λ_{Hive}: Formal Semantics of an Edge Computing Model based on JavaScript

Matias Teragni[0000-0002-0994-0619]

Claudia Pons[0000-0003-1149-0976]

Universidad Abierta Interamericana, Argentina

ABSTRACT

Edge computing is a paradigm that applies virtualization technology that makes it easier to deploy and run a wider range of applications on the edge servers and take advantage of largely unused computational resources. This article describes the design and formalization of Hive, a distributed shared memory model that can be transparently integrated with JavaScript using a standard out of the box runtime. To define such model a formal definition of the JavaScript language was used and extended to include modern capabilities and custom semantics. This extended model was used to prove that the distributed shared memory can operate on top of existing and unmodified web browsers. The proposed model guarantees the eventual synchronization of data across all the system and provides the possibility to have a stricter consistency using standard http operations. The technical feasibility of this proposal was empirically validated by a prototype that yields reasonably low propagation time and allows the distribution of preexisting JavaScript code without any major modifications. Additionally, the comprehensive formalization of the Hive execution model allows developers to guarantee certain properties of the synchronization mechanism, such as efficient and no blocking.

Keywords: Distributed Shared Memory, Edge Computing, Cloud Computing, JavaScript, Synchronization mechanism, Formal Semantics.

INTRODUCTION

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user (Foster, Zhao, Raicu, & Lu, 2008). It

enables increased productivity and efficiency (Armbrust, Fox, Griffith, & Joseph, 2010). The cloud paradigm changes various processes, patterns, practices, and philosophies, and its adoption must be carefully analyzed. The authors in (Abdelwahhab & Mostefai, 2020) provide a practical framework for decision-making on cloud paradigm adoption, which is based on cloud standards and best practices. Despite the influence of cost effectiveness, on-demand service, and scalability, cloud computing faces many challenges, such as security, performance, orchestration, and fault tolerance. Several investigations propose technologies and tools to overcome these challenges, such as the models for task scheduling presented in (Alakbarov, 2022) and in (Tuli & Malhotra , 2022) or the model for fault mitigation described in (Adeyinka Osuolale , 2022) and the capability-based access control proposed in (Kaushik & Gandhi, 2020) to ensure that only authorized users will be able to access the data.

Edge computing (Shi, Cao, Zhang, Li, & Xu, 2016) as well as Fog computing (Ahuja & Wheeler , 2020) emerged as natural responses to that conflict. These distributed computing paradigms bring computation and data storage closer to the location where it is needed, by extending the Cloud to the Internet of Things (IoT) devices. In (Shi, Cao, Zhang, Li, & Xu, 2016) the authors not only elaborate on the potential of this distribution paradigm, but also expose the greatest challenges that must be faced to guarantee the viability and popularize this technology.

Modern edge computing (Chelliah & Surianarayanan, 2021) extends this approach through virtualization technologies that make it easier to deploy and run a wider range of applications on the edge servers and take advantage of largely unused computational resources such as the ones present in IoT gadgets. Some examples are, the algorithms for workflow scheduling which consider cost, energy and load balancing in heterogeneous environment described in (Bisht & Vampugani, 2022); the concurrency control protocol for IoT transactions presented in (Al-Qerem, Alauthman, Almomani, & Gupta, 2020); the heuristic algorithms for virtual machine placement and workload assignment defined in (Wang, Tornatore, & Zhao, 2021), and heuristics on the cost-effectiveness of user allocation solutions with the objective of maximizing the number of users allocated to edge servers while minimizing the number of required edge servers. (Lai, He, Grundy, & Chen, 2020).

In the context of the edge computing paradigm, Hive (Teragni,, Moran, & Zabala, 2020) - an abstraction layer compatible with standard JavaScript and Node.js - was designed and implemented. It provides a distributed shared memory on top of existing web browser, like the ones present in smartphones or tablets. In this way Hive enables developers to take advantage of the biggest unused processing power available today, without incurring in the extra cost of deploying a network of devices. To this end Hive provides a virtual cloud server that enables collaboration and sharing among applications deployed on different distributed devices.

Regarding specific technologies, JavaScript (Zakas, 2016) is one of the core technologies in edge computing, most websites use it for implementing web page behavior on the client-side. All major web browsers have a dedicated JavaScript engine to execute the code on the user's device. Node.js is a major platform for building JavaScript applications for the server, cloud, mobile, and IoT platforms. However, a major challenge for research and tooling development for Java Script is the lack of a formal specification of its complete execution model. This execution model involves multiple event queues, some implemented in the native C++ runtime, others in the Node.js standard library API bindings, and still others defined by the JavaScript ES6 promise language feature. These queues have different rules regarding when they are processed, how processing is interleaved, and how/where new events are added to each queue. These subtleties are often the difference between a responsive and scalable application and one that exhibits a critical failure.

To mitigate the problem, this paper presents a comprehensive formalization of the Hive execution model, providing a high-level conceptual framework for reasoning about the execution of Hive applications. To define such model a formal definition of the JavaScript language was used and extended to include modern capabilities and custom semantics. This extended model was used to prove that the distributed

shared memory can operate on top of existing and unmodified web browsers. The proposed model guarantees the eventual synchronization of data across all the system and provides the possibility to have a stricter consistency using standard http operations. The technical feasibility of this proposal was empirically validated by a prototype that yields reasonably low propagation time and allows the distribution of preexisting JavaScript code without any major modifications

The paper is structured as follows. In Section 2 the Hive synchronization model is summarized. In Section 3 the formal semantics of Hive is defined on top of accepted formal semantics of asynchronous Java Script. In section 4, the usefulness of having formal semantics is shown across different applications. In section 5 a set of use cases are presented with the aim of corroborate the validity of the proposal, and a lightweight evaluation is performed by applying a propagation time metric. The paper is completed by reviewing and contrasting with related work in Section 6 before discussing lines of future work in section 7 and finally offering the conclusions.

THE HIVE SYNCHRONIZATION MECHANISM

The Hive synchronization mechanism, whose preliminary version was introduced in (Teragni, Zabala, & Blanco, 2018), aims to provide a variable consistency replication between the nodes connected to the distributed system, operating with eventual consistency (Bailis & Ghodsi, 2013) by default and allowing the programmer to explicitly request a higher consistency (at least causal consistency). There are two main actors in the process, the Memory Nodes, which are a source of truth for a fraction of the distributed shared memory, and the Processor Nodes, which are a large ephemeral fleet of devices that access and operate on top of the distributed shared memory.

The Memory nodes, analogous to the RAM of a physical computer, have by design the single responsibility of receiving and ordering mutations to their respective memory regions, providing eventual consistency and a query interface that complies with most of the standard http features, particularly read requests (GET), mutation requests (POST/PUT) and conditional requests (IF-MATCH / IF-NONE-MATCH headers). Those requirements are easily fulfilled by most of the modern cloud DBMS without any modification, successful tests have been performed with CouchDB, Amazon's SimpleDB, IBM's CloudAnt, and Google's Firebase. Firebase was used for implementing the Hive prototype, mainly due to the better performance it provided when propagating changes.

On the other hand, the Processor nodes, analogous to the CPUs of a physical computer, are intended to be standard web browsers (running on computers, tablets, mobile devices, etc.) where the synchronization layer (a JavaScript library) executes alongside the user code to propagate and receive mutations between the local memory and the Memory Node. Each Processor node holds a complete replica of the shared state, ensuring that it can execute to completion without needing to maintain a connection, tolerating uncontrolled network environments. When connection is reestablished an information exchange occurs with the Memory Node to receive and send the changes occurred to the shared memory since the last synchronization.

To maximize the scope of the proposed work, the synchronization happens in the background, without an explicit call from the program, allowing the seamless use of already existing libraries, and without needing a modified version of the browser, ensuring that all the customer devices can become processing nodes by just entering the correct website.

