A Component Architecture for the Internet of Things

Christopher Brooks*, Chadlia Jerad†*, Hokeun Kim*, Edward A. Lee*,
Marten Lohstroh*, Victor Nouvellet‡*, Beth Osyk*, and Matt Weber*
*University of California, Berkeley †University of Manouba ‡Institut National des Sciences Appliquées de Lyon
Email: {cxh, chadlia.jerad, hokeunkim, eal, marten, victor.nouvellet, beth, matt.weber}@eecs.berkeley.edu

***Abstract*—We describe a component-based software architecture for the Internet of Things in which proxies for Things and services that we call "accessors" interact with one another under a concurrent, time-stamped, discrete-event (DE) semantics. These proxies are analogous to web pages, which proxy a cloud-based service such as a bank, but instead of being designed to interface those services with humans, accessors are designed to interface services and Things with other services and Things. A deterministic DE semantics is combined with a widely used pattern for handling network interactions that we call "asynchronous atomic callbacks" (AAC). AAC enables many concurrent pending requests to be active at once without blocking and without the treacherous concurrency pitfalls of threads. In effect, our architecture combines AAC with actors where the actor model has been endowed with a temporal semantics. We show how this architecture can leverage the previously reported Secure Swarm Toolkit (SST) to achieve state-of-the-art authentication, authorization, and encryption of interactions across networks.**

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. Supported by Wegner's argument that *interaction* is more powerful than algorithms [45], Lohstroh and Lee [30] point out that interaction indeed 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 of systems that rely on open networks is 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 cyberphysical systems achieve the same balance, where the benefits of open networks outweigh the costs?

The web focuses on the interaction between people and information or services hosted on servers or in the cloud. The IoT, on the other hand, will emphasize interactions between Things and Things, Things and cloud services, and Things and people, rather than people with cloud services. In this paper, we describe a design pattern that we call accessors[1], where an accessor is like a web page for a Thing or service, but instead of being designed for humans to interact with it, it is designed for other Things and services to interact with it.

Accessors are based on an adapted actor model, where an accessor is a parameterized actor with input ports and output ports through which timed events stream. The accessor serves as a proxy for a Thing or service that may be local or remote. This proxy is analogous to the proxying of a service that occurs in your web browser when you download HTML and JavaScript from a web server. The browser instantiates a proxy for a remote service, such as your bank. The proxy runs on the local host, your computer, but interacts with a remote service using mechanisms that are largely invisible and irrelevant to you, the user of the proxy.

The input ports and parameters of an accessor are analogous to form boxes on web pages, and output ports are analogous to rendered pages. But form boxes are designed for human input, and rendered pages for human consumption. The input and output ports of accessors are designed for interaction with other Things and services. The accessor itself provides functionality analogous to the scripts that a web page runs in the browser, which communicate in proprietary ways with the (possibly remote) Thing or service. A critical part of our model is that the host environment that executes the accessor must have standardized capabilities, just as browsers today (mostly) support common languages (HTML and JavaScript) with a common set of bindings that the program can use (the Window object and the XMLHttpRequest object, for example). This enables web designers to design a proxy that will work in any browser. Analogously, accessors are designed to execute in a variety of hosts, ranging from deeply embedded processors to cloud servers.

We also describe a design environment called CapeCode[2] that can be used to compose accessors to create services which can then further be proxied by accessors. CapeCode is, in

[1] http://accessors.org
[2] http://capecode.org

effect, a computer-aided design tool for IoT applications. It facilitates debugging and design-space exploration by providing a friendly graphical environment that includes all of the analytical tools of Ptolemy II, on which it is based.

Our focus in this paper will be to show how the particular actor model by which accessors interact with one another, a timed discrete-event model, matches well the requirements of IoT applications. It provides a measure of determinism that helps to counter the chaos of unpredictable latencies and unreliable networks that is intrinsic to applications that are distributed on the open Internet. In particular, its deterministic semantics enables well-defined test cases, rigorous specifications, and reliable error checking. Deterministic semantics means that there is a well-defined notion of "correct behavior," and that behavior is repeatable. Our semantics also enables a more deterministic use of timing by replacing best-effort timeouts with a model of time that has a semantic notion of simultaneity and well-defined ordering of events. Finally, we show how accessors can leverage edge computing to improve security, privacy, predictability, and robustness to network outages.

