A JavaScript Extension Providing Deterministic Temporal Semantics for the Internet of Things

Chadlia Jerad
Edward A. Lee

Acknowledgement

This work was supported in part by the Fulbright Scholar Program, a program of the United States Department of State Bureau of Educational and Cultural Affairs, and by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

A JavaScript Extension Providing Deterministic Temporal Semantics for the Internet of Things

Chadlia Jerad
UC Berkeley & University of Manouba
Berkeley, CA 94720 & ENSI, Manouba, 2010, Tunisia
Email: chadlia.jerad@berkeley.edu & chadlia.jerad@ensi-uma.tn

Edward A. Lee
UC Berkeley
Berkeley, CA 94720
Email: eal@eecs.berkeley.edu

***Abstract*—This paper is about reconciling the highly asynchronous untimed interactions that prevail on the Internet with time-sensitive operations of Things in the Internet of Things (IoT). Specifically, this paper addresses a design pattern that is widely used on the Internet called asynchronous atomic callbacks (AAC). We show that it is possible and practical to endow AACs with temporal semantics that can make system behaviors more repeatable and testable and can make the interactions between cyber services and physical Things safer. We show how a well-defined notion of logical time is compatible with AAC and can be used to endow applications with semantic notions of simultaneity, give them more control over the ordering of events, and enable the specification of real-time behaviors that nevertheless recognize the often long and highly variable latencies introduced in the Internet. We introduce labeled logical clock domains (LLCDs), which permit arbitrary mixtures of synchronized and unsynchronized behaviors and we show how LLCDs can be realized in the JavaScript language, which is widely used in Internet applications. We give a formal model for LLCDs and an execution algorithm that is compatible with standard JavaScript engines.**

I. Introduction

The Internet of Things (IoT) is the class of cyberphysical systems (CPS) that leverage Internet technology for interactions between the physical world and the cyber world. The vision embodied by IoT appeals to the imagination of many—our environment and virtually anything in it will turn "smart" by having otherwise ordinary things be furnished with sensors, actuators, and networking capability, so that we can patch these things together and have them be orchestrated by sophisticated feedback and control mechanisms. As Wegner argued in [?], *interaction* opens up limitless possibilities for Things to harness their environment and compensate for a lack of self-sufficient cleverness. Sensors aside, a connection to the Internet alone allows a Thing to tap into an exceedingly rich environment—unleashing a real potential for making things smarter.

Ensuring safety, reliability, privacy, and security, however, becomes extremely challenging. There is precedent, however, for high confidence systems that use open networks. Today, the world's financial system operates almost entirely electronically and with heavy use of the open Internet. No engineered system is perfect, but the benefits appear to outweigh the risks, and losses due to technical failures and malicious actors are simply factored into the cost of operation. Can cyber-physical systems achieve the same balance, where the benefits of open networks outweigh the costs?

We believe that they can, and this paper represents a step in that direction. Specifically, we show how to attach a rigorous model of time to interactions between IoT devices and between those devices and Internet-hosted services.

A. *Using the Internet for CPS*

Specific Internet technologies of interest for IoT include networking (e.g. IP, TCP, UDP), the worldwide web (e.g. HTTP, HTML5), cloud computing, and programming languages for client and server-side functionality. But these technologies were not designed for interaction with Things. The most egregious mismatch concerns timing. Timing of Internet technologies is strictly a "best effort" affair where the goal is simply to be sufficiently responsive that humans do not loose patience. But when talking about Things, timing can matter quite a lot. It matters when a self-driving car applies the brakes or a robot arm on a factory floor moves.

Some kinds of real-time behavior are not realistically achievable with today's Internet technology. For example, it is unlikely that the feedback control laws governing a self-driving car can be realized in the cloud using RESTful interfaces [?], which rely on HTTP and carry all context state information in each exchange of information. The latency of responses from the cloud-based service is likely to be too high and, more important, too variable.

There are market forces that are driving Internet technology towards more controllable latencies. Interactive services such as distributed gaming and video teleconferencing demand controllable latencies and are difficult to achieve today with high quality and reliability. To address this demand, a recent industry trend is the emergence of time-sensitive networking (TSN) technology. The Time-Sensitive Networking task group of the IEEE 802.1 working group is in the final stages of issuing standards with promising new capabilities that are compatible with open networks.

Some elements of TSN technologies are already widely deployed, though not yet widely used. For example, the IEEE 1588 [?] standard for clock synchronization, which first appeared in 2002 and was substantially revised in 2008, is supported in essentially all Ethernet PHY chips on the market today. It has been deployed in quite a lot of networking gear, but it has not yet risen to the level of a service for application developers except in niche applications [?]. This technology is capable of synchronizing clocks on a local-area network to nanosecond precision, and it is compatible with legacy Ethernet and TCP/IP equipment. The TSN standards that are expected this year, particularly updates of the 802.1AS standard, could be the galvanizing force that will lead to worldwide high-precision clock synchronization. If we combine this technology with edge computers, which can function as gateways that ensure controlled timing on local area networks, then networks with deterministic latencies and reliability delivery that are compatible with the Internet are within reach. We assume in this paper that such networks will become widely deployed.

Even if these market forces are wildly successful, the Internet is unlikely to ever provide the level of determinism needed for many safety-critical, latency-sensitive services. Its most important defining features are openness, which inevitably will increase variability in quality of service, and its geographical distribution, which inevitably increases latencies. Nevertheless, some temporal properties *are* achievable even with today's Internet technology. Some applications can immediately benefit from these properties by becoming more deterministic, more testable, and better able to detect and adapt to failures or degradations of network services. And then as the network infrastructure improves, we will become able to exploit these same temporal properties and more to deliver innovative services.

B. Asynchronous Atomic Callbacks

In the IoT, Things are intrinsically concurrent. Any two physical devices will typically be executing at the same time. Their execution can also be concurrent with cloud-based services, or, more generally, with any software with which they are interacting. Consequently, concurrency looms large in the IoT.

