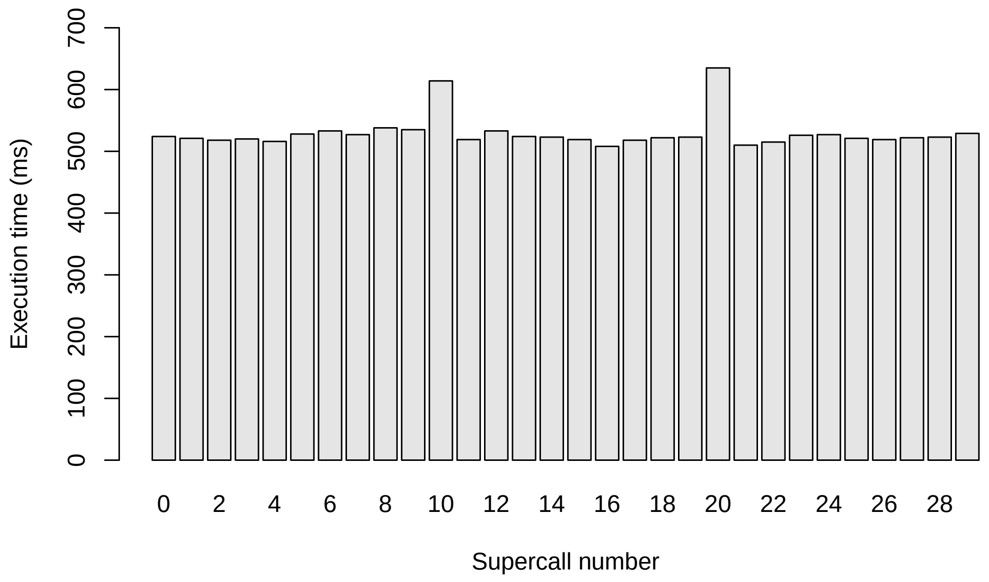


Figure 7.5: Variation of supercall execution times in the DMTT application.

Chapter 8

Related Work

8.1 Apache Hama

Apache Hama [46] is a framework for distributed parallel computing which implements the bulk synchronous parallel (BSP) programming model. It uses the Hadoop Distributed File System (HDFS). In Hama, a program is a sequence of supersteps with barrier synchronization between them. Each superstep consists of three phases: local computation using local data values, global communication for inter-process exchange of locally computed data, and barrier synchronization to ensure that all actions in the superstep complete.

Hama's architecture follows the master-slave model and consists of three components: BSP master, Groom server (slave component), and synchronization component. The BSP master schedules jobs and assigns tasks to the Groom servers. Each Groom server runs one or more BSP tasks assigned by the BSP master. The synchronization component (ZooKeeper) executes on the same node as the BSP master and provides efficient barrier synchronization for the BSP tasks.

Hama is a versatile framework suitable for use in different application domains like graph processing, streaming, and machine learning. Many frameworks for processing big data are designed for handling data-intensive tasks. Hama instead focuses on computation-intensive tasks.

One of the advantages of Hama is support for inter-process communication. Unlike Hadoop where direct communication between Map and Reduce tasks is not allowed, Hama allows BSP tasks to directly exchange messages. This avoids the I/O cost of using the distributed file system for indirect communication.

In addition to HDFS, Hama can integrate with any distributed file system. It also provides support for general-purpose computing on graphics processing units (GPGPU). One of the main limitations of Hama is the BSP master being a single point of failure.

Comparative Analysis

JAMScript's supercalls are similar to Apache Hama's supersteps from the BSP programming model. Both perform computations running in parallel on a set of workers. In Hama's master-slave model, computations execute on Groom servers coordinated by a BSP master. In JAMScript's controller-worker architecture, computations run on both the primary edge server (controller) and the device nodes (workers) connected to it. Unlike Apache Hama's BSP tasks which can exchange messages, in JAMScript the device nodes (workers) can communicate only with the primary edge server and not with each other.