Hive implements a Conflict-free Replicated Data Type (CRDT) (Preguiça, Baquero, & Shapiro, 2018) to represent the complete state of the shared memory. It creates a tree-like structure where both complex entities (objects, arrays) and simple values (Numbers, Strings, Booleans, etc.) can be represented, and the mutations have the smallest possible scope of influence, by separating nested structures into flat, uniquely identified siblings, reducing the risk of concurrent mutations. A LWW CRDT is used to handle the case

when two or more nodes operate in the same specific region at the same time, providing an eventually consistent state for the memory (Teragni,, Moran, & Zabala, 2020).

Each mutation and its associated location received by the Memory Nodes is applied to its local replica of the shared state, emitting in the process a unique identifier that imposes a global total ordering of the events, to be replicated in the processor nodes as soon as connectivity permits. Figure 1**Error! Reference source not found.** shows the process that handles the command to create a new instance within the shared memory space. The process begins with a validation that the object that was requested to create does not already exist in the local replica, avoiding the creation of duplicate instances and ensuring that the addition operation is idempotent, thus being able to tolerate a classic problem of distributed systems that is the reception of duplicate messages. In case the requested object does not already exist in the local replica then it is evaluated if the data type is primitive, which would imply that the value received is the resulting value, and otherwise a new object is created, and the changes are applied within that object according to the mapSnapshotToObject function. Once the local construction of the object is completed according to the information received, it is added to the collection of loadedObjects using its unique identifier as a key, allowing future access to it. Finally, references to the newly created object are searched in the local replica, updating those pointers, correcting the object graph, thus ending the creation process.

Flowchart in Figure 1 shows that locally reflecting the existence of a new instance in the shared memory space is a functionality that strongly depends on two additional processes: the mapSnapshotToObject process on the one hand, that is the function responsible for updating the values of local objects to reflect the shared state reported by Firebase and the checkForReferences process on the other hand, that guarantees the integrity of the object graph in the presence of out-of-order updates.

The Hive prototype also allows the construction of strict consistency by using an additional data structure (i.e., Locks, where the identifiers of the objects that are taken are stored) and http conditional requests. Also, Hive has the infrastructure to distribute processing without dealing with the complexities inherent in synchronization by providing event queue abstraction that operate in a similar way to the producer-consumer model. The full specification of the entire prototype can be read in (Teragni M. , 2022).

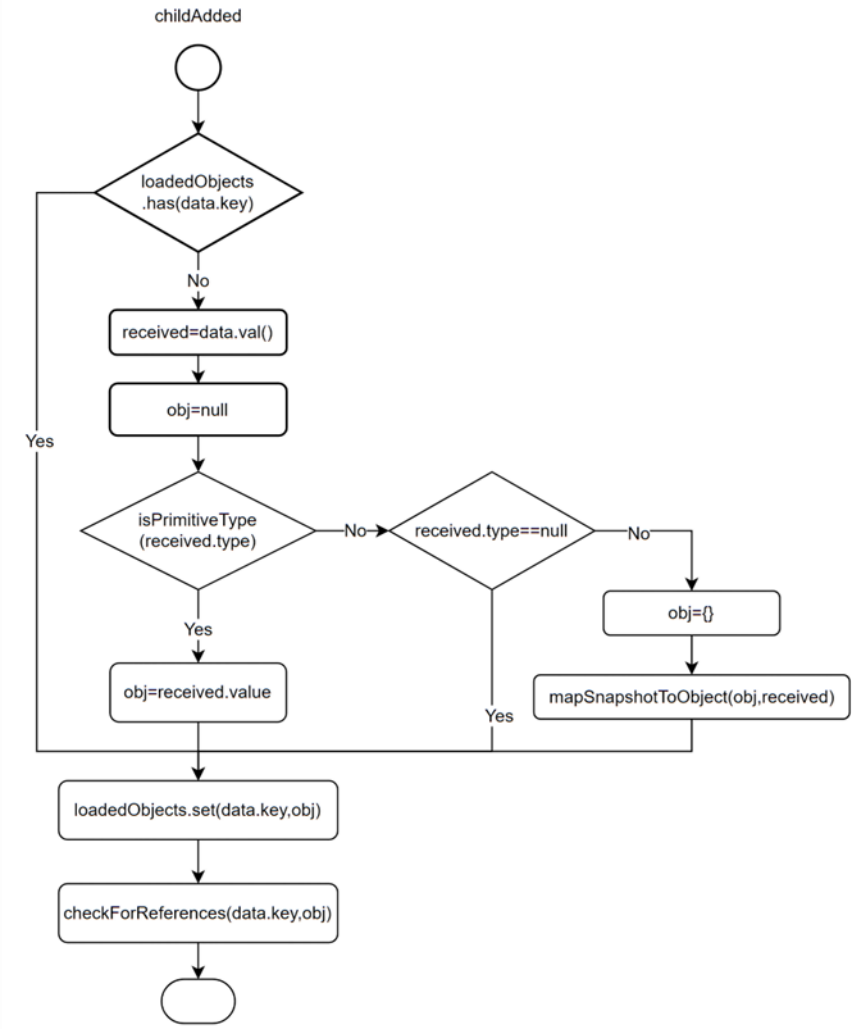

Figure 1. Hive behavior – ChildAdded

FORMALIZATION OF THE HIVE SYNCHRONIZATION MODEL

The formalization of the proposed synchronization model is essential to guarantee the desired semantics and to allow programmers to reason correctly about the source code they are writing.

Since Hive is based on JavaScript, the Hive formalization is accordingly based on the most accepted JavaScript formalizations. In particular, the operational semantics of JavaScript was presented in (Guha, Saftoiu, & Krishnamurthi, 2010) where λjs, an abstract syntax of the essential operations, together with a set of formal rules defining their semantics, were defined. In addition, a desugaring process was defined by expanding the rest of the JavaScript operations into semantically equivalent subroutines made up of essential operations. Afterwards, Guha's work was extended in (Loring, Marron, & Leijen, 2017) defining the λ_{async} language that incorporates the semantics of micro-tasks and event queues which are necessary to enable the modeling of the asynchronous behavior of both JavaScript and Node.js, where different priorities of promises are handled.

These recognized formalizations were taken as a starting point for building the Hive formalization. In this section, the definitions are presented incrementally, starting with the most elementary synchronization constructs (simple synchronization), continuing with a promise-based synchronization model, to finally describe the Hive's full synchronization constructs.

Notation

For defining the Hive semantics, the same well-known notation used in λ_{js} and λ_{async} is used. However, in order to facilitate the reading of this work, a few key points are presented:

- As defined in (Guha, Saftoiu, & Krishnamurthi, 2010), the execution context is referred as E, and $E[e]$ implies evaluating the expression e in such context.
- The expression $e[x/v]$ is the substitution of the variable x for the value v in all the occurrences of x in the expression e.
- H is used to refer a memory heap that contains a set of locations l and the values v associated with them. $H[l]$ is equivalent to obtaining the value associated with the location l, and $H[l \longmapsto v]$ is equivalent to mutating the heap by saving the value v in the location l.
- The complete computational semantic is expressed in rules that follow the pattern $H \vdash e \rightarrow H' \vdash e'$ that means that in a context H, applying the expression e is equivalent to applying the expression e' in the context H', allowing us to perform iterative reductions.
- The semantic rules are applied reiteratively until no other rule is applicable.

A simple synchronization model

Based on the synchronization mechanism described in section 2, Hive builds a shared memory between different JavaScript execution contexts and integrates the synchronization process into the execution of each node. Specifically, the synchronization of the different memory replicas is based on two operations, on the one hand the operation push() propagates the local changes to the other nodes, and on the other hand, the operation pull() is responsible for reflecting the external nodes changes on the local memory replica.