A concurrency pattern that is widely used in the Internet is **asynchronous atomic callbacks** (**AAC**). The AAC pattern, a central feature of the JavaScript programming language, is used extensively in Web programming, both on the server side (using for example Node.js, http://nodejs.org, and Vert.x, http://vertx.io) and on the client side, in browsers. On the server side, it has proven scalable to very large numbers of clients and servers. It has also been used in some other (non-web) applications such as parallel computing (e.g. Active Messages [?]) and embedded systems (e.g. TinyOS [?]).

AAC depends on a functional style of programming, where functions are first-class objects in the language. Functions are invoked asynchronously, typically when some request that has long and/or variable latency has been satisfied. When making such a request, an application will pass a **callback function** as an argument to the service that is to satisfy request. That service then initiates the asynchronous callback when the request has been satisfied. For example, when making a request for data from a URL on the Internet, the application will provide a callback function to be invoked when the data has been retrieved, and that data will be passed as an argument to the callback function. This is better than blocking the application to wait for the response from the Internet because the time to retrieve the response can be quite long and highly variable, and the program may become unresponsive.

Importantly, every such asynchronously invoked function invocation is atomic with respect to every other function invocation; that is, a callback function invocation waits until no other function is being executed before beginning, and the callback function executes to completion before any other function can begin executing. This atomicity distinguishes the AAC concurrency model from interrupt-driven I/O, threads, and many asynchronous remote procedure call mechanisms.

AAC comes with costs, however. First, it becomes essential to write code carefully to consist only of quick, small function invocations. A long-running function will block all callback functions, reducing the responsiveness of applications. Second, AAC accentuates the chaos of asynchrony, where achieving coordinated action can become challenging. For example, if you make multiple requests in sequence to a service, each time passing a callback function, there is no assurance that the callbacks will be invoked in the same order as the requests. Both problems are important for IoT, where heavy computation may be required to analyze sensor data, and coordinated physical actions may be dependent on the order in which things occur.

Because of these limitations, several alternatives mix AAC with other concurrency models. Many JavaScript implementations realize a thread-like mechanism called a Web Worker, which runs tasks in the background concurrently with the main AAC function invocations. Unlike threads, these Web Workers cannot share data with the main application, but rather send messages to the main application, which, if it is listening, will invoke a callback to handle the message. ECMAScript 6, a recent version of JavaScript, enriches AAC with a cooperative multitasking model called "generators," which allows a function to suspend execution at well-defined points, allowing other functions to be invoked while it waits for some event. The Vert.x framework enriches AAC with so-called "verticles" (think "particles"), which can execute in parallel while preserving atomicity. Verticles can interact with one another through a publish-and-subscribe bus or through shared but immutable data structures. Click [?] mixes push and pull interactions with AAC in very interesting ways to create very efficient network routers. Calvin [?], Node-RED (https://nodered.org/), and NoFlo (https://noflojs.org/) use a dataflow concurrency model for interactions between services that are using AAC.

In this paper, we mitigate the limitations of ACC by endowing it a temporal semantics. The temporal semantics includes a notion of simultaneity, enforcement of causal data

```
var x = 0;
function increment() {
  x = x + 1;
}
function decrement() {
  x = x - 2;
}
function observe() {
  console.log(x);
}
setInterval(increment, 1000);
setInterval(decrement, 2000);
setInterval(observe, 4000);
```

Fig. 1. JavaScript example illustrating weak temporal semantics.

dependencies, and a logical time line that can be bound to real time (approximately) to achieve real-time behaviors. Although our work is not limited to JavaScript, it is useful for understanding to refer to a concrete language, so we will give all our examples in JavaScript.

C. Delayed Callbacks

To get timed behavior, most AAC frameworks support **delayed callbacks**. For example, most JavaScript environments provide a `setInterval`(F, T) function, where function F is to be invoked after T milliseconds and then again periodically with intervals of T milliseconds. Of course, the actual time of the function invocations cannot be *exactly* every T milliseconds, since that would require a perfect timekeeper, which does not exist, and it would require that the JavaScript engine be idle at every multiple of T milliseconds, since the AAC model requires that the function invocation wait until the engine is idle. We expect (and get) some jitter in the actual timing of the function invocations. Such jitter is unavoidable in any software platform.

But the situation is worse because the time T actually has very little meaning at all. It is interpreted in the JavaScript language as a suggestive guideline to please invoke the function at some time near the multiples of T milliseconds. When there are multiple delayed callbacks, there are no guarantees on the order of invocation of the callbacks even if the time intervals are identical or related by integer multiples.

Consider the JavaScript program in Fig. 1. This program defines a variable `x` that is shared among all the functions. The code defines functions that increment `x` by one, decrement `x` by two, and observe the value of `x`. It then sets up a periodic callbacks to increment `x` by one every second, decrement `x` by two every two seconds, and observe the value of `x` every four seconds. You might think this would result in the observed value of `x` always being 0, but there is no assurance of this. On version v5.3.0 of Node.js running under MacOS Sierra, the observer sometimes sees 0, sometimes -1, and sometimes 1. Even more insidious, most observations *are* 0, so testing the program may lull the programmer into a false sense of confidence about its behavior. This is not an error in Node.js. It simply reflects the weak temporal semantics of the `setInterval` function.

In the IoT, it is common to build programs that interact periodically with actuators. Since actuators affect the physical world and can do damage, we believe that we need much stronger temporal semantics.

D. Informal Labeled Clock Domains

We achieve a stronger temporal semantics by defining **labeled logical clock domains** (**LLCD**s) within which time values all have specific meanings with respect to one another. To use LLCDs with the program in Fig. 1, we can replace the last three lines with the following:

```
setInterval(increment, 1000, 'A');
setInterval(decrement, 2000, 'A');
setInterval(observe, 4000, 'A');
```

where the third argument is a label for a clock domain. By using the same label in all three calls, we assert that the specified times share the same logical time line. With this specification, the `observe` function `x` will only see value 0.

Under JavaScript principles, the code in Fig. 1 executes atomically. In our temporal semantics, logical time in any clock domain does not elapse during the execution of any such atomic chunks of code. Assume a logical time value of 0 when this code executes. Then the time arguments to the `setInterval` functions will all be relative to this logical time. This program, therefore, expresses that at logical time 1000 (one second), function `increment` should be invoked. At logical time 2000, functions `increment` and `decrement` should be invoked *in that order* and *atomically*. That is, no other function should be invoked between the invocation of `increment` and `decrement`, and hence any observer of variable `x` will only see the combined effects of the two functions. We call this combined invocation of the functions an **atomic action** (**AA**). The two functions, therefore, are logically simultaneous, but also causally ordered. At logical time 4000, functions `increment`, `decrement`, and `observe` will be invoked atomically and in that order.