In Apache Hama, the synchronization is managed by a dedicated synchronization component. In JAMScript, the primary edge server performs the synchronization functions. In Apache Hama's BSP model, barrier synchronization is performed at the end of each superstep. When a process reaches the barrier, it waits until all other processes have reached it. Similarly, a JAMScript supercall ends only when both the primary edge server (controller) and all device nodes (workers) have finished their computations. Similarly to a synchronization barrier, upon reaching a supercall during program execution, non-primary edge servers wait until the primary edge server executes the supercall and returns a result.

8.2 Client-Edge-Server for Stateful Network Applications

Many cloud computing applications perform computations at the edge instead of the cloud or client. Doing computations at the edge instead of the cloud decreases the response latency. Performing computations at the edge instead of the client lessens the computational resource requirements on the client.

The Client-Edge-Server for Stateful Network Applications (CESSNA) [47] framework and runtime environment aims to solve the challenges of dealing with edge failures and client mobility while maintaining state at the edge. The edge keeps state for each client-server session. The session state consists of messages and data exchanged between client and edge and between edge and server. The framework provides strong state consistency guarantees in case of edge failures. After a failure, the new edge responds to the client/server with the same messages that would have been returned by the original edge if it had not failed.

To integrate with CESSNA, the edge application code must use a special API to communicate with the client/server and perform non-deterministic operations. The edge framework implementing the API maintains a log which records the order in which messages are processed as well as information about non-deterministic events. To recover the session state at another edge, the runtime replays the log.

The log could get very large and slow down recovery. To reduce the log size, snapshots capturing the state of the edge application are created at regular time intervals. CESSNA provides two implementations for generating snapshots. Container isolation uses Docker checkpointing with a Python API to perform checkpoints. Software isolation uses custom checkpointing with a Rust API to perform checkpoints and read/write state.

The framework provides two recovery modes: local and remote. In the local recovery mode, snapshots are stored locally and the failed and new edge runtimes share fast storage. In the remote recovery mode, snapshots are stored locally as well as on another node (e.g., client, server, another edge) from which the new edge fetches the snapshot. A limitation of the current design is the lack of support for handling multiple clients per session in an edge application.

Comparative Analysis

Both CESSNA and JAMScript aim to solve the challenges of developing stateful edge applications which tolerate edge failures and client mobility. In CESSNA, the edge state is associated with a single client-server session. The framework provides strong state consistency guarantees in case of edge server failures. In JAMScript, the edge state can be associated with either a single device (private data) or multiple devices (shared data). Shared data is constructed at the edge during a supercall, while private data can also be constructed outside of a supercall. Data generated inside a supercall is regenerated in case of edge server failures. The regenerated data might differ from the data that would have been generated without failures but it reflects changes in the physical world on which it is based.

CESSNA and JAMScript manage fault tolerance at a different granularity level. CESSNA aims to make edge failures transparent to the client, while JAMScript involves the devices in the recovery process. To achieve fault tolerance, CESSNA performs logging and checkpointing. The log records information about messages exchanged between client, edge and cloud as well as non-deterministic events. In case of edge server failure, the log is replayed at the new edge server. To reduce the log size, checkpoints capturing the state of the edge application are created at regular time intervals. In JAMScript, fault tolerance is achieved at the supercall level. The application developer must put critical code inside a supercall. If the primary edge server fails during a supercall, the backup edge server takes over as primary and re-executes the supercall. Both the new primary edge server and devices re-execute their computations.

8.3 Resilience of Stateful IoT Applications

Designing fault-tolerant systems for edge computing applications is challenging due to the dynamic nature of the edge environment, the heterogeneity of the computing entities, and the interaction with the physical world. Computing entities may join or leave the system at any time due to mobility or failure. Applications run on a wide range of hardware with different processing and storage capabilities. Devices often interact with the physical world and in many cases must perform actions at specific times and locations.

The authors of [48] propose a fault-tolerant system for stateful edge applications which considers dynamicity, heterogeneity, and interactions with the physical world. To assist in failure recovery, the state of the application is saved at specific execution points. The system monitors both the network and computing entities for failures. When a failure occurs, the system is reconfigured and the saved application state is used during recovery. The recovery process takes into account the validity of actions in the physical world. Actions that are no longer consistent with the physical world are ignored.