The abstract language λ_{hive} is defined as an extension of λ_{async}. Equation 1 shows a slice of the abstract syntax of λ_{hive}. For conciseness, only the extra constructs added to λ_{async} - the expressions push() and pull() - are presented. These expressions are syntax sugar because they are completely constructed on top of the existing structures but provide a shorthand way for describing and identifying the synchronization points.

$$e \in \text{Exp} ::= \ldots \mid \text{push}() \mid \text{pull}()$$

Equation 1. λ_{hive} Syntax extensions

The process of making pending changes effective by replicating them from the local memory to the global replica is performed by the push() operation, which returns a sequence number for the propagated changes. Equation 2 displays its semantics. The definition is abstract enough allowing different implementations depending on the needs and the available providers.

$$\frac{LS = \{\ldots, (n, \{(l_1, v_1), \cdots, (l_x, v_x)\})\} \quad D = H \backslash LS[n]}{push = (\textbf{func}\ (\emptyset)\{\ \textbf{return}\ LS = (LS, (\textbf{upload}(D), D))\})}$$

$$\frac{SH = \{\dots, (n, \{(l_1, v_1), \cdots, (l_x, v_x)\})\} \quad D = \{(l_\mathrm{p}, v_\mathrm{p}), \cdots, (l_z, v_z)\}}{\mathbf{upload}(\mathrm{D}) = SH = (SH, (n+1, D)); n = n+1}$$

Equation 2. Shared Memory Semantics – Send-Changes

The definition in Equation 2 assumes a set LS that contains the complete history of changes, ordered by their respective sequence numbers. Then the set D contains all those elements found in memory space H that are not contained in the latest recorded version. The operation push() uploads those changes and updates the set LS with the latest propagated changes. Thus, the upload() function is intentionally left unspecified to provide flexibility in the implementation of the Memory Nodes if it fulfills the following conditions:

- The upload function operates against a Memory Node.
- It receives a set of changes to apply, and synchronously ensures that the state of the Memory Node reflects those changes.
- It returns a comparable unique identifier for those changes, establishing in the process a global ordering of the mutations received by the Memory Node.

On the other end of the replication, the Equation 3 describes the pull() operation, responsible of applying remote mutations to a local replica. This operation applies the changes included in the set D, that come from changes made in other memory nodes, to the local memory H. It also updates the set LS. Mutations are applied in order based on the sequence numbers associated with each of them.

$$\frac{\begin{gathered} LS = \{(n, \{(l_1, v_1), \dots, (l_x, v_x)\}), \dots, (n+k, \{(l'_1, v'_1), \dots, (l'_y, v'_y)\} \\ C = (r, \{(l''_1, v''_1), \cdots, (l''_z, v''_z\}) \\ D = \{\, (l, v) \in C[r] | \forall n < r \ \vee \ l \notin LS[n] \vee n < r\} \\ D \approx \{(a_1, b_1), \cdots, (a_w, b_w)\} \end{gathered}}{pull = (\mathbf{func}\ (\emptyset)\{\ \mathbf{return}\ LS = LS, \{r, D\}; H[a_1 \longmapsto b_1]; \cdots; H[a_w \longmapsto b_w]\})}$$

Equation 3. Shared Memory Semantics – Receive-Changes

The definitions of $pull()$ and $push()$ in combination make it possible to guarantee the eventual convergence of any node, assuming that these functions are called periodically by each replica. That is to say, to build a consistent shared memory, the periodic and eventual execution of these processes must be assured.

Limitations of the simple synchronization model

Given the single-threaded execution of JavaScript, constructing an infinite loop to make these calls, as shown in Code 1, implies that, even if the synchronization is carried out, the user code will lose priority, since it will never be executed unless it is specifically built on Promises and continuations. And since the special expression • is not part of the accessible JavaScript syntax, it is not possible to trigger the asynchronous processing queue within a function that did not terminate, and this loop cannot be constructed without using a special version of the virtual machine.

$$while(true)\ (pull(\emptyset); push(\emptyset); \bullet)$$

Code 1. Invalid Synchronization loop

Synchronization model based on promises

Based on the semantics of asynchronous promises defined in (Loring, Marron, & Leijen, 2017), a recursive operation can be constructed that is periodically executed, making use of the expression in Code 2.

$$\text{AsyncRepeat} = (\textbf{func}\ (f)\{\ \textbf{return}\ Promise().then((f(\emptyset)); (\text{AsyncRepeat}(f))); \bullet\})$$

Code 2. Shared Memory Semantics – Async-Repeat

The function associated with AsyncRepeat receives a second function as a parameter, scheduling its execution as a continuation of a new promise that, when resolved, will recursively call AsyncRepeat with the same parameter, causing a function f to always exist pending to execute. Starting from the possibility of generating non-blocking loops, then, the synchronization can be started by executing the code displayed in Code 3.

$$(\text{AsyncRepeat}(\textbf{func}\ (\cdots)\{\ \textbf{return}\ \text{pull}(\emptyset); \text{push}(\emptyset)\}))$$

Code 3. Shared Memory Semantics – Full Background-Sync

This implies the initialization of a background process that periodically, depending on the load of the asynchronous queue, will perform an update of the local data, and a propagation of the mutations to the other nodes.

Limitations of the synchronization model based on promises

This implementation, although functional and consistent with the promise-based model, has two major limitations, firstly, the process that allows finding the differences between the current Heap and the last registered state (i.e., the precondition of push()) is a process with an extremely high computational cost, and secondly, the synchronization could be blocked as seen in the case illustrated in Code 4. Considering that once a Promise is created, it will not be executed until the end of the current execution, assuming that obj is a shared variable, the synchronization will not occur until the execution of the user code ends, causing a total stop of the synchronization process. The simple program in Code 4 never completes its execution, and thus effectively prevents any type of asynchronous process from starting. Both detecting and avoiding these constructs would require modifying the JavaScript runtime, which is not convenient.

```
let obj={x:1}
while(true)
{
  obj.x+=1;
}
```

Code 4. User Code - Infinite loop

The HIVE Synchronization model

Both issues described above can be solved taking advantage of a feature introduced in the ECMAScript specification ES6 (ECMA, s.f.), known as Proxy Objects. This feature allows the creation of objects that intercept all the messages and operations applied to a particular object. In this way, before starting the shared memory synchronization a set of proxies that intercept requests for mutations on local objects can be built, thus giving the possibility of performing the synchronization process even when the user code is running.

Neither λ_{js} nor λ_{async} include Proxy Objects in their constructs. And in fact, in (Krishnamurthi, 2011) that is built on top of λ_{js}, Proxy objects are mentioned as future work. Therefore, the syntax for creating Proxy Objects is proposed in Equation 4. Its associated semantics is defined in Equation 5, which is compliant with the ECMAScript specification ES6. The rule states that the expression denotes the creation of an object holding three properties, the location of the intercepted (l), the definition of the interceptors (h), and the prototypical property that points to the Proxy prototype.

$$e ::= e \mid Proxy(l\ e)$$

Equation 4. Create-Proxy syntax

$$\frac{(l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H}{Proxy(l\ h) \hookrightarrow \{\text{"\_\_proto\_\_":Proxy.prototype , "target": } l\text{ , "handler": } h\ \}}$$

Equation 5. Create-Proxy Semantics

The Hive synchronization mechanism makes use of Proxy objects and extends the behavior of JavaScript objects accessors (Getters and Setters) in the cases where proxy objects are involved. Equation 6 shows the semantics at play when the proxy interceptor has a function f associated with the keyword "get", this function must be executed, receiving as parameters the intercepted object, and the name of the requested property.

$$\frac{h = \{\cdots \text{"get"}: f \cdots\} \quad f = \textbf{func}\ (target\ \ property)\{\ \textbf{return}\ e\} \quad (l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H}{\{\text{"\_\_proto\_\_":Proxy.prototype , "target": } l\text{ , "handler": } h\ \}[str_x] \hookrightarrow (f((\textbf{deref}\ l)\ \ str_x))}$$

Equation 6. Proxy Semantics – GetField-Intercepted-Proxy

In case the interceptor does not define an operation associated with "get", the requested property of the real object will be returned, as defined in Equation 7.

$$\frac{h = \{str'_1 : v'_1 \cdots str'_n : v'_n\} \quad \text{"get"} \notin (str'_1\ \cdots\ str'_n) \quad (l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H}{\{\text{"\_\_proto\_\_":Proxy.prototype , "target": } l\text{ , "handler": } h\ \}[str_x] \hookrightarrow (\textbf{deref}\ l)[str_x]}$$