The atomicity of these combined function invocations makes them logically simultaneous, whereas their ordering ensures that dependency constraints can be specified in the code. The program in Fig. 1, when augmented with clock domain labels as above, expresses that function `observe` should be invoked after `decrement`, which in turn should be invoked after `increment`, at all times when all three are invoked. This is a strong temporal semantic property that enables the construction of much more deterministic programs.

Note that this temporal semantics is independent of the actual physical time at which these functions are invoked. The `setInterval` functions express a *desire* to invoke functions at specified physical times, but as we have observed before, this is impossible to do precisely (or even to define what that means). A *good* implementation of this program will invoke the functions at physical times close to the specified times, but a *correct* implementation only needs to invoke them in the right order and preserve atomicity. The physical time at which they are invoked is a quality metric, not a semantic property.

```
1  var http = require('http');
2  var x = 0;
3  function increment() {
4    x = x + 1;
5  }
6  function decrement() {
7    x = x - 2;
8  }
9  function observe() {
10   console.log(x);
11 }
12 setInterval(increment, 1000, 'A');
13 setInterval(decrement, 2000, 'A');
14
15 function handler(request, response) {
16     setInterval(observe, 1000, 'A');
17     response.statusCode = 200;
18     response.end('Started');
19 };
20 var server = http.createServer(handler);
21 server.listen(8080, 'localhost');
```

Fig. 2. Asynchronous joining of a clock domain.

Suppose that instead of the last three lines we had

```
setInterval(increment, 1000, 'A');
setInterval(decrement, 2000, 'B');
setInterval(observe, 4000, 'A');
```

This code specifies *two* distinct LLCDs. In LLCD with label `'A'`, at multiples of 4000, the `increment` and `observe` functions will be invoked atomically in that order. Hence, the `decrement` function cannot be invoked between them. Since the LLCD labeled `'B'` is created after the one labeled `'A'`, our semantics ensures that at multiples of 2000, `decrement` will be invoked after all relevant label `'A'` invocations have occurred. Therefore, with the above code, the `observe` function will deterministically observe value 2 rather than zero. Hence, use of LLCDs gives the programmer much more control over atomicity and ordering of callback invocations.

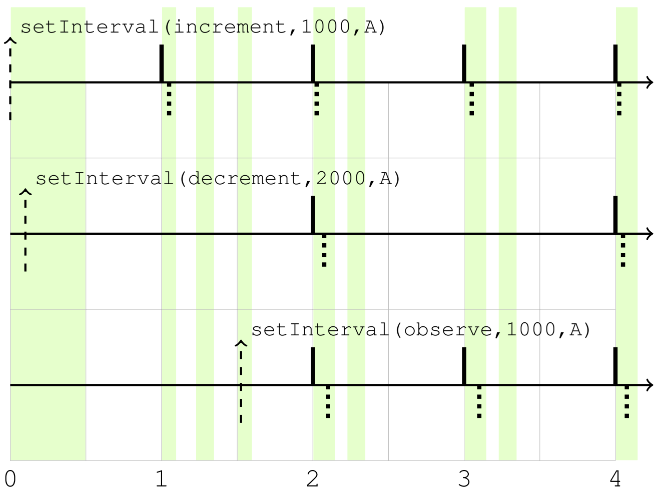
In Internet computing, many things happen asynchronously, often with long latencies or in reaction to external, uncontrollable events. Consider the example shown in Fig. 2. In this example, unlike Fig. 1, the periodic invocation of the `observe` function does not begin until an external HTTP request arrives, for example from a user via a browser. This is representative of IoT applications where a web interface is used to start, stop, or otherwise control some timed service involving Things. The way this program works is that the last two lines release an asynchronous callback, where the function `handler` will be invoked whenever a user points their browser at the URL http://localhost:8080. That `handler`, when executed, on line 16 releases a delayed periodic callback that will cause the function `observe` to be invoked at multiples of 1000 milliseconds, starting 1000 milliseconds from the current time in clock domain `'A'`. Whenever this callback occurs, the current logical time of clock domain `'A'` will be the logical time at which the last AA for `'A'` was executed. Since the clock label is `'A'`, this function will be invoked synchronously with `increment` and `decrement`, but after them. Hence, even though it is started asynchronously, it "joins" a synchronous periodic atomic action. The function `observe` will start executing at a nondeterministic time, but will then deterministically print 0 and 1 alternating.

In our semantics, the joining can occur at any time, but once the joining has occurred, each invocation of `observe` will be simultaneous with every second invocation of `decrement` and will occur after that invocation of `decrement` in a single AA. Moreover, the first invocation of `observe` will occur at an integer multiple of 4000 in logical time, or put another way, logically simultaneously with an even-numbered invocation of `decrement`. By this mechanism, we embrace the asynchrony that is intrinsic in the Internet while nevertheless providing a strong temporal semantics.

Fig. 3 illustrates one possible execution of this program. It shows the logical and physical times at which various operations occur. Assume for this illustration that physical time and logical time have the same units so that we can share the same horizontal axis for both. Assume further that physical time starts at 0 coincident with the first release of a callback.

The vertical bands show AAs. The leftmost band, lasting from physical time 0 to 0.5 (seconds), represents the execution of the main AA, the program in Fig. 2. This program releases two timed callbacks and one asynchronous callback. At physical time 0, line 12 of Fig. 2 executes, releasing the callback function `increment` to be executed at intervals of 1000ms. Since this is the first appearance of clock domain `A`, the time origin and the current time for this clock domain is set to 0. Physically, however, the invocations of `increment` cannot possibly occur at exactly the specified times. The dotted lines dropping from the time line indicate that each of these invocations occurs slightly later than the corresponding multiple of 1000ms in physical time.