The failure management protocol involves a number of global and distributed entities. The main global entities are the Global Manager and Application Lifecycle Manager. The Global Manager has a bird's-eye view of the network and computing entities. It tracks failed entities and manages the recovery process. The Application Lifecycle Manager is responsible for deploying application components. The main distributed entities are Fog Agents and Software Element Loggers. Fog Agents monitor the system for failures. Software Element Loggers are used for saving the state of the application.

The failure management protocol implements three strategies for saving application state: checkpointing, message log, and function call record. The strategy used by a computing entity (edge server, device) depends on its properties (local storage, communication model). Strategies can also be combined. For message log and function call record, Software Element

Loggers intercept messages and function calls and log them before passing them to the application. When a new checkpoint is created successfully, the previous state (checkpoints, message logs, function call records) for that computing entity is deleted. Each event occurring in the physical world is annotated with a validity timer. The timer indicates the period of time during which the event is consistent with the physical world.

Fog Agents are deployed at the edge to monitor both the network and computing entities. They send status updates to the Global Manager. Each agent has a local backup agent that replaces it in case of failure. Edge servers are monitored using a heartbeat mechanism. To monitor devices, agents can either observe application messages sent by the devices or send them ping requests. The former is suitable for devices that communicate regularly with the edge. It is the preferred method as it does not affect the device or network.

The actions to recover from a failure depend on the type of the entity, the associated state saving strategy, and whether the entity interacts with the physical world. The appropriate actions are determined by the Global Manager. The failed entity is replaced with another one (if possible) by the Application Lifecycle Manager. If a checkpoint exists, it is used to initialize the state of the entity. The message log and function call record are used to replay messages and function calls recorded after the last checkpoint. Events with expired validity timers are not replayed because they are no longer consistent with the physical world.

Comparative Analysis

The fault tolerance mechanism proposed by the authors handles both edge server and device failures, while the one in JAMScript is designed only for the edge. The two systems also differ in the granularity of fault tolerance. The proposed approach is fine-grained. It records function invocations and messages exchanged between nodes. In contrast, JAMScript's fault tolerance mechanism is coarse-grained. It records supercalls and their results. By selectively placing code inside supercalls, the cost of providing application reliability is minimized.

8.4 EdgeCons

In modern edge computing applications, large numbers of devices are connected to the edge. The devices generate events at high rates which need to be ordered. Ordering the events at the cloud introduces additional latency which affects the user experience. The events can instead be ordered at the edge. One possible approach involves running a consensus protocol among the edge servers.

The EdgeCons [49] consensus protocol aims to achieve fast event ordering for delay-sensitive applications deployed on the edge. It employs a fixed set of edge servers and runs a sequence of Paxos instances among them. The leadership of the Paxos instances changes dynamically based on the history of previous runs of the consensus protocol. Unlike Paxos which does not guarantee making progress in case of failures, EdgeCons guarantees making progress by periodically using the cloud. It is assumed that the cloud never fails and the edge is always connected to it.

The consensus process in EdgeCons is organized into epochs. Each epoch consists of a fixed number of Paxos instances. The first phase (leader election) of the Paxos algorithm is skipped because the leader for each instance is known in advance. This speeds up reaching consensus as there are fewer communication rounds. The assignment of leaders for the next epoch is a result of a deterministic process performed independently by each edge server at the end of the current epoch.

The consensus protocol assigns leadership based on the history of previous runs. A Paxos instance is effective if and only if a majority of edge servers agree on the proposed value. The system tracks the number of effective instances for each edge server. The number of instances assigned to an edge server is proportional to the number of effective instances recorded for that server. The assigned Paxos instances are evenly spaced within the epoch.