II. Motivating Example

Consider a device, such as a tablet, smart phone, or augmented reality (AR) goggles, that has a camera, an interactive screen, an audio speaker, and a microphone. Consider an app on this device that invokes an image processing service to recognize Things seen by its camera and then overlays the Things in the display with current sensor data and any interactive controls provided by the Thing. Fig.1 shows such an overlay in what might be a mechanical room of the future. Consider further that the device can respond to voice commands to scroll through a suite of recognized Things in the field of view or turn on and off the overlay display. Such an app would be useful, for example, for factory floor inspection, equipment maintenance, configuring smart conference rooms, and myriad other applications.

All of the technology exists today to build such an app, and indeed similar systems are familiar to those working in the field of augmented reality. But anyone familiar with the technologies involved will realize that the complexity is considerable and that the result is likely to be brittle. Very likely, a realization today will be a stovepiped solution, where every component is entirely under the control of a single vendor. But making something that is open, for example something that is able to discover devices from new vendors in the local environment or to leverage machine learning that integrates data from outside sources will be extremely challenging.

Fig. 2 shows a prototype of such a system built using accessors in the CapeCode design environment. Our prototype constructs the overlay display shown in Fig. 1, a user interface for a Thing detected in the local environment. The app tracks movement as the camera pans over the scene. As motivation for subsequent discussion, we walk you through what the components of this prototype do. Hopefully, the reader will

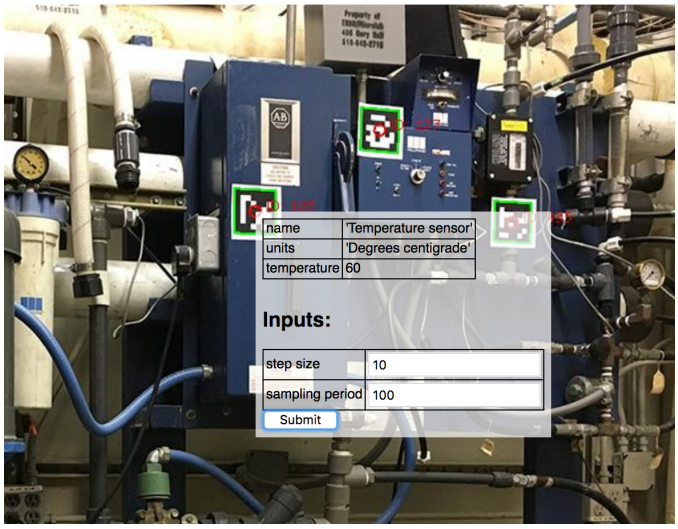

Fig. 1. Augmented reality display.

see readily that this is, in fact, a prime example of the use of reusable components in a platform-based design. It will also lend insight into the reasons for our choice of using a discrete-event semantics to govern the interactions between accessors.

First, the three boxes in the figure whose icons contain "JS" encapsulate small, simple scripts that are specific to this app. These are the only components in the design that are not intended to be reusable. They are specific to this app. In our prototype, they are written in a few lines of JavaScript using the same framework used to create the reusable components, which we call "accessors." All remaining components, those with icons not containing "JS," are accessors and are intended to be reusable.

Beginning at the left of the top row of icons, the Camera accessor provides access to a hardware device: a camera, connected to the host computer that runs this app. That accessor outputs a stream of images and has parameters for controlling the frame rate, resolution, and selection of camera, in case there is more than one camera on the host. It can alternatively provide access to a network-connected camera, in which case the structure of the app does not change at all. Only the parameters change.

To the upper right, receiving the stream of images, is an ObjectRecognizer accessor. This can use any of a variety of technologies to recognize Things in images. In our prototype, we simply assume that Things are labeled with AR tags, which are like simplified QR codes that are easier for cameras to recognize at a distance. Three AR tags can be seen in Fig. 1. More elaborate technologies could identify objects by their visual appearance, with the help of a challenge-response interaction, leveraging indoor localization and device telemetry, or with the help of a discovery service such as the Summon app [46].