The second time line shows line 13 of Fig. 2 executing slightly later in physical time, but since logical time remains at 0, the `decrement` function callback it releases will first be



setInterval(increment,1000,A)
setInterval(decrement,2000,A)
setInterval(observe,1000,A)
0
1
2
3
4

Fig. 3. Time alignment between physical time (dotted) and logical time (solid) for labeled logical clock domains. The vertical bands show atomic actions (AAs).

executed at logical time 2000, synchronous with `increment`, but physically slightly later.

The third time line shows what happens when the asynchronous callback `handler` initiates a new periodic action using the same clock-domain label that is already in use. In this example, at physical time 1.5 (approximately), the callback function is invoked and executes line 16 in Fig. 2. When this happens, logical time for clock domain `'A'` is 1000ms (1 second), so the first invocation of `observe` will occur at logical time 2s, synchronous with `increment` and `decrement` but physically later than them.

E. Outline

The remainder of this paper is organized as follows. Section II gives background on programming languages and models that express temporal properties. The following section introduces our vision for the determinism of temporal semantics. The basic idea and its formalization are described in section IV. Section V describes our implementation in pseudo code. In Section VI, we explain alternative semantic models that we rejected and consider fault management for situations where a program fails to keep up with real time. Finally, we draw conclusions in section VII.

II. Background

Recent years have seen an explosion of innovation in programming languages and programming models. New languages, such as Rust, Scala, Hack, Clojure, Julia, F#, Go, and Dart, and frameworks, such as Apache Spark, Microsoft Orleans [**?**], and Akka, codify programming models that manage parallel computing resources, scalable workloads, and/or long network latencies.

A common thread in these new languages is to embrace elements of functional programming, particularly to make functions first-class objects in the language. This enables use of the AAC design pattern. A common thread in the frameworks (Spark, Orleans, Akka) is support for stream computation based on some variant of actors [**?**], [**?**]. None of these new languages or frameworks, however, has real-time properties.

There is a long and checkered history of programming languages that include timing constructs (Modula [**?**], [**?**], PEARL [**?**], Ada [**?**], Occam [**?**], Real-Time Euclid [**?**], and Erlang [**?**], for example). These improve things by including in the language some of the mechanisms of a real-time operating system (RTOS), which means that a model (a program) is more self-contained. One does not need to combine the semantics of the language with the semantics of a separate and distinct RTOS to interpret the model. Few of these languages survive, however, and all fall short of yielding a deterministic modeling paradigm. In this paper, we focus on a higher-level provisioning of real-time properties, above the language, at the component level and in the concurrency model, specifically in the model of computation.

Another recent trend is to focus on real-time data analytics. The emerging IoT promises a flood of sensor data. Many organizations already are collecting but not effectively using vast amounts of data. Consulting and market research company Gartner calls "dark data" the "information assets that organizations collect, process and store in the course of their regular business activity, but generally fail to use for other purposes." Real-time data analytics implies both timing constraints and streaming data.

Computing on streaming data fundamentally means that you don't have all the data, but you have to deliver results. It differs from standard computation in that the data sets are unbounded, not just big, and you can't do random access on input data, which constrains the types of algorithms you can use. Major research efforts, such as the industry-funded RISELab launched at Berkeley in 2016 (Real-time Intelligent Secure Execution, https://rise.cs.berkeley.edu/), are getting a lot of attention. Examples of algorithmic innovations for real-time streaming data include adaptations of machine learning and optimization algorithms [**?**], [**?**] and adaptations of formal methods [**?**] to operate on streams.

But the timing constraints in real-time data analytics are often not comparable to timing constraints in IoT. For safety-critical cyber-physical systems, timing may effect things that are more important than business opportunities. For safety, stronger semantic models are required. Specifically, we need *semantic* temporal properties (not just quality metrics). Some temporal properties can be assured with high confidence even today, and some will have to wait for advances in networking and computing technology. In either case, clear definitions of those properties is valuable, either to better exploit today's technology or to drive innovation in networking.

III. Temporal Semantics

What do we mean by "temporal semantics"? Fundamentally, what we mean is that certain temporal properties should be elevated from *quality metrics* to *correctness criteria*. In programs, we are accustomed to being able to assume that every execution of a program will (with very high probability) perform exactly the logical functions specified by the program. These logical functions are correctness criteria in that failing to perform them correctly is treated as a fault condition. We believe that certain timing properties should similarly be correctness criteria. What timing properties?

Consider a program that wishes to take two distinct orchestrated actions A and B at 100ms intervals. We can argue that it is physically impossible for these actions to be simultaneous, but that would be missing the point. It may be very useful to have these actions be *logically* simultaneous. What does this mean? It could mean that any observer of these actions will at all times have counted the same number of actions A and B that have occurred. That is, if the observer has seen n A actions, then it has also seen n B actions. Note that this requirement is independent of timing precision and is most certainly physically realizable (with certain assumptions about what an "observer" is). It gives a clean semantic notion to simultaneity. This is an example of a temporal property that can be a correctness criterion.

In addition to a logical notion of simultaneity, a useful temporal semantics should provide for clear ordering of events. First, each component in a system must be able to distinguish past, present, and future. The *state* of a component at a "present" is a summary of the past, and it contains everything the component needs to react to further stimulus in the future. A component changes state as time advances, and every observer of this component should see state changes in the same order.

A more subtle ordering of events arises from the intuitive notion of causality. If one event A causes another B, then every observer should see A ordered before B. Put another way, any observer that can see both A and B that sees an effect from B should also be able to see any effects from A.

These requirements are much easier to meet with an abstract semantic notion of time than one that is more closely tied to physics. But we are interested in cyber-*physical* systems, so we cannot ignore physics. On the physical side of CPS, it is natural to be tempted to adopt a notion of time directly from physics. However, this puts us squarely in a minefield, since time is a poorly understood physical phenomenon [**?**], and models of time differ significantly between Newtonian, quantum, and relativistic physics. However, our goal is not to understand the physics of time, and the usefulness of our models is not determined by how well they match physics. Instead, the usefulness of our models depends on whether we can build physical systems that match the behavior of our models with high confidence [**?**]. This leads to significantly different choices for modeling time.