Equation 7. Proxy Semantics – GetField-Non-Intercepted-Proxy

Similarly, when an update is requested and the interceptor has a function associated with the keyword "set", the result of the operation will be the execution of such function, passing the name of the property and the value to be assigned as parameters to the intercepted object, as defined in Equation 8.

$$\frac{h = \{\cdots \text{"set"}: f \cdots\} \quad f = \textbf{func}\ (target\ \ property\ \ value)\{\ \textbf{return}\ e\} \quad (l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H}{\{\text{"\_\_proto\_\_":Proxy.prototype, "target": } l\text{, "handler": } h\ \}[str_x] = v \hookrightarrow f((\textbf{deref}\ l)\ \ str_x\ \ v)}$$

Equation 8. Proxy Semantics – SetField-Intercepted-Proxy

And finally, if the interceptor does not have a definition for the write operation, the proxy will trigger the operation of the intercepted object, as specified in Equation 9.

$$\frac{h = \{str'_1 : v'_1 \cdots str'_n : v'_n\} \quad \text{"set"} \notin (str'_1 \ \cdots \ str'_n) \quad (l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H}{\{\text{"\_\_proto\_\_":Proxy.prototype , "target": } l \text{ , "handler": } h \}[str_x] = v \hookrightarrow (\mathbf{deref}\ l)[str_x] = v}$$

Equation 9. Proxy Semantics – SetField-Non-Intercepted-Proxy

Thus, taking advantage of the Proxy objects the previously proposed synchronization mechanisms can be improved, allowing the construction of arbitrary user code, and performing the synchronization with a smaller granularity. Equation 10 shows the specification of the share() function that builds a Proxy to the object it receives as a parameter. It uses a function g to intercept the "get" operations, causing an update of the local data before returning the requested value, and uses a function s to intercept the "set" operations, sending to the server the information of the specific updates on the mutated object.

$$\frac{\begin{array}{c}(l, \{str_1 : v_1 \cdots str_n : v_n\}) \in H \\ g = (\mathbf{func}(target\ prop)\{\mathbf{return}\ pull(\emptyset); target[prop]\}) \\ s = (\mathbf{func}(target\ prop\ value)\{\mathbf{return}\ target[prop] = value; push((l, target))\})\end{array}}{share = (\mathbf{func}(l)\{\mathbf{return}\ Proxy(l\ \{\text{"get"}: g,\ \text{"set"}: s\})\})}$$

Equation 10. λ_hive Semantics – Add-To-Shared-Memory

The share() function is exposed to the programmer through two different calls, hive.set() to add an object to the shared memory space and hive.get() to get an existing object from the shared memory space. Both operations are only syntax sugar added to simplify comprehension thus do not require modifications to the JavaScript runtime. Equation 11 defines the syntax.

$$e ::= \ e \mid hive.get(e) \mid hive.set(e, e)$$

Equation 11. λhive Sugar Syntax – get & set

Both hive.get() and hive.set() operations are defined in the context of an object R that resides in the memory space H acting as an access dictionary, which keeps a record of all objects referenceable directly by the programmer.

In Equation 12 the hive.set() operation is defined. It receives a string to be used as a key, and an object to be incorporated into the shared memory space. The provided object is added to the R dictionary and the v_x object is intercepted by calling to share(). Since R∈H, the changes to this dictionary will also be propagated with the rest of the changes.

$$\frac{R = \{str_1 : v_1 \cdots str_n : v_n\} \quad R \in H}{hive.set(str_x, v_x) \hookrightarrow R[str_x] = v_x; share(v_x)}$$

Equation 12. Shared Memory Semantics – hive.set

On the other hand, as defined in Equation 13, the function hive.get() retrieves a direct reference to a referenceable object already present in the shared memory by obtaining the value associated with the key provided in the dictionary *R*, and guarantees that such reference is intercepted by the function share() to detect and propagate the changes made in that instance.

$$\frac{R = \{str_1 : v_1 \cdots str_x : v_x \cdots str_n : v_n\} \quad R \in H}{hive.get(str_x) \hookrightarrow share(R[str_x])}$$

Equation 13. Shared Memory Semantics – hive.get

In the fragment in Code 5, it is easy to appreciate how this approach eliminates the previously mentioned problems. By intercepting all the mutations (set operations) Hive keeps a complete list of the changes that were applied locally since the last synchronization point and therefore, all the information needed to propagate those changes to the rest of the system, without the need to do any swipe of the complete memory. On the other hand, by intercepting all the accessors, Hive ensures that there is a window to synchronize the changes with no regard to the user's code or if it is non-terminating. Intercepting the Set operation gives us a chance to propagate changes to other devices as soon as they happen, and intercepting Get operations provides the opportunity to ensure the local state is up to date before proceeding with the user's code.

```
let obj=hive.set('key',{x:1})
while(true)
{
  obj.x+=1;
}
```

Code 5. JS Code - Shared Infinite loop

To observe the underlying behavior, this code can be expanded by replacing the hive.set() call that internally performs a share(), returning a Proxy, which intercepts the read and write operations. Thus, the program in Code 5 is equivalent to the code shown in Code 6. When the += operator is applied, a reading of the current value of the variable is performed and then the addition is executed.

```
R['key']={x:1};
let obj= share(R['key']);
while(true)
{
pull();
obj[x]=obj[x]+1;
push((ref(obj), obj));
}
```

Code 6. JS Code - Expanded Shared Infinite Loop

It is clear to see in the expanded version of the code that it reasonably gives rise both to the reception of mutations from other nodes, before performing the requested operations, and to the propagation of the mutations that caused as soon as they happen, and how the infinite loop built by the user does not prevent or harm the synchronization of the shared memory.

In practice, both calls to pull() and push() can be grouped into batches to avoid unnecessary synchronizations, thus being able to control the length of the acceptable inconsistency window in the context of eventual consistency.

USEFULNESS OF λ_hive FOR VERIFICATION

Verifying the JavaScript Standard specification

The formalization presented in the previous sections is essential to verify the behavior of the synchronization layer in the context of JavaScript. JavaScript is specified by ECMA-262 (EcmaScript) (ECMA, n.d.) in an imperative pseudo-code but lacks a full formal definition. Although that specification is enough to be used as a guideline to implement a JavaScript Virtual Machine, it does not provide a clear

mechanism to reduce expressions, thus making it hard to detect edge cases, requiring literally to interpret it step by step to arrive at a result.

In this section, the correspondence between the informal ECMA-262 specification and its corresponding Hive formalization is showed. For this purpose, the ECMA-262 specification of the Get operation (reading a value) from a Proxy Object, displayed in Figure 2, is evaluated as an example. The same procedure can be applied to the rest of the operations.

```
The [[Get]] internal method of a Proxy exotic object O takes arguments P (a property
key) and Receiver (an ECMAScript language value). It performs the following steps when
called:

1. Assert: IsPropertyKey(P) is true.
2. Let handler be O.[[ProxyHandler]].
3. If handler is null, throw a TypeError exception.
4. Assert: Type(handler) is Object.
5. Let target be O.[[ProxyTarget]].
6. Let trap be ? GetMethod(handler, "get").
7. If trap is undefined, then
  a. Return ? target.[[Get]](P, Receiver).
8. Let trapResult be ? Call(trap, handler, « target, P, Receiver »).
9. Let targetDesc be ? target.[[GetOwnProperty]](P).
10. If targetDesc is not undefined and targetDesc.[[Configurable]] is false, then
  a. If IsDataDescriptor(targetDesc) is true and targetDesc.[[Writable]] is false,
then
    i. If SameValue(trapResult, targetDesc.[[Value]]) is false, throw a TypeError
exception.
  b. If IsAccessorDescriptor(targetDesc) is true and targetDesc.[[Get]] is undefined,
then
    i. If trapResult is not undefined, throw a TypeError exception.
11. Return trapResult.
```

Figure 2. ECMA-262 Object Proxy Get Definition

The Hive formalization displayed in Equation 6 and Equation 7 covers the steps 1 through 6 of this sequence The specific condition evaluated on step 7 is the difference between those two behaviors.