The script labeled "TagToAccessor" receives from the ObjectRecognizer an array of zero or more IDs for AR tags found

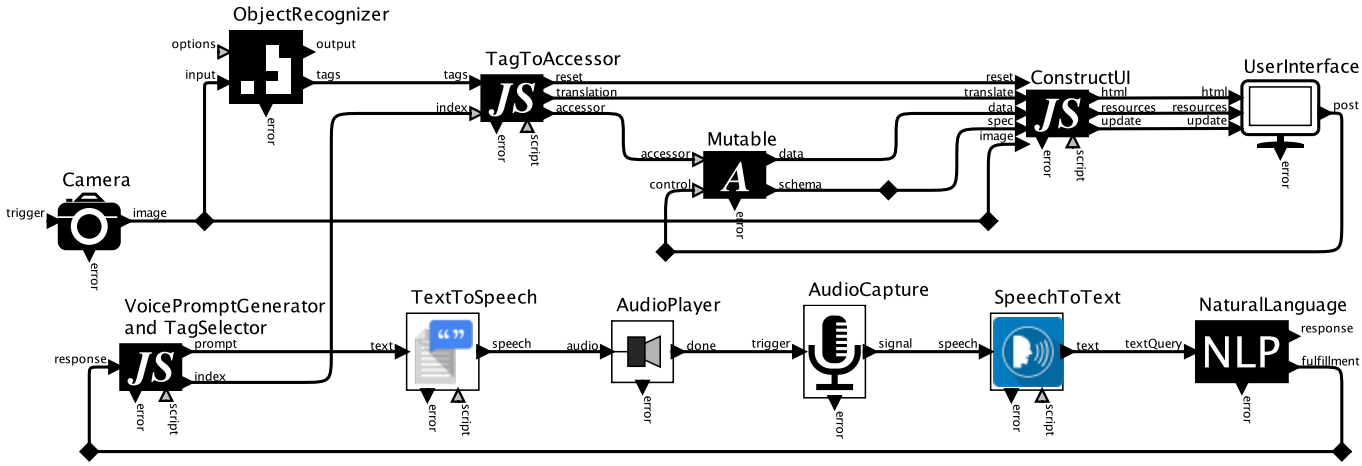

Fig. 2. Augmented reality application for interacting with sensors and actuators. This composition of accessors generates the interface shown in Fig. 1

in each image frame along with the X-Y position of the tag in the field of view. Based on the "index" input (which comes from the lower loop of icons, discussed below), it selects one of these tags and looks up an accessor for a Thing associated with the tag's ID. In the simplest case, TagToAccessor could perform a table lookup to match a tag with an accessor for a Thing. A more advanced TagToAccessor implementation would use a location-based discovery service to dynamically obtain the tag to accessor matching for the user's current environment and update the matching when new AR tags are deployed. The accessor itself could come from a web service or from a local edge computer that is aware of devices in the local environment. TagToAccessor feeds that accessor to an accessor labeled "Mutable."

Mutable is perhaps the most interesting component here. It accepts as input an accessor, which it checks for compatibility with its ports, and if it is compatible, reifies the accessor and delegates handling of streaming inputs and outputs to it. The input to Mutable is the source code for an accessor that it instantiates and begins executing. Note that since this source code can be downloaded at the time of instantiation, it can be assumed to be up-to-date and compatible with the current version of the Thing's API, which itself may be periodically updated, for example, to patch for security vulnerabilities.

The Mutable accessor can be seen as an abstract interface specification for candidate accessors. The reified accessor effectively replaces the Mutable accessor, taking its place in the block diagram. If later a new accessor appears, it will be reified and will replace the previously reified accessor, which will be shut down. In this case, Mutable expects the provided accessor to have an input port named "control" and two output ports, "data" and "schema." All three ports are typed to handle JSON formatted data. It can also accept accessors that partially match, for example omitting the control input port, as we explain below. The schema output provides the app with a specification of what is expected on the control input port. That specification is used by the ConstructUI script to build an HTML table with input boxes as shown in Fig. 1. The UserInterface accessor uses any resource on the local host that can display HTML5. On a laptop computer this could be a browser, whereas on a mobile device it could be a service provided by the operating system or app development framework such as Apache Cordova.