The most common choice for modeling physical time is Newtonian time. But ironically, Newtonian time proves not so practical for cyber-physical systems. The most obvious reason is that digital computers do not work with real numbers. Computer programs typically approximate real numbers using floating-point numbers, which can create problems. For example, real numbers can be compared for equality (e.g. to define "simultaneity"), but it rarely makes sense to do so for floating point numbers. In fact, some software bug finders, such as Coverity, report equality tests of floating point numbers as bugs. Moreover, addition of floating point numbers is not associative. This precludes any clean notion of simultaneity. Fortunately, these problems have been solved, and we can simply adopt the quantized superdense model of time that has been shown effective for both cyber and physical models [**?**].

In the next section, we give formal definitions of simultaneity and of event ordering for the AAC programming pattern.

IV. Labeled Clock Domains

Our temporal semantics implies simultaneous existence of at least two time lines, logical time(s) and physical time. Unfortunately, natural language did not evolve to give us vocabulary nor idioms for simultaneously talking about multiple time lines. Hence, we have to be very careful with our wording in our explanations, and even then, we risk confusion. Even such common words as "when" can become treacherous. We will do our best.

Logical time is an ideal, a model that imposes constraints on execution order for software. Physical time, or wall-clock time, is the real time that is perceived by humans or by machines. For this time, we assume a Newtonian model, that all measurements of this time are flawed, and that no action can be precisely placed on this time line. Nevertheless, physical time is useful because the degree to which logical time aligns with it can form a quality metric. A real-time system will strive for close concurrence between logical and physical times, at least for some of its actions.

Physical time and logical time do not progress at the same rate. Logical time will advance in a computational world, in discrete jumps. Physical time, according to Newton's model, advances smoothly and uniformly. We assume that the execution platform has a real-time clock accessible to the program so that the program can at any time obtain an estimate of the current physical time. We denote physical times or estimates of physical times with lower case variable names, such as t, and logical times with upper case, such as T. Physical time is a real number and it is not important what the origin is.

Physical times t are real numbers $t \in \mathbb{R}$, and logical times T are natural numbers $T \in \mathbb{N}$. We assume an **interpretation mapping** between these where a logical time T is assumed to correspond to a physical time $t = \omega(T)$ and physical time t is assumed to correspond to a logical time $T = \Omega(t)$. The function $\omega\colon \mathbb{N} \to \mathbb{R}$ is one-to-one, and the function $\Omega\colon \mathbb{R} \to \mathbb{N}$ is onto.

We focus three key mechanisms:

- A **delayed periodic callback**, realized in JavaScript as a function `setInterval`(F, T, L). This function has three arguments, a function F to invoke in the future, a logical time value $T \in \mathbb{N}$, and a clock domain label L. This function initiates periodic invocation of the specified function, where the first invocation occurs T logical time units in the future on the logical time line with label L. If no label L is given, then a new **anonymous clock domain** is created and its origin is set to $\Omega(t)$, where t is the current physical time.
- A **delayed one-time callback**, realized in JavaScript as a function `setTimeout`(F, T, L). The arguments are the same, but instead of setting up a periodic function invocation, it sets up a one-time invocation of the specified callback function F to occur T logical time units in the future on the logical time line with label L (or on an anonymous time line).
- An **asynchronous callback**, realized in JavaScript by a variety of functions that take at least one argument F, a function to be invoked in the future when some condition in the environment is satisfied. The web server message handler in Fig. 2 is an example.

Each of these three mechanisms is said to **release** a callback. The release occurs at some physical time $t \in \mathbb{R}$ called the **release time**, and the execution of function F starts at some later physical time(s) $t' > t$ called the **callback execution time**. In all three cases, the callback function F, when executed, can release further callbacks. Those releases will occur at some physical time $t'' > t'$.

We assume a sequential execution model where callbacks are released in a well-defined order. We further assume that all operations take time, and hence no two callbacks can have the same release time. Note however that, in practice, if release times are obtained from a real-time clock that has some granularity defined by the underlying operating system, then these release time may in fact be equal. Many operating systems, for example, provide estimates of real time as integer multiples of one millisecond, and these estimates are often more coarsely quantized, for example only having values that are integer multiples of ten milliseconds. Based on these estimates, therefore, two callbacks may appear to have the same release times. We nevertheless assume that the execution engine can keep track of the *order* in which callbacks are released, which is well defined because of the sequential execution model. We will represent this order by considering *idealized* release times, unquantized real numbers. That is, if one callback has release time t and another has release time t', then $t \neq t'$ and if $t < t'$, then the first callback was released first. Hence, callback releases are totally ordered by release time.

Execution of a program is a sequence of **atomic actions** (**AA**s) as in JavaScript. An AA may release several callbacks. These releases may specify clock domain labels that have not been seen before in any previous release. In this case, a **new clock domain** is created with label L provided as an argument to function that releases the callback. A delayed callback may also be released with no specified clock domain label, in which case a new **anonymous clock domain** is created.

We assign to a new clock domain with label L a **physical time origin** t_L, which is the release time of the first release in this clock domain, and a **logical time origin** $T_L = \Omega(t_L)$. For convenience, we define two partial functions $o\colon \mathcal{L} \rightharpoonup \mathbb{R}$ and $O\colon \mathcal{L} \rightharpoonup \mathbb{N}$, where $\mathcal{L}$ is the set of labels, to be

$$o(L) = t_L \tag{1}$$
$$O(L) = T_L = \Omega(t_L). \tag{2}$$

The function o is one-to-one, but because of time quantization, O is not. Two clock domains could have the same logical time origin.

Once a clock domain has been created, logical time in that clock domain does not advance until after the AA that creates the clock domain completes. Hence, within an AA that creates a new clock domain, logical time remains at the logical time origin for that clock domain. In fact, it does not advance until a delayed callback in that clock domain is executed. At that point, logical time for that clock domain advances by the amount of the delay in the delayed callback. If multiple clock domains are created within an AA, then their physical time origins $o(L)$ are assured to be different, but not their logical time origins $O(L)$.

In a typical execution environment, execution begins with a **main** AA, e.g. the execution of a top-level program like that in Figs. 1 and 2. If no callbacks are released during that execution, then the program terminates. Such a program is not interesting to us because it has no temporal behavior. Hence, we assume that the main AA releases some callbacks.