In order to ensure progress, EdgeCons periodically involves the cloud which is assumed to be always running and reachable. In addition to Paxos instances, each epoch has quasi-Paxos instances at predetermined points which give control to the cloud for event ordering. If an edge server is unable to get its proposed value accepted by the system after several attempts, it sends the value to the cloud. The values are ordered at the cloud by time of arrival. During a quasi-Paxos instance, the cloud sends to the edge servers the ordered list of values collected since the last quasi-Paxos instance. The edge servers are required to accept these values.

The authors of EdgeCons discuss two limitations of the current design. The first limitation is related to the leadership assignment algorithm which gives more shares to edge servers with a large number of effective Paxos instances in the recent past. If an edge server has network problems for a period of time and then recovers, it may take a while to receive the same proportion of shares as before the failure. The second limitation is the fixed number of quasi-Paxos instances in each epoch. If there is a spike in the edge workload, a larger number of quasi-Paxos instances might be needed temporarily to make fast progress.

Comparative Analysis

EdgeCons runs a Paxos-based consensus protocol among the edge servers to achieve fast ordering of edge events. JAMScript uses a supercall log to record supercall lifecycle events, execution results, and changes in the selection of edge servers in the zone. The supercall log is accessed by all edge nodes. A distributed implementation of the supercall log requires running a consensus protocol like Paxos among the edge servers. This adds communication overhead but does not require a permanent connection from the edge to the cloud. Instead, JAMScript implements a centralized supercall log assuming that the cloud is always reachable from the edge. There are no concurrent updates to the log because only the primary edge server in the zone writes to it. All other edge servers only read from it.

8.5 Griffin

In edge computing, large amounts of data are generated at devices connected to edge servers which collect and process the data. The edge servers run distributed collaborative applications like autonomous driving, augmented reality gaming, machine learning, environmental sensing, and many others. A typical requirement for such applications is the generation of results in real time with minimal delays. This requirement combined with the dynamic nature and heterogeneity of the edge environment present unique implementation challenges.

A key feature of all collaborative applications at the edge is the efficient sharing of state among the edge clients. Typically multiple clients contribute to and process some shared data in order to achieve a common goal. Data sharing can be implemented in two ways: with an external shared storage service or using an application framework. The first approach is preferred because it provides the ability to use well-known RESTful design patterns. Its main advantage is decoupling computations from state management. It is easier to implement fault tolerance in a stateless application framework.

The authors of [50] identify three main properties of the edge computing environment which affect the requirements for a shared storage service: distribution, heterogeneity, and dynamicity. Edge servers run at different geographic locations without a centralized server for hosting the storage service. Each edge server might have different storage capacity and network bandwidth. The edge environment can often change due to mobility or failures.

The distributed collaborative applications running at the edge have different requirements with regards to data types, consistency, and performance. In an edge machine learning application, the shared state consists of the model parameters updated by multiple workers. In a real-time massively multiplayer online game, players interact with each other in a shared virtual world. In this case, the implementation must ensure low latency and bandwidth use for the best gaming experience. In an autonomous driving application, data sharing between cars can help predict traffic and avoid road obstacles.

The authors formulate several key requirements for a shared storage system. The abstractions and APIs requirement focuses on the supported data types (text, images, and videos) and data access semantics with the key-value interface being the most popular. The data locality requirement deals with reducing latency and efficient use of network bandwidth. Other requirements address the need to support edge servers with different hardware as well devices like cars and phones which are expected to change their location frequently. An important requirement is the ability to recover from node failures and operate with limited network connectivity. The shared storage service must also scale to a large number of edge servers, clients, and stored objects. A robust monitoring infrastructure must provide capabilities for detecting failures and node mobility.

The authors present a design for Griffin, a shared storage service for the edge, which satisfies the above requirements. Griffin is a hierarchical distributed storage service with multiple layers. A data storage daemon runs on every edge node. Resource allocation is centralized on the cloud where decisions about data creation and storage location are made. The actual read and write operations are performed by the client. Clients bootstrap using a service address in the cloud known in advance. Each data item is tagged with a space-time label to support data locality. Griffin utilizes data replicas for dealing with failures.