The equation in Fig 6 shows the semantics at run time when the proxy interceptor – the handler object - has a function f associated with the keyword "get" ($h = \{\cdots \text{"get"}: f \cdots\}$); Thus, the variable trap is not undefined. Then this function must be executed in step 8, receiving as parameters the intercepted object, and the name of the requested property($f\left((\text{deref } l)\ \ str_x\right)$). Finally, the result is returned in step 10.

In case the interceptor does not define an operation associated with "get" ($h = \{str_1' : v'_1 \cdots str_n' : v_n'\}$ "get" $\notin (str_1' \ \cdots \ str_n')$), thus trap is undefined and the requested property of the real object will be returned, as defined in equation 7.

The same procedure can be applied to analyze the rest of the Proxy Operations, and to the sugared operations defined in this article.

Verification of the λ_{hive} synchronization mechanism

The operations proposed in λ_{hive} can be grouped based on the direction of the information flow. On the one hand, the local changes are propagated to the remote nodes and on the other hand, the respective changes are received from the remote nodes.

Propagating local changes

The sending of local changes is composed of two steps, detecting which changes existed since the last synchronization and then sending these changes to the memory node in charge of their replication and distribution. In Equation 2 the change set to be sent is defined as the difference between the current memory state versus the state of the last synchronized record $D = H \backslash LS[n]$. There are multiple factors that prevent that comparison directly, from the lack of a JavaScript API that exposes memory, to performance issues associated with performing the search. The Proxy Object defined in Equation 10 is used to obtain the set of differences. It associates a function to the $handler$ set, sending each modification to the synchronization platform. Thus, instead of going through all the memory looking for the differences, the set to be sent is constructed in a trivial way as the accumulation of changes received through that interception.

While the set LS, used to keep track of what was synchronized so far to know what should be propagated next, is equivalent to part of the functionality provided by the Firebase libraries, which internally handle the versioning of the data necessary to provide eventual consistency. **Error! Reference source not found.**Once the set of changes to be propagated is determined, Equation 2 details the semantics of sending that set to the memory node, whose main concerns are the existence of SH as a representation of the shared state to which the new changes will be added, and the generation of a new unique identifier associated with such set that allows to synchronize correctly the different local replicas to each node. Then the set SH is equivalent to the internal state handled by firebase, which can receive mutations through the Set () or Update() operations, generating in both cases a unique identifier.**Error! Reference source not found.**

Receiving remote changes

Once received by the memory node, the changes are propagated to the different clients, where each one must apply the semantics defined in

Equation 3. Shared Memory Semantics – Receive-Changes

3, updating their local state in case the changes received are more recent than the local state. In particular, the set C groups the new version number together with all the mutations that must be applied. When notifying of changes Firebase allows the subscription to different events for the creation, deletion, or modification of elements, previously filtering those modifications received that have occurred before local mutations on the same location, analogous to the set $D = \{ (l, v) \in C[r] | \forall n < r \ \vee \ l \notin LS[n] \bigvee n < r \}$.

When applying changes to local memory, modeled as $LS = LS, \{r, D\}; H[a_1 \mapsto b_1]; \cdots; H[a_w \mapsto b_w]$, the update of the set LS is performed directly by Firebase which internally handles the version numbers of the data structure. The mutations to the set H that represents the accessible memory of the program are performed using the logic described in the flowchart in Figure 1, which shows the three notifications that Firebase issues to relevant changes.

The mapSnapshotToObject function is used to identify and modify those fields that need to be updated from living objects in the JavaScript memory. On the other hand, the checkForReferences function is used to update the references between local objects, ensuring that the relationship graph between local objects is homologous to the graph on the other remote nodes.

EMPIRICAL VALIDATION AND EVALUATION OF THE PROPOSAL

Use Cases

To validate the operation of the synchronization layer described in this article, multiple experiments were designed and implemented that seek to make use of the synchronization mechanisms provided from the

point of view of an application developer. Experiments can be downloaded from https://hiveproject.github.io/Firebase/Demo

Simple synchronization

The smallest example that makes use of the synchronization engine consists in displaying on the screen the content of an object that is being synchronized, so that it is clearly perceptible when a node makes a mutation on it. The complete Code can be found at https://hiveproject.github.io/Firebase/Demo/Samples/PrintData

Concurrent mutations in real time

In this scenario, two factors are added that allow the user to clearly make visible the added value of the proposed synchronization layer. First, each connected node will make concurrent edits to its own data, an exercise that by design will not cause conflicts, but will allow it to take advantage of synchronization between multiple nodes. In addition, a graphical representation was provided that allows the user to easily appreciate that the synchronization model preserves and propagates intermediate values before repeated modifications and also can observe the low propagation times, and glimpse the applicability of this technology to build multiplayer games. In particular, each user (connected browser) have a small square that the user can move at ease using the mouse, and all users will be able to observe in real time all the movements made by each other user. Full code at https://hiveproject.github.io/Firebase/Demo/Square/

Processing Distribution

This example makes use of processing queues to distribute the calculation of minimum paths between two points in a maze, allowing nodes to be added to speed up computation. There are two separate applications (web pages) that use shared memory as a synchronization mechanism. This is a demonstration to the compatibility of the transparent synchronization layer with pre-existing libraries, since it makes use of third-party JavaScript code that worked on shared objects without the need for any modification. Full code at https://hiveproject.github.io/Firebase/Demo/Astar/

Heavy object synchronization

Like the previous case, this example seeks to distribute processing, the main difference being that the data that is synchronized between the nodes is much more extensive. Complete code in https://hiveproject.github.io/Firebase/Demo/ImageProcessor

Prototype evaluation: Propagation times

The main metric by which the synchronization layer is evaluated is the **propagation time**, that is, the number of milliseconds that elapse from when a node makes a change until it is received by the other nodes in the network. Although performance is not the primary objective of this proposal, and there is much room for improvement, initial measurements are encouraging.

To collect empirical samples on this metric, the following experiment was designed, similar in operation to the **ICMP ping** command. Given an initial node (also called **Ping Node**, where the samples will be collected), 3 empty collections (called **before**, **middle** and **after**) located in the shared memory will be initialized. This implies that the other nodes of the system will have access to them.

- The **Ping Node** will insert into the collection **before** the current date and time measured according to your clock.
- The **Echo Node** will monitor the collection **before** waiting to find a new item in that collection.
- The **Echo Node** will insert into the **middle** collection the current date and time measured according to your clock.

- The **Ping Node** will detect the new record in the **middle** collection, inserting in the collection **after** the current date and time according to your clock

The process is described in Figure 3.

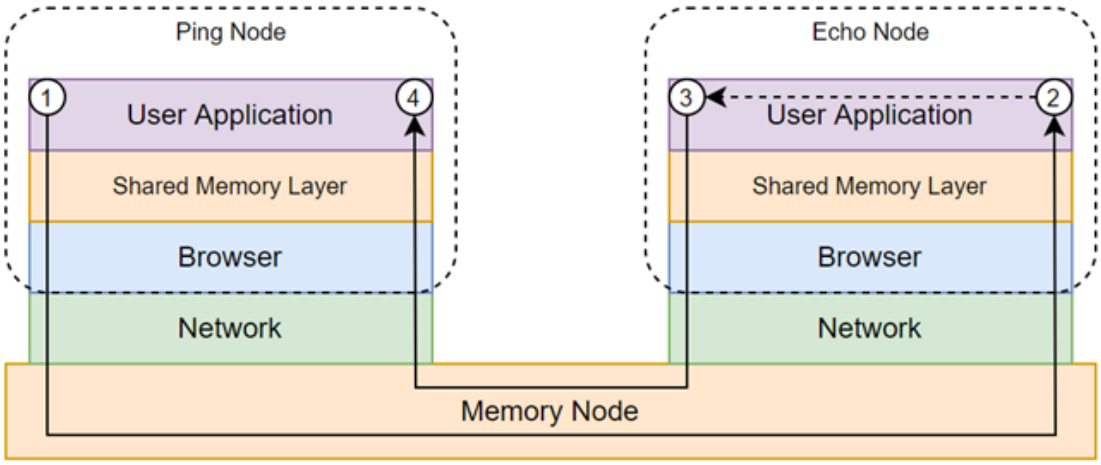

Figure 3. Prototype Evaluation - Propagation Time Benchmark

Once the cycle is over, the date and time taken at the beginning and end of the process (in the **before** and **after** collections) that were taken using the same clock will be counted, so they will be an effective measure of the full travel time (roundtrip time) for that cycle, and the corresponding record can be used for the **middle** collection. to measure the difference between the clocks of the nodes involved. Times are be measured as integer values in Epoch Unix Time. It is important to keep in mind that, since the measurement is carried out as part of the user's application, the times and delays include the synchronization costs of the components together with the latencies and delays of the networks involved in the experiment and are a concrete measure of time that can be observed by any other application that makes use of the proposed synchronization mechanisms. Table 1 shows values captured in this way, adding the columns corresponding to PropTime (propagation time) and ClockDrift (difference between clocks) calculated according to the measurements defined in Equation 14.