After the main AA completes, each subsequent AA consists of a sequence of one or more invocations of callback functions. These AAs may be triggered by the passage of time or by some asynchronous action in the environment, such as the arrival of an HTTP request. An AA is triggered by the passage of time if either a delayed periodic callback or a delayed one-time callback has been released and the logical time at which that callback should be invoked is smallest logical time of all pending callbacks. If more than one callback is to be invoked at the same logical time for a clock domain, then all those callbacks are invoked atomically. When that AA is executed, logical time for the clock domain that triggered it is set to the logical time of the callback(s) that triggered it.

For example, if the program in Fig. 1 is modified to use clock domain A in all calls to `setInterval`, then its execution proceeds as follows. When the first `setInterval` (line 11) is executed, the logical clock domain A is created and its origin is set to t_A and T_A, physical and logical respectively. Logical time remains at T_A until after the entire script has been executed. Once the main script has been executed, there is nothing more to do, so the execution engine examines the pending delayed callbacks and finds that the smallest callback time is $T_A + 1000$. It then sets a real-time timer to wait until physical advances to approximately $\omega(T_A{+}1000)$, unless that real time has already passed, in which case it proceeds directly to execute the callback(s) in an AA. If it sets a timer, then it will execute the callback only after that timer expires. For the example in the figure, at approximately physical time $\omega(T_A{+}1000)$, the logical time of clock domain A will be set to $T_A + 1000$ and `increment` will be executed. The execution engine then again has nothing to do, so it again waits, this time for another 1000 milliseconds. At approximately physical time $\omega(T_A + 2000)$, logical time will be set to $T_A + 2000$ and `increment` and `decrement` will execute in sequence within an AA. At approximately physical time $\omega(T_A + 3000)$, `increment` executes again. At approximately physical time $\omega(T_A + 4000)$, `increment`, `decrement`, and `observe` will execute in sequence within an AA. Then the whole pattern repeats.

If instead the last three lines are

```
setInterval(increment, 1000, 'A');
setInterval(decrement, 2000, 'B');
setInterval(observe, 4000, 'A');
```

then the pattern is similar, except that at physical times near $\omega(T_A + m2000)$ for $m \in \mathbb{N}$, $m \geq 1$, an AA will execute `increment` (followed by `observe` if m is even). At physical times near $\omega(T_B + m2000)$, `decrement` will be invoked. We are guaranteed that $T_B \geq T_A$, but since they can be equal, `decrement` may be invoked at the same logical time as `increment` followed by `observe`, but it is guaranteed to be invoked after them because they execute atomically. All of these executions are guaranteed to occur in exactly this order regardless of how much physical time passes.

The program in Fig. 2 is similar except that the asynchronous callback `handler` may be invoked between any two AAs, most likely while the execution engine is waiting for time to elapse. In this case, during the execution of the `handler` callback, logical time T_A for clock domain is A has whatever value was last set by a delayed callback, which will be $T_A + m1000)$ for $m \in \mathbb{N}$, $m \geq 0$. Hence, the interpretation of line 16,

```
setInterval(observe, 1000, 'A');
```

is that `observe` should start being invoked at the next multiple of 1000. Moreover, `observe` will join the invocations of `increment` (and if m is even, `decrement`) in the same AA.

Our temporal semantics has the following key properties:

1) New clock domains may be created at any time during execution. Their origin will be set to a logical time that approximates current physical time.
2) For each clock domain, all callbacks that are scheduled to be invoked at the same logical time will be invoked in the same AA. This makes them logically simultaneous, since an AA is atomic.
3) Those callbacks will be invoked within the AA in the same order in which the callbacks were released. This means that precedence constraints can be expressed and are respected.
4) During the execution of an AA, all clock domains are frozen. Logical time does not advance. Hence, each AA logically executes instantaneously.
5) When an AA is triggered by the passage of time, exactly one of the logical clocks, the one corresponding to the trigger, is advanced so that its logical time matches the time of the delayed periodic or one-time callback that triggered it. All other clock domains remain unchanged. Hence, logical time will advance non-uniformly across clock domains.
6) When an AA is triggered by an asynchronous callback, logical clocks are left unchanged from whatever value they had before the AA is executed.

There are exactly two sources of potential nondeterminism in this model. First, when clock domains are created, their origin is nondeterministic because it depends on the execution time of code preceding it, and that execution time is not under the control of the program. Second, an asynchronous callback can occur between any two AAs and can release delayed callbacks. If a program has only one clock domain and no asynchronous callbacks, then it is completely deterministic.

The following definition summarizes our notation:

Definition 1 A **delayed callback** is a tuple $C = (F, T, L, t, p)$ where

- F is the callback function,
- T is the delay,
- L is the clock domain label,
- t is the release time, and
- p is a boolean that is true if the callback is periodic and false if it is one time.

Asynchronous callbacks can be similarly defined:

Definition 2 An **asynchronous callback** is a tuple $A = (F, t, \tau)$ where

- F is the callback function,
- t is the release time, and
- τ is the physical time at which invocation is requested by the environment.

As noted in this definition, the environment requests invocation of an asynchronous callback at some physical time τ. At that time, the main loop of the execution engine may be idle, or it may be executing an AA. If it is idle, then it can immediately begin executing the callback in a new AA. If it is not idle, then it must queue an AA to execute the callback. In this paper, we assume that queued AAs are executed in FIFO order whenever the currently executing AA completes. In principle, more elaborate prioritized ordering could be used, but this is beyond the scope of this paper.

V. Implementation

A data structure **Callback** has fields:

- F: Function to invoke.
- T: The logical time of the next invocation of this function.
- L: The logical clock label or null for an anonymous clock.
- P: The period with which to invoke F, or -1 for one time.

We also assume a data structure **Clocks** that is a hashtable indexed by clock labels with value equal to the current logical time for that clock.

A function **execute**(C) executes a list C of Callbacks as follows. It assumes (and checks) that every Callback on the list has the same logical time T. It sets the current logical time for each clock labels L on the list to T, Clocks(L) = T, just prior to executing the first callback on the list with label L. It executes F for the first Callback in the list followed by all the Fs for subsequent Callbacks in the list that have the same (non-null) logical clock label. As it executes these functions, it removes each of these functions from the list. It then repeats this process on second (now first) F in the list. It continues until the list is empty.