Griffin provides support for multiple consistency models. The consistency model is not determined during application development. Developers specify declaratively expected latencies and desired consistency models. At runtime the system picks the optimal model for every data access. The main challenge is predicting latencies and loads in the edge environment. Griffin uses graph-based optimization to make decisions about data placement and replication. Griffin saves the status of the system in a graph. The vertices represent edge sites and are annotated with resource utilization. The links represent connections between edge sites and are annotated with network latency and bandwidth. A monitoring system collects system statistics used by the optimization engine to build the graph.

Comparative Analysis

Griffin is a shared storage service for the edge with a hierarchical distributed architecture. A data storage daemon runs on every edge node. Decisions about where the data is created and stored are made on the cloud using graph-based optimization which takes into account resource utilization, network latency and bandwidth. In JAMScript, each device and edge node has a dedicated data store. The cloud is not involved in data storage decisions about other nodes. Data generated at the devices is automatically propagated by the JAMScript runtime to the primary edge server in the zone as well as to the cloud. Similarly, data generated at any edge server is propagated to the cloud.

Griffin deals with failures by placing data replicas at strategic locations. In JAMScript, data generated at device nodes and propagated by the runtime to the primary edge server during a supercall is replicated by the underlying data layer to the backup edge server. To ensure supercall recovery in case of primary or backup edge server failure, two copies of the data generated at the primary edge server during the current and previous supercalls are always kept.

JAMScript distinguishes between private and shared data. Private data is generated at the primary edge server based only on data from one device. Private data is not associated with a supercall. Shared data is generated at the primary edge server based only on data from devices participating in a supercall. The supercall ensures that the device data arriving at the edge server is time-aligned.

Chapter 9

Conclusions and Future Work

The Internet of Things (IoT) has experienced explosive growth in recent years. In the Cloud of Things (CoT) computing model, cloud computing systems provide the data processing capabilities which the IoT devices lack. Edge computing brings powerful computing resources closer to the devices to reduce the latency of access. Specialized software frameworks have been developed to address the challenges of edge environments such as device mobility, disconnections, and network latency variability. The JAMScript language and middleware offer a unique platform for developing edge-oriented IoT applications. JAMScript implements efficient inter-node communication through synchronous and asynchronous activities, automatic data propagation with loggers and broadcasters, and the ability to execute a distributed computation on many devices.

In this thesis, we introduced application zones for grouping together computing nodes in physical proximity executing the same JAMScript program. We explained how the partitioning of the physical space into zones enables performing synchronized distributed computations, generating shared data at the edge, and dealing with edge server failures. We discussed the roles of the primary and backup edge servers in a zone in recovering from edge server failures. We presented the design and implementation of supercalls which enhance the functionality of synchronous controller-to-worker activities in JAMScript. We demonstrated the use of supercall programming for performing collective distributed computations in a zone and explained how the automatic re-execution after failure works. We introduced private and shared data and compared their reliability guarantees. We incorporated application zones, supercalls, and private and shared data in a comprehensive fault tolerance scheme for recovering from edge server failures. We discussed node and network failures and system recovery in different scenarios. As a proof of concept, we built an application for distributed multiple-target tracking. We reviewed the implementation and analyzed the experimental results from a simulation with flight data. The experiments showed that the JAMScript runtime recovers quickly from edge server failures and the overhead of the implemented fault tolerance scheme is very small.

As part of future work, the JAMScript runtime will employ network-aware scheduling to optimize the supercall execution order. In the current implementation, the order in which supercalls run is determined at compile time. Supercalls are executed in the order that they appear in the program and according to the rules for chaining JavaScript promises. Currently, a supercall computation starts only if a quorum of devices confirms participation. Additional types of preconditions for supercall execution could be added. For example, a supercall could run only if a subset of devices from another supercall confirms participation. A supercall might also require the network latency to be below a specified threshold. The JAMScript runtime will use the supercall dependency graph and preconditions to make scheduling decisions. When independent supercalls are submitted to the runtime, the network-aware scheduler will examine the preconditions of each one along with the network conditions and try to execute first the supercall that is most likely to succeed.