The reified accessor in Mutable is a proxy running on the local host that represents the Thing identified in the camera's field of view. Depending on the capabilities of the host, the reified accessor may communicate with the Thing via Bluetooth, Wi-Fi, ZigBee, or any other technology supported by both the host and the Thing. The actual communication mechanism and protocols can be proprietary to the Thing and its accessor, just as the details of the communication between a web page and a bank are specific to the bank.

The design pattern here is similar to what has proven so effective in the web. To make browsers talk to banks, for example, the world's banks could have gotten together and established a standard messaging protocol that would include specifications for messages to view balances, make payments, etc. But this is not what happened. Instead, the designers of browsers standardized an execution environment within which a proxy for the bank, in the form of downloaded JavaScript code, could execute. Analogously, the makers of Things could get together to establish standard messaging protocols, for example to turn on lights or establish a temperature setpoint. But that strategy is not likely to work very well; it will be foiled by the very richness of the IoT ecosystem and its dynamically evolving nature. Our design pattern is inspired by what has worked so well in the web. We standardize on the execution environment for proxies for Things.

Once the Thing's accessor has been reified, it begins producing output data. Notice here a subtle but important point. The diagram in Fig. 2 is not a simple flowchart nor a simple dataflow diagram. Were it either of these, the Mutable accessor

block would produce one datum on its output in reaction to each accessor and/or control input. But this is not what it does. Each time it receives a new unique accessor input, it reifies the accessor, and that accessor begins to produce outputs that can be either spontaneous or reactions to inputs. If the outputs are spontaneous, they occur at some rate determined by the device and will continue to emerge from the Mutable accessor until a new accessor input is received or the device itself stops providing data.

The sensor data emerging from the Mutable accessor streams right to the ConstructUI script actor. In our prototype, this script produces HTML and CSS code to be overlaid on the image, as shown in Fig. 1. This HTML code includes input elements that can be used to send control data back to the accessor that reifies Mutable. In Fig. 1, the "step size" and "sampling period" controls are fed back to the Thing's accessor whenever the user taps or clicks on the Submit button.

The loop at the bottom illustrates the integration of entirely disjoint technologies into the app in order to get voice control. This could be used, for example, to scroll between tags in the event that multiple AR tags are detected in the image. Beginning at the bottom left, the "VoicePromptGenerator and Tag Selector" script starts off by outputting text which then gets converted to an audio signal using a text-to-speech service. This could be a cloud-based service such as those provided by Google or Amazon or a locally realized service. The resulting audio data is fed to an AudioPlayer accessor, which provides access to the local audio hardware. Having produced a voice prompt, the AudioCapture accessor is triggered, which listens for a response. The resulting audio signal is sent to a SpeechToText accessor, which again could use a cloud-based service or a local one. In our prototype, we then further process the resulting text using a natural language engine, in our case the one provided by Google at API.AI, which can convert natural language into specified fulfillment commands. This is fed back to the leftmost script, which can produce a new prompt and/or an index to select a new tag in the image. This part of the app could, for example, state, "I found three devices. Is this the device you want?" If the user says "no" or "not really" (the natural language processor would handle such variability) it could scroll to the next one and state "How about this one?"

III. The Accessor Pattern

An early version of the accessor pattern is given by Latronico et al. [24], who explain them using a diagram similar to Fig. 3. The upper part of the figure shows a "swarmlet," which is a composition of components called "actors" connected by streams. The middle actor is an accessor and serves as a proxy for a "swarm service or thing." The proxy runs on the accessor host and communicates with the service or Thing by some proprietary mechanism. Lohstroh and Lee [30] distinguish the interaction between the proxy and its service or Thing, which conforms to a "vertical contract," from the interaction between the accessor and the other actors in the swarmlet, which conforms to a "horizontal contract." The horizontal