A **CallbackQ** class is a priority queue sorted by logical time with special support for periodic actions. Specifically, when you put a Callback on this queue that has period $P \geq 0$, then this callback is (logically) inserted into the queue an infinite number of times, once for every invocation. This class supports the following operations:

- isEmpty(): Return true if the queue is empty.
- put(c): Put the specified Callback c on the queue.
- nextTime(): Return the next earliest time on the queue.
- poll(): Return and remove a list of Callbacks scheduled to be executed at nextTime().

Note that this class needs to be fairly sophisticated to take into account periodic actions. It has to be carefully implemented to preserve the order in our semantics and to adjust the T field of each periodic Callback before returning it. Our implementation has one global instance of CallbackQ.

A function **currentTime**(L) returns the current time for the clock label L, or if no clock label is given, then it returns $\Omega(t)$, where t is current physical time.

We assume a standard main loop like that found in JavaScript that executes ready tasks in FIFO order. Asynchronous callbacks go directly into this event loop's FIFO queue when they become ready for execution.

We further assume a **timer**(F, t) function that sets a real-time timer to execute F as close as possible to time t. At that time, it will put F on the queue for the main loop. This function returns a handle h that can be used to cancel the timer if it has not expired using a **cancel**(h) function. Our execution engine never has more than one pending such timer and it uses a global variable H that is the handle to that timer. This variable is null when there is no pending timer. Another global variable W records the logical time at which the one pending timer will expire, if there is one.

Execution begins by initializing the global variables H = null, $W = \infty$, an empty Clocks, and an empty CallbackQ. It then executes the application's main program, which may release callbacks. It then invokes a function NEXT(), defined below, which executes any callbacks for which it is time and then sets a timer to wait until the time at which the next anticipated timed callback will be invoked.

```
procedure NEXT
    if CallbackQ.isEmpty() then
        $W \leftarrow \infty$                    ▷ No pending timed callbacks
        return
    end if
    $T \leftarrow$ CallbackQ.nextTime()          ▷ Next logical time
    $t \leftarrow$ currentTime()                 ▷ Current physical time
    while $t \geq T$ do                          ▷ Anything ready to execute?
        $C \leftarrow$ CallbackQ.poll()          ▷ Get list of callbacks
        execute($C$)                             ▷ Execute callbacks
        $T \leftarrow$ CallbackQ.nextTime()
        $t \leftarrow$ currentTime()
    end while
    $H \leftarrow$ timer(NEXT, $T$)              ▷ Start a timer
end procedure
```

The application's main program and any of its callbacks may call `setTimeout`, which is defined as follows:

```
procedure SETTIMEOUT($F, T, L$)
    if ! $L$ then                                ▷ No label, anonymous clock
        $L \leftarrow$ some unique label
    end if
    if ! Clocks($L$) then                        ▷ Clock does not exist
        Clocks($L$) $\leftarrow$ currentTime()
    end if
    $T_n \leftarrow$ Clocks($L$) + $T$           ▷ Next execution time.
    $c \leftarrow$ new Callback($F, T_n, L, -1$)
    CallbackQ.put($c$)
    if $T_n \leq W$ then                         ▷ Time is earlier than timer.
        cancel($H$)
        $W \leftarrow T_n$
        $H \leftarrow$ timer(NEXT, $T$)
    end if
end procedure
```

The function `setInterval` is similarly defined.

To verify, we have implemented the formal model using Real-Time Maude [?]. The Appendix shows key parts of this model, which we have used to check some properties of the semantics, for example that callbacks released at the same logical time with the same clock label and delay execute in the same sequence as their releases.

VI. DISCUSSION

Our semantics significantly improves the determinism of programs compared to the standard best-effort timing and nondeterministic ordering of functions like `setTimeout` and `setInterval` in JavaScript. There remain two sources of nondeterminism, both of which are useful for dealing with the inevitable vagaries of the Internet. The first is that when a new clock is created, its origin is set to an estimate of current physical time. This value will depend on execution times prior to this occurrence, but it means that an application can be very dynamic, starting a family of predictable behaviors at any time. The second is that asynchronous callbacks will be executed nondeterministically between timed atomic executions. Other than these two mechanisms, the order of execution of all functions is completely defined by the program.

Note that this order is respected even the execution of the program completely fails to keep up with real time. The `timer` function described in the previous section will be used only when the execution is ahead of real time and performing any further execution will result in executing functions too early. But we cannot do magic. There is no way to keep up with real time if the computation load is too heavy.

For this reason, safety-critical applications will need to perform sanity checks when callbacks are invoked. A recommended practice is to use the `currentTime` function to compare logical time to real time, and when the difference exceeds some threshold, switch the application into a fault handling mode. No matter what semantics one uses, faults are always possible, in which case executions may not behave as intended. Programs should be written to expect faults to occur and to handle them accordingly. Our mechanism at least enables detection of such faults, albeit possibly late. If, for example, some callback function performs a very long execution, then our recommended strategy will discover the fault only after it has finished. To discover it sooner, we would have to sacrifice the key atomicity property of AAC and allow preemption. This is a big price to pay because it probably requires the fault handler to abort the entire execution of the application program since its integrity can no longer be guaranteed. Nevertheless, this may be necessary for some safety-critical applications.

We considered and rejected several alternative designs before settling on the semantics we have presented. One candidate that initially seemed very appealing was that the meaning of T in a release like setTimeout(F,T,A) would be to invoke the function F at the earliest logical time $O(A) + mT$, where m is an integer, after the current time. Instead, we interpret T to mean to invoke F at Clocks(L) + T, relative to the current time of the clock with label L. The alternative design is intuitive for simple periodic tasks and it enables quick recovery in the event that some task takes more than the expected time so that physical time passes logical time. However, we realized that a sequence of releases like the following becomes very difficult to control under the alternative design:

```
setTimeout(F, 3, A);
setTimeout(G, 4, A);
```

It will be very hard to know whether F will be invoked before G and how much logical time will elapse between their invocations. This will depend on exactly when the above commands are issued. Only at the very start of the execution of the program will the order of invocation be easy to predict, so this semantics proves problematic for long running programs.