Another area of future work is addressing the challenges of device mobility between application zones. In the current implementation, the zone to which a device belongs is determined on device start-up. The runtime will be enhanced to detect when a device crosses zone boundaries. When a device enters a new zone, it will connect to the primary edge server of the new zone. Shared data associated with the old zone will not be available in the new zone. The device will have access to previously generated private data.

Another future research problem is studying the performance of the supercall implementation in a variety of edge computing applications and scenarios. The goal is to improve the performance of the supercall implementation and deploy the JAMScript prototype in latency-sensitive environments such as high-speed vehicular networks, augmented reality, and real-time gaming.

References

[1] M. H. Miraz, M. Ali, P. S. Excell, and R. Picking, “A review on Internet of Things (IoT), Internet of Everything (IoE) and Internet of Nano Things (IoNT),” in *Proceedings of the 6th International Conference on Internet Technologies and Applications (ITA)*, 2015, pp. 219–224. [Online]. Available: https://doi.org/10.1109/ITechA.2015.7317398

[2] Z. N. Aghdam, A. M. Rahmani, and M. Hosseinzadeh, “The role of the Internet of Things in healthcare: Future trends and challenges,” *Computer Methods and Programs in Biomedicine*, vol. 199, p. 105903, 2021. [Online]. Available: https://doi.org/10.1016/j.cmpb.2020.105903

[3] F. Sadoughi, A. Behmanesh, and N. Sayfouri, “Internet of Things in medicine: A systematic mapping study,” *Journal of Biomedical Informatics*, vol. 103, p. 103383, 2020. [Online]. Available: https://doi.org/10.1016/j.jbi.2020.103383

[4] I. B. Aris, R. K. Z. Sahbusdin, and A. F. M. Amin, “Impacts of IoT and big data to automotive industry,” in *Proceedings of the 10th Asian Control Conference (ASCC)*, 2015, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ASCC.2015.7244878

[5] C. Yang, W. Shen, and X. Wang, “Applications of Internet of Things in manufacturing,” in *Proceedings of the 20th IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2016, pp. 670–675. [Online]. Available: https://doi.org/10.1109/CSCWD.2016.7566069

[6] D. Hicks, K. Mannix, H. M. Bowles, and B. J. Gao, “SmartMart: IoT-based in-store mapping for mobile devices,” in *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications*

and Worksharing (CollaborateCom), 2013, pp. 616–621. [Online]. Available: https://doi.org/10.4108/icst.collaboratecom.2013.254116

[7] T. Qiu, N. Chen, K. Li, M. Atiquzzaman, and W. Zhao, "How can heterogeneous Internet of Things build our future: A survey," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 3, pp. 2011–2027, 2018. [Online]. Available: https://doi.org/10.1109/COMST.2018.2803740

[8] H. N. Saha, S. Auddy, A. Chatterjee, S. Pal, S. Pandey, R. Singh, R. Singh, P. Sharan, S. Banerjee, D. Ghosh, and A. Maity, "Pollution control using Internet of Things (IoT)," in *Proceedings of the 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON)*, 2017, pp. 65–68. [Online]. Available: https://doi.org/10.1109/IEMECON.2017.8079563

[9] M. S. U. Chowdury, T. B. Emran, S. Ghosh, A. Pathak, M. M. Alam, N. Absar, K. Andersson, and M. S. Hossain, "IoT based real-time river water quality monitoring system," *Procedia Computer Science*, vol. 155, pp. 161–168, 2019. [Online]. Available: https://doi.org/10.1016/j.procs.2019.08.025

[10] L. Özgür, V. K. Akram, M. Challenger, and O. Dağdeviren, "An IoT based smart thermostat," in *Proceedings of the 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, 2018, pp. 252–256. [Online]. Available: https://doi.org/10.1109/ICEEE2.2018.8391341

[11] K. Sehgal and R. Singh, "IoT based smart wireless home security systems," in *Proceedings of the 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2019, pp. 323–326. [Online]. Available: https://doi.org/10.1109/ICECA.2019.8821885