$$RTT = After - Before$$
$$PropTime = \frac{RTT}{2}$$
$$ClockDrift = Before + PropTime - Middle$$

Equation 14. Prototype Evaluation - Propagation Time Measurements

Before	***Middle***	***After***	***RTT***	***PropTime***	***Clock Drift***
1624981407842	1624981408818	1624981408139	297	148.5	-827.5
1624981407942	1624981408919	1624981408244	302	151	-826
1624981408043	1624981409013	1624981408334	291	145.5	-824.5

Table 1. Prototype Evaluation - Example of propagation times

The described test was performed with a **Ping Node** located in the Autonomous City of Buenos Aires, Argentina, against three different **Echo Nodes**, collecting more than a thousand samples for each case:

- One present on the same local network

- One present in the same city (Autonomous City of Buenos Aires, Argentina), in another physical location.
- One present in another country (London, United Kingdom)

In this way it can be seen given the proposed architecture, where the **Memory Node** is located in the Cloud, that the distance between the Nodes to be synchronized does not impact the propagation times.

	Average propagation time	*Average propagation time*
Very Close Nodes (Local Network)	220.39 ms	187 ms
Nearby Nodes (same city, CABA, AR)	236.51 ms	190 ms
Far Nodes (different countries), London, UK)	215.36 ms	150 ms

Table 2. Prototype Evaluation - Propagation Time Measurement Results

Of the 3000 samples taken between these three scenarios, as can be seen in the histogram in the Figure 4, the changes were propagated in less than 260ms for 72% of the attempts, 20% reached consensus in less than 180ms, being the average time 175ms.

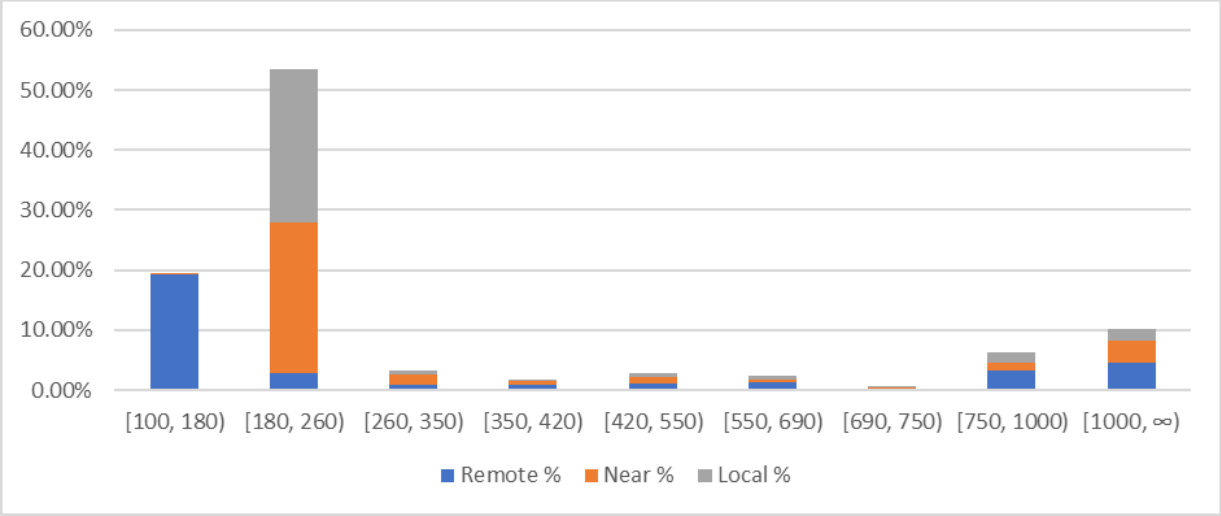

Figure 4. Prototype Evaluation - Propagation Time Histogram

It is important to note that the times are consistent regardless of the distance between the nodes, since there is no direct connection between them, and consequently that the performance of the system tends to be stable and predictable.

RELATED WORK

There are multiple technological platforms with objectives similar to those proposed in this work. This section lists some of the most relevant, focusing on what characterizes them, and how the current proposal differs from them, leaving a clear contrast of the differential value provided by this work.

Related Technologies

Hadoop

Hadoop is an open-source framework for distributed processing, which handles the storage and processing of large volumes of data in scalable clusters. Hadoop allows users to execute code written in the Java language where there are constructions, such as threads, shared access, and locks, without guaranteeing their respective semantics, and without failing or throwing errors at undefined behaviors. (Apache Software Fundation, 2019) (White, 2012)

Python

Python is one of the languages that has grown the most in recent years, especially in the scientific context, mainly due to design decisions that prioritize the comprehensibility of the code. It has a large number of libraries that allow the execution of distributed code, counting among the most adopted parallelPython, DisPy and Celery (Python, General Python FAQ, s.f.) (Peters, s.f.) (Python, Parallel Processing and Multiprocessing in Python, s.f.)

These implementations entail two major problems that the present work seeks to mitigate. First, as in the previous case, there is a general-purpose programming language whose semantics could not be fully adapted to the distributed context, in particular all these libraries expect functional code to be written, but they have no way to prevent the programmer from making use of any of the other tools that Python provides (such as object orientation, static variables, etc.) giving rise to indefinite behavior.

Secondly, most of these implementations require a conscious and literal management of the execution environments, where the programmers must inform where they will execute, adding complexity to their task, in addition to the complexity inherent in the tolerance to failures, retries, etc.

Map Reduce

There are multiple libraries that allow the distribution of processing defined in JavaScript focusing on the map-reduce pattern. The main ones are MRSJ, Acio.js and Pando.All these are based on the Map-Reduce paradigm, which simplifies the distribution of computation, but limits the possible application cases, that is, they do not allow solutions of general use, but focus on solving high computing problems. (Dean & Ghemawat, 2004) (Ryza & Wall, 2010) (Constela, 2019) (Lavoie, Hendren, Desprez, & Miguel, 2019)

Hive differs strongly from these alternatives since:

-It is general purpose. Putting the focus on local state synchronization rather than simple processing distribution, it can encompass more use cases, allows developers to make use of all the language at their disposal, and allows pre-existing libraries to be transparently reused.

-It allows to reach much greater scales by making use of the Memory Nodes (similar to pando servers) as services in the Cloud, and by treating the nodes connected to the system as ephemeral clients, there is no bottleneck of having a single monolithic server that is responsible for replicating the messages.

- By concentrating on the execution of local code, the applications created that use this platform can be designed in such a way that it can be operated in a disconnected way, accepting the eventuality of consistency, while the above mentioned platforms require connectivity to operate at all times.

Generic Messages

Generic Messages (GEMs) (Bonetta, Salucci, Marr, & Binder, 2019) is a library for Node.Js, complements JavaScript by adding controlled and secure shared memory that allows users to take advantage of hardware availability, thus being able to build cache for services that require scaling, optimize communications between microservices, improve data transfer times and achieve zero-copy messaging. GEMs implements the Java memory model.