VII. Conclusion

The Internet is highly asynchronous while physical Things are highly timing dependent. If the IoT is to be applied to important and safety-critical applications, these contradictions need to be reconciled. We offer in this paper a modest addition the commonly used AAC model of computation that has proved so effective in the Internet. Our modest addition endows timed callbacks with a rigorous semantics, making it much easier to control the order and atomicity of events.

We have integrated our proposed semantics with an actor-oriented discrete-event programming model for IoT called "accessors" (see http://accessors.org and [?]). This integration combines AAC a highly concurrent, parallelizable, and scalable actor model augmented with time stamps. But this integration will need to be the topic of another paper.

There are enormous opportunities for further work. First, in this paper, we have not really addressed language design. We suspect that programming languages that directly embrace temporal properties may be much more effective than those that just append timing as an afterthought using delayed callback functions. Second, we have not addressed how the alignment between logical time and physical time might be enforced. Third, there are opportunities for distributed real-time execution that can be combined with our timed AACs, using for example techniques like Ptides [?], [?].

Acknowledgment

This work was supported in part by the Fulbright Scholar Program, a program of the United States Department of State Bureau of Educational and Cultural Affairs, and by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Appendix

We first briefly introduce Real-Time Maude. In the algebraic specification community, data types are called "sorts" and they are introduced in Real-Time Maude using the keyword sort, while subsort is used for sub-typing. The keyword op is used to define constructors and/or modifiers on top of sorts. The behavior of operations is described by means of conditional and unconditional equations (respectively ceq and eq). Rules (introduced by the keywords crl, for conditional, and rl, for unconditional) remove the symmetry of the equations. They are interpreted as local concurrent transitions. In object-oriented modules, a system state is usually described as a multiset of objects and messages. Classes are introduced using the keyword class, together with the attributes list with their respective sorts.

We start by giving the definition of CallbackQ. It is declared as follows:

```
sort CallBackQ CallBack .
subsorts CallBack < CallBackQ < NEConfiguration .
op _::_ : CallBackQ CallBackQ
  -> CallBackQ [assoc ctor] .
op empty : -> CallBackQ [ctor] .
op put : CallBack CallBackQ -> CallBackQ .
```

Labeled logical clock domains are defined as objects of the class LLCD. The sort Label is defined as a subsort of Oid, Object Identifier.

```
sort Label .
subsort Label < Oid .
class LLCD | origine : Time, Origine : Time,
  CurrentTime : Time .
```

Physical time is modeled by the class clock. The attribute currentTime records the value of the current time, while the timer attribute saves the timer value. Physical time elapsing is modeled by the rule tick. This rule transforms the system from one state to another, where it increments the current time and enables the execution of instructions (tick(false)).

```
class clock | currentTime : Time, timer : Time .
op tick : Bool -> NEConfiguration .
rl [tick] :
  {< physTime : clock | currentTime : ct,
    timer : tt >  tick(true) C:Configuration}
 =>  {< physTime : clock | currentTime : ct + R,
      timer : tt > tick(false) C:Configuration}
in time R [nonexec] .

sort callbackFunction .
class delayedCallback | task : callbackFunction ,
  delay : Time , llcd : Oid,
  release : Time, Release : Time,
  periodic : Bool .
```

The functions setTimeout and setInterval are Messages (using the keyword msg) with the following arguments: the physical time of the release, the identifier for the callback function, the delay, and the label of the clock domain.

```
msg setInterval :
  Time Oid callbackFunction Time Label -> Msg .
msg setTimeout :
  Time Oid callbackFunction Time Label -> Msg .
```

We give below the example of a `setInterval` message consumption, which creates a periodic delayed callback with an existing label. The message will be consumed only if the current physical time is greater to or equal to the message release. Consequently, the callback queue is updated. Furthermore, if the callback needs to execute before the current set timer, then the timer is updated.

```
ceq
  < physTime : clock | currentTime : ct',
    timer : tt >
  tick(false) CBQ:CallBackQ
  < l : LLCD | origine : o, Origine : O,
    CurrentTime : CT >
  setInterval(ct, oid, cbf, dly, l)
=
  if tt lt (CT plus dly) then
    < physTime : clock | currentTime : ct',
      timer : tt >  tick(true)
    < l : LLCD | origine : o, Origine : O,
      CurrentTime : CT >
    < oid : delayedCallback | task : cbf,
      delay : dly, llcd : l,
      release : ct', Release : CT plus dly,
    periodic : true >
    put(aac(oid, o, ct', CT plus dly),
      CBQ:CallBackQ)
  else
    < physTime : clock | currentTime : ct',
      timer : CT plus dly > tick(true)
    < l : LLCD | origine : o, Origine : O,
      CurrentTime : CT >
    < oid : delayedCallback | task : cbf ,
      delay : dly, llcd : l,
      release : ct', Release : CT plus dly,
    periodic : true >
    put(aac(oid, o, ct', CT plus dly),
      CBQ:CallBackQ)
 fi
if ct' ge ct .
```

The following conditional equation describes a callback execution. Indeed, if the current physical time is greater than or equal to the timer, and the timer is greater than or equal to the release time of the callback at the top of the queue, then the callback is pulled, the execution is recorded, and the clock domain current time is updated as well as the timer.

```
ceq
  < physTime : clock | currentTime : ct, timer : tt >
  (aac(oid, o, r, R) :: aac(oid', o', r', R')
    :: CBQ:CallBackQ)
  < oid : delayedCallback | task : cbf, llcd : l,
    release : r, Release : R,
    delay : dly, periodic : true >
  < l : LLCD | origine : o, Origine : O,
    CurrentTime : ct' >
  OL:OrderedList tick(false)
=
  < physTime : clock | currentTime : ct, timer : R' >
  put (aac(oid, o, r, R plus dly),
    (aac(oid', o', r', R') :: CBQ:CallBackQ))
  < oid : delayedCallback | task : cbf, llcd : l,
    release : r, Release : R plus dly,
    delay : dly, periodic : true >
  < l : LLCD | origine : o, Origine : O,
    CurrentTime : R >
  (OL:OrderedList ++ (oid @L R @P ct)) tick(true)
if (R le tt) and (tt le ct) .
```