[12] R. K. Kodali, V. Jain, S. Bose, and L. Boppana, "IoT based smart security and home automation system," in *Proceedings of the 2nd International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 1286–1289. [Online]. Available: https://doi.org/10.1109/CCAA.2016.7813916

[13] M. Martínez-Díaz and F. Soriguera, “Autonomous vehicles: Theoretical and practical challenges,” *Transportation Research Procedia*, vol. 33, pp. 275–282, 2018. [Online]. Available: https://doi.org/10.1016/j.trpro.2018.10.103

[14] M. Aazam, I. Khan, A. A. Alsaffar, and E.-N. Huh, “Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved,” in *Proceedings of the 11th International Bhurban Conference on Applied Sciences Technology (IBCAST)*, 2014, pp. 414–419. [Online]. Available: https://doi.org/10.1109/IBCAST.2014.6778179

[15] S. Patidar, D. Rane, and P. Jain, “A survey paper on cloud computing,” in *Proceedings of the 2nd International Conference on Advanced Computing and Communication Technologies (ACCT)*, 2012, pp. 394–398. [Online]. Available: https://doi.org/10.1109/ACCT.2012.15

[16] S. Khare and M. Totaro, “Big data in IoT,” in *Proceedings of the 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, pp. 1–7. [Online]. Available: https://doi.org/10.1109/ICCCNT45670.2019.8944495

[17] R. Birke, L. Y. Chen, and E. Smirni, “Data centers in the cloud: A large scale performance study,” in *Proceedings of the 5th IEEE International Conference on Cloud Computing (IEEE Cloud)*, 2012, pp. 336–343. [Online]. Available: https://doi.org/10.1109/CLOUD.2012.87

[18] M. Caballer, C. de Alfonso, F. Alvarruiz, and G. Moltó, “EC3: Elastic Cloud Computing Cluster,” *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1341–1351, 2013. [Online]. Available: https://doi.org/10.1016/j.jcss.2013.06.005

[19] D. A. Popescu, N. Zilberman, and A. W. Moore, “Characterizing the impact of network latency on cloud-based applications’ performance,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-914, 2017. [Online]. Available: https://doi.org/10.17863/CAM.17588

[20] S. Yi, Z. Hao, Z. Qin, and Q. Li, “Fog computing: Platform and applications,” in *Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies*

(HotWeb), 2015, pp. 73–78. [Online]. Available: https://doi.org/10.1109/HotWeb.2015.22

[21] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2017.2778504

[22] W. Ahmed and Y. W. Wu, "A survey on reliability in distributed systems," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1243–1255, 2013. [Online]. Available: https://doi.org/10.1016/j.jcss.2013.02.006

[23] J. Grover and R. M. Garimella, "Reliable and fault-tolerant IoT-edge architecture," in *Proceedings of the 17th IEEE Conference on Sensors (IEEE Sensors)*, 2018, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ICSENS.2018.8589624

[24] 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, no. 1, pp. 57–80, 2020. [Online]. Available: https://doi.org/10.1007/s10723-019-09498-8

[25] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Towards fault tolerant fog computing for IoT-based smart city applications," in *Proceedings of the 9th IEEE Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0752–0757. [Online]. Available: https://doi.org/10.1109/CCWC.2019.8666447

[26] R. Wenger, X. Zhu, J. Krishnamurthy, and M. Maheswaran, "A programming language and system for heterogeneous Cloud of Things," in *Proceedings of the 2nd IEEE International Conference on Collaboration and Internet Computing (CIC)*, 2016, pp. 169–177. [Online]. Available: https://doi.org/10.1109/CIC.2016.033

[27] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[28] D. Flanagan, *JavaScript: The Definitive Guide*, 6th ed. O'Reilly Media, Inc., 2011.

[29] M. Barr, *Programming Embedded Systems in C and C++*, 1st ed. O'Reilly & Associates, Inc., 1998.

[30] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*, 2nd ed. O'Reilly Media, Inc., 2006.