This implementation has several points in common with the proposal in this article. Although both are based on the same language and allow the construction of a shared memory, there are fundamental differences from the scope and some design principles. In particular, GEMs makes use of a modified execution context, leaving aside the restriction of this project of not requiring any modification to the execution engine in order to take advantage of pre-existing devices.

Distributed Execution Framework

Distributed Execution Framework (DEF) (Feilhauer & Sobotka, 2016) consists of a set of applications that provide a PaaS where various clients using different programming languages can build and make use of an extensive library of highly parallelizable functions. DEF allows the construction and execution of algorithms in the cloud, making use of the processing power available there.

Hive, on the other hand, seeks the opposite, to decentralize processing to take advantage of the hardware present in customers, using the cloud merely as a mechanism of persistence and synchronization.

Formal models

Numerous works have developed formal models to explain different aspects of Cloud Computing and to reason rigorously about its properties. In particular the proposal in (Amoretti, Grazioli, & Sen, 2015) defines a formal semantics for high-level actions supporting Mobile Cloud computing on a particular framework. In (Loulergue, Gava, Kosmatov , & Lemerre, 2012) a usual software stack of a cloud environment is analyzed from the perspective of formal verification. This software stack ranges from applications to the hypervisor, arguing that most of the layers could be practically formally verified. In (Sahli, Bouanaka , & Taki Eddine Dib, 2014) the Maude formal language is used to execute and analyze a formal specification of a cloud computing architecture and its elasticity. (Challita, Paraiso , & Merle, 2017) proposes fclouds, a formal-based framework for semantic interoperability in multi-clouds. This framework contains a catalogue of formal models that mathematically describe cloud APIs and reason over them. The work in (Achilleos, Kritikos, Rossini, Kapitsaki, & Dom, 2019) defines the Cloud Application Modelling and Execution Language (CAMEL), which (i) allows users to specify the full set of design time aspects for multi-cloud applications, and (ii) supports the models@runtime paradigm that enables capturing an application's current state facilitating its adaptive provisioning.

The main difference between the proposal presented in this article and most of the alternatives is that it provides a distributed shared memory without requiring a specially modified runtime (meaning it can execute on pre-existing web browsers) and that it does not require any major rework of the user code since the synchronization itself is performed transparently behind the scenes. The Hive formalizations was built on top of the current formalizations of asynchronous JavaScript. Specifically, the work in (Loring, Marron, & Leijen, 2017) presents the first comprehensive formalization of the Node.js asynchronous execution model and defines a high-level notion of async-contexts to formalize fundamental relationships between asynchronous events in an application. This proposal was based on limited prior work on the semantics of the asynchronous JavaScript execution model and diagnostic tools for it. In particular: (Guha, Saftoiu, & Krishnamurthi, 2010) presents a core calculus for the procedural JavaScript language λjs which is extended by (Madsen, Tip, & Lhoták, 2015) to include a partial formalization of the asynchronous semantics for Node.js but omits the recently added ES6 Promise semantics and uses a simplified model for the drain rules for the event queues in Node.js. work in (Alimadadi, Mesbah, & Pattabiram, 2016) uses an implicitly defined model for asynchronous execution that is encoded in the dynamic analysis and rules for "Callback scheduling" and "Callback invocation". Other work applied concrete implementation details of a specific version of LibUV and Node.js, (Davis, Thekumparampil, & Lee, 2017) uses the deprecated v0.12.7 version of Node.js, and the details of the event-loop semantics are implicitly encoded in the tool implementation.

FUTURE WORK

The proposed distributed shared memory model showed potential for wide adoption, but requires attention to other areas before it can be considered production ready. There are three main lines of work to be investigated:

- **Security & Data Privacy**: The current design does not have a way to grant different permissions to different Processor Nodes, meaning that any node in the distributed system can read and modify every piece of data present in the shared memory. This can pose a security risk for the application and a privacy risk for the users whose data is collected by the system. A possible way to deal with this situation is to add a role-based authorization scheme to the operations of the Memory Nodes, since they already impose a physical partition of the distributed shared memory, allowing the definition of different regions in the shared memory that can only be accessed by a specific set of users.

- **To complete the integration with JavaScript**: Since JavaScript does not provide a way to programmatically inspect the program's memory nor the lexical scope where functions are defined without either modifying the browser or using debug/developer tools, the proposed synchronization platform does not replicate everything, it ignores classes, functions and delegates since their correct behavior cannot be guaranteed. Further research in solving this issue would improve the synchronization platform allowing automatic replication of the complete application code and not just its state, simplifying the programmer's job and enabling new features like automatic load balancing.

- **Distributed shared memory optimization**: The current implementation was not designed to be particularly efficient and there is a lot of room to optimize it, reducing the required resources and improving the user experience. We identified as the most impactful initiatives:

    - The Processing Nodes currently replicate and process mutations for the complete shared memory, even if they are not intended to access most of it. By reducing the synchronization scope, we can reduce the memory footprint, the network load, and the cost of propagating the changes through the distributed system

    - The current Distributed Garbage Collector is sufficient to ensure the memory does not grow beyond control but force a large computational cost on a single processing node and can cause slowness or inconsistencies on the distributed system. This can be greatly improved and poses its own research challenges.

CONCLUSION

In this article a distributed shared memory model that is transparently integrated into a standard JavaScript runtime environment has been described. The complexity of Edge Computing is reduced by enabling the transparent usage of the large number of untapped devices and providing concurrency to a classically single-threaded environment.

To define such model a formal definition of the JavaScript language was used and extended to include modern capabilities and custom semantics. This extended semantics was used to prove that the proposed distributed shared memory can operate on top of existing web browsers, without the need to modify them, allowing the incorporation of a huge number of currently available nodes, such as phones and personal computers, as a part of the distributed system.

The model guarantees the eventual synchronization of data across all the system and provides the possibility to have a stricter consistency using standard http operations, providing the developer with the required tools to obtain the desired behavior from the distributed system.

Supported on the formal specification, the technical feasibility of this proposal was empirically validated by a prototype that yields reasonably low propagation time and allows the distribution of preexisting JavaScript code without any major modifications

On the other hand, the comprehensive formalization of the Hive execution model offers a foundation for the construction of program analysis tools. It provides a high-level conceptual framework for reasoning about the execution of Hive application. Without the formalization to verify the correct behavior of the system becomes difficult, since the lack of failures during empirical testing does not mean the absence of them, especially in the context of a distributed system where many variables can affect the execution.

Additionally, the formalization allows developers to ensure that the synchronization mechanism is efficient since it avoids searches throughout the entire shared memory. Also, it allowed to prove that a blocking code by a user does not obstruct the synchronization of the shared state.

The proposed distributed shared memory model showed potential for wide adoption, but the work done so far focused on the technical viability of the synchronization process and requires attention to other areas before it can be considered production ready.

REFERENCES

Abdelwahhab, S., & Mostefai, S. (2020). Strategic Outsourcing to Cloud Computing: A Comprehensive Framework Based on Analytic Hierarchy Process. *International Journal of Cloud Applications and Computing, 10*(1), 11-27. doi:10.4018/IJCAC.2020010102

Achilleos, A., Kritikos, K., Rossini, A., Kapitsaki, G., & Dom, J. (2019). The cloud application modelling and execution language. *Journal of Cloud Computing, 8*.

Adeyinka Osuolale , F. (2022). Reactive Hybrid Model for Fault Mitigation in Real-Time Cloud Computing. *International Journal of Cloud Applications and Computing (IJCAC), 12*(1).

Ahuja , S., & Wheeler , N. (2020). Architecture of Fog-Enabled and Cloud-Enhanced Internet of Things Applications. *International Journal of Cloud Applications and Computing (IJCAC), 10*(1). doi:10.4018/IJCAC.2020010101

Alakbarov, R. (2022). An Optimization Model for Task Scheduling in Mobile Cloud Computing. *International Journal of Cloud Applications and Computing (IJCAC), 12*(1), 1-17.

Alimadadi, S., Mesbah, A., & Pattabiram, K. (2016). Understanding Asynchronous Interactions in Full-Stack JavaScrip. *Proceedings of the 38th International Confer-ence on Software Engineering (ICSE'16). 2016. doi:10.1145/2884781.2884864.*

Al-Qerem, A., Alauthman, M., Almomani, A., & Gupta, B. (2020). IoT transaction processing through cooperative concurrency control on fog–cloud computing environment. *Software Computing, 24*, 5695–5711. doi:10.1007/s00500-019-04220-y

Amoretti, M., Grazioli, A., & Sen, V. (2015). A formalized framework for mobile cloud computing,. *Service Oriented Computing and Applications, vol.9, no.3-4, Springer*.

Apache Software Fundation. (2019, 11 4). *Hadoop Project*. Retrieved from https://hadoop.apache.org/

Armbrust, M., Fox, A., Griffith, R., & Joseph, A. (2010). A View of Cloud Computing. *Communications of the ACM, 53*(4), 50-58.

Bailis, P., & Ghodsi, A. (2013). Eventual consistency today: limitations, extensions, and beyond. *ACM Queue, 56*(5), 55-63. Retrieved 6 18, 2022, from https://dl.acm.org/citation.cfm?id=2462076

Bisht, J., & Vampugani, V. (2022). Load and Cost-Aware Min-Min Workflow Scheduling Algorithm for Heterogeneous Resources in Fog, Cloud, and Edge Scenarios. *International Journal of Cloud Applications and Computing (IJCAC), 12*(1), 20. doi:10.4018/IJCAC.2022010105

Bonetta, D., Salucci, L., Marr, S., & Binder, W. (2019). GEMs: shared-memory parallel programming for Node.js. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Challita, S., Paraiso , F., & Merle, P. (2017). Towards Formal-Based Semantic Interoperability in Multi-Clouds: The FCLOUDS Framework,. *IEEE 10th International Conference on Cloud Computing*, (pp. 710-713).

Chelliah, P., & Surianarayanan, C. (2021). Multi-Cloud Adoption Challenges for the Cloud-Native Era: Best Practices and Solution Approaches. *nternational Journal of Cloud Applications and Computing (IJCAC), 11*.

Constela, J. (2019, 11 5). *joseconstela/acio-js*. Retrieved from Github: https://github.com/joseconstela/acio-js

Davis, J., Thekumparampil, A., & Lee, D. (2017). Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. *Proceedings of the Twelfth European Confer-ence on Computer Systems (EuroSys'17)*, doi:10.1145/3064176.3064188.

Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. Retrieved 11 4, 2019, from http://research.google.com/archive/mapreduce.html

ECMA. (n.d.). *Standard ECMA-262, ECMAScript Language Specification*. Retrieved 2 5, 2020, from http://www.ecma-international.org/publications/standards/Ecma-262.htm

Feilhauer, T., & Sobotka, M. (2016). DEF - a programming language agnostic framework and execution environment for the parallel execution of library routines. *Journal of Cloud Computing, 5*(1). doi:DOI: 10.1186/s13677-016-0070-z

Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008). Cloud Computing and Grid Computing 360-Degree Compared. *arXiv: Distributed, Parallel, and Cluster Computing*, 1-10. Retrieved 4 19, 2022, from https://arxiv.org/abs/0901.0131

Guha, A., Saftoiu, C., & Krishnamurthi, S. (2010). The essence of javascript. *arXiv: Programming Languages*, 126-150. Retrieved 1 27, 2020, from http://cs.brown.edu/research/plt/dl/jssem/v1/gsk-essence-javascript-r5.pdf

Holmes, A. (2012). *Hadoop in Practice.* Greenwich, CT, USA: Manning Publications Co.

Kaushik, S., & Gandhi, C. (2020). Capability Based Outsourced Data Access Control with Assured File Deletion and Efficient Revocation with Trust Factor in Cloud Computing. (I. Global, Ed.) *International Journal of Cloud Applications and Computing (IJCAC), 10*(1). doi:10.4018/IJCAC.2020010105

Krishnamurthi, S. (2011). *S5: A Semantics for Today's JavaScript*. Retrieved 2 5, 2020, from The Brown PLT Blog: http://blog.brownplt.org/2011/11/11/s5-javascript-semantics.html

Lai, P., He, Q., Grundy, J., & Chen, F. (2020). Cost-Effective App User Allocation in an Edge Computing Environment. *IEEE Transactions on Cloud Computing,, 10*(3). doi:10.1109/TCC.2020.3001570

Lavoie, E., Hendren, L., Desprez, F., & Miguel, C. (2019). Pando: Personal Volunteer Computing in Browsers. *arXiv*. doi:arXiv:1803.08426

Loring, M. C., Marron, M., & Leijen, D. (2017). Semantics of asynchronous JavaScript. *Sigplan Notices, 52*(11), 51-62. Retrieved 1 27, 2020, from https://microsoft.com/en-us/research/wp-content/uploads/2017/08/nodeasynccontext.pdf

Loulergue, F., Gava, F., Kosmatov , N., & Lemerre, M. (2012). Towards verified cloud computing environments. *International Conference on High Performance Computing & Simulation (HPCS)*, (pp. 91-97).

Madsen, M., Tip, F., & Lhoták, O. (2015). Static Analysis of Event-Driven Node.js JavaScript Applications . *Conference on ObjectOriented Programming, Sys-tems, Languages, and Applications (OOPSLA'15), doi:10.1145/2858965.281*, 505–519.

Peters, T. (n.d.). *PEP 20 – The Zen of Python*. Retrieved 11 4, 2019, from Python Software Foundation: https://www.python.org/dev/peps/pep-0020/

Preguiça, N., Baquero, C., & Shapiro, M. (2018). Conflict-free Replicated Data Types (CRDTs). *arXiv: Distributed, Parallel, and Cluster Computing*. Retrieved 6 18, 2022, from https://arxiv.org/abs/1805.06358

Python. (n.d.). *General Python FAQ*. Retrieved 11 26, 2019, from Docs.python.org: https://docs.python.org/2/faq/general.html#why-is-it-called-python

Python. (n.d.). *Parallel Processing and Multiprocessing in Python*. Retrieved 11 4, 2019, from https://wiki.python.org/moin/ParallelProcessing

Ryza, S., & Wall, T. (2010). MRJS: A JavaScript MapReduce Framework for Web Browsers.

Sahli, H., Bouanaka , C., & Taki Eddine Dib, A. (2014). Towards a Formal Model for Cloud Computing Elasticity. *IEEE 23rd International WETICE Conference*, (pp. 359-364).

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE internet of things journal, 3*(5), 637-646.

Teragni, M. (2022). *HIVE: Shared Distributed Memory Model Based on Cloud and Edge Computing. PhD Thesis.* Argentina: University of La Plata.

Teragni, M., Zabala, G., & Blanco, S. (2018). A cloud powered relaxed heterogeneous distributed shared memory system. *Proceedings of XXIV CACIC (Argentinean Congress on Computer Science) ISBN 978-950-658-472-6.* Buenos Aires, Argentina: UNLP. Retrieved from http://sedici.unlp.edu.ar/handle/10915/73040

Teragni,, M., Moran, R., & Zabala, G. (2020). An Edge Focused Distributed Shared Memory. *Communications in Computer and Information Science, . Springer, Cham. Print ISBN 978-3-030-61217-7 - Online ISBN 978-3-030-61218-4, vol 1291*.

Tuli , K., & Malhotra , M. (2022). A Hybrid Approach for Task Scheduling in the Cloud Environment. *International Journal of Cloud Applications and Computing (IJCAC), 12*(1).

Wang, W., Tornatore, M., & Zhao, Y. (2021). Infrastructure-efficient Virtual-Machine Placement and Workload Assignment in Cooperative Edge-Cloud Computing over Backhaul Networks. *IEEE Transactions on Cloud Computing (early access)*. doi:10.1109/TCC.2021.3107596

White, T. (2012). *Hadoop: The definitive guide.* O'Reilly Media, Inc.

Zakas, N. C. (2016). *Understanding Ecmascript 6: The Definitive Guide for JavaScript Developers*. Retrieved 11 5, 2019, from https://amazon.com/understanding-ecmascript-definitive-javascript-developers/dp/1593277571