[31] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000. [Online]. Available: https://doi.org/10.1109/2.876288

[32] M. Cantelon, M. Harter, T. J. Holowaychuk, and N. Rajlich, *Node.js in Action*, 1st ed. Manning Publications Co., 2013.

[33] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js," in *Proceedings of the 17th IEEE International Conference on Computational Science and Engineering (CSE)*, 2014, pp. 661–668. [Online]. Available: https://doi.org/10.1109/CSE.2014.142

[34] L. P. Chitra and R. Satapathy, "Performance comparison and evaluation of Node.js and traditional web server (IIS)," in *Proceedings of the 2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, 2017, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ICAMMAET.2017.8186633

[35] L. Ben Arfa Rabai, B. Cohen, and A. Mili, "Programming language use in US academia and industry," *Informatics in Education*, vol. 14, no. 2, pp. 143–160, 2015. [Online]. Available: https://doi.org/10.15388/infedu.2015.09

[36] W. Grosso, *Java RMI*, 1st ed. O'Reilly & Associates, Inc., 2001.

[37] J. L. Carlson, *Redis in Action*, 1st ed. Manning Publications Co., 2013.

[38] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani, "An open ecosystem for mobile-cloud convergence," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 63–70, 2015. [Online]. Available: https://doi.org/10.1109/MCOM.2015.7060484

[39] R. Olaniyan and M. Maheswaran, "Synchronous scheduling algorithms for edge coordinated Internet of Things," in *Proceedings of the 2nd IEEE International Conference on Fog and Edge Computing (ICFEC)*, 2018, pp. 1–10. [Online]. Available: https://doi.org/10.1109/CFEC.2018.8358725

[40] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017. [Online]. Available: https://doi.org/10.1109/COMST.2017.2745201

[41] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017. [Online]. Available: https://doi.org/10.1109/COMST.2017.2682318

[42] S. M. A. Ataallah, S. M. Nassar, and E. E. Hemayed, "Fault tolerance in cloud computing – survey," in *Proceedings of the 11th International Computer Engineering Conference (ICENCO)*, 2015, pp. 241–245. [Online]. Available: https://doi.org/10.1109/ICENCO.2015.7416355

[43] A. Ganesh, M. Sandhya, and S. Shankar, "A study on fault tolerance methods in cloud computing," in *Proceedings of the 4th IEEE International Advance Computing Conference (IACC)*, 2014, pp. 844–849. [Online]. Available: https://doi.org/10.1109/IAdCC.2014.6779432

[44] A. Javed, A. Malhi, and K. Främling, "Edge computing-based fault-tolerant framework: A case study on vehicular networks," in *Proceedings of the 16th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2020, pp. 1541–1548. [Online]. Available: https://doi.org/10.1109/IWCMC48107.2020.9148269

[45] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[46] K. Siddique, Z. Akhtar, E. J. Yoon, Y.-S. Jeong, D. Dasgupta, and Y. Kim, "Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications," *IEEE Access*, vol. 4, pp. 8879–8887, 2016. [Online]. Available: https://doi.org/10.1109/ACCESS.2016.2631549

[47] Y. Harchol, A. Mushtaq, J. McCauley, A. Panda, and S. Shenker, "CESSNA: Resilient edge-computing," in *Proceedings of the 2nd Workshop on Mobile*

Edge Communications (MECOMM), 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3229556.3229558

[48] U. Ozeer, X. Etchevers, L. Letondeur, F.-G. Ottogalli, G. Salaün, and J.-M. Vincent, "Resilience of stateful IoT applications in a dynamic fog environment," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*, 2018, pp. 332–341. [Online]. Available: https://doi.org/10.1145/3286978.3287007

[49] Z. Hao, S. Yi, and Q. Li, "EdgeCons: Achieving efficient consensus in edge computing networks," in *Proceedings of the 1st USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2018. [Online]. Available: https://www.usenix.org/conference/hotedge18/presentation/hao

[50] A. Trivedi, L. Wang, H. Bal, and A. Iosup, "Sharing and caring of data at the edge," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/trivedi