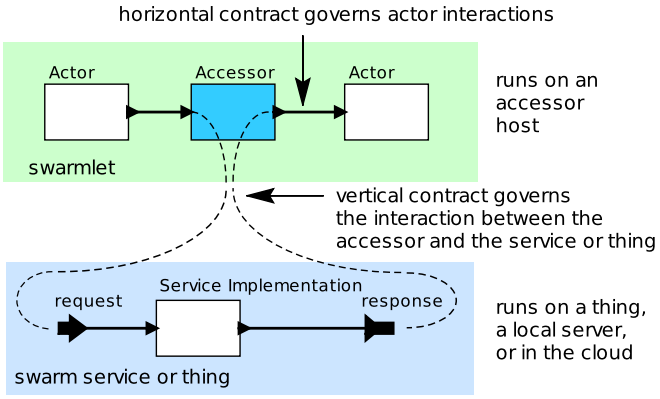

Fig. 3. Accessor in a network of actors.

```
exports.setup = function() {
  this.input('trigger');
  this.output('data', {
    'type': 'JSON',
    'spontaneous': true
  });
}
var httpClient = require('http-client');

exports.initialize = function() {
  var self = this;
  this.addInputHandler('trigger', function() {
     httpClient.get(
         'http://accessors.org',
         function(data) {
           self.send('data', JSON.parse(data.body));
         }
     );
  });
}
```

Fig. 4. Simple accessor for a RESTful service.

contract enables composition of multi-vendor services and Things, whereas the vertical contract enables device and host-specific interaction mechanisms, using for example any of a variety of radio or networking technologies and protocols such as HTTP, CoAP, TCP sockets, or WebSockets.

The host that runs the accessor may be a microcontroller, mobile device, edge computer, or server, whereas a Thing it interacts with is typically a separate piece of hardware, not necessarily proximate to the host. Or the accessor can access a cloud-based service.

A simple accessor definition is shown in Fig. 4. The "setup" function defines two ports, an untyped "trigger" input and JSON-typed "data" output. The accessor then declares that it requires an "http-client" module, which must be provided by the host if the host is to be compatible with this accessor. The host, moreover, can restrict the sorts of HTTP requests that the module supports, as done for example by the "same-origin" policy in browsers, which enhances security of web pages. The "initialize" function adds an input handler for the "trigger" input. Whenever an event arrives on this input, this

handler function will be invoked. On line 13, the input handler function uses the required module to make an HTTP request. It provides to that module a callback function that will parse the response and send it to the "data" output port. Of course, a better designed accessor should check for errors and handle timeouts, but we hope this is enough to get a feel for how an accessor is specified.

In this example, inputs at the "trigger" port cause an output to be produced sometime later. These inputs and outputs interact with other accessors or actors according to the horizontal contract, which is based on actors [17], [1], but with a more deterministic temporal semantics similar to that used in discrete-event simulators. We explain this in the next section.

IV. Coordination

Coordination can be thought of as constrained interaction [44]. Much like type systems, which can prevent programs from "going wrong" [35], coordination can make programs satisfy certain desirable properties, such as determinism, liveness, and fairness, *by construction*. Importantly, disciplined coordination reduces the need for burdensome validation and testing. A coordination model or language [38] implements coordination rules that endow a coordinated ensemble with a semantics, often called a "model of computation" (MoC). For accessors, the host realizes the MoC.

The MoC is common for all accessors, but the mechanism by which the accessor interacts with a Thing or service is not. That mechanism can be built on top of established standards, such as HTTP, MQTT, datagrams, WebSockets, etc., but there is too much diversity among devices and services to reasonably expect common details to emerge. Hence, an accessor can be thought of as an adapter that translates between two incompatible protocols. Such adapters have been used in other design frameworks for heterogeneous systems such as Metropolis [10]. Of course, within specific application domains, such as Internet-controlled lighting, manufacturers could benefit from establishing local standards. This would enable, for example, the same accessor to work with products from multiple vendors. But nothing about our approach requires the establishment of such standards. Our approach is therefore friendlier to innovation, multi-vendor compositions, and new entrants into markets.

As illustrated by the example in Fig. 2, IoT applications tend to be highly concurrent, so concurrency needs to figure prominently in the MoC. In that application, concurrency appears in two ways. First, the actors in the model are concurrent in that they can (conceptually or physically) execute at the same time modulo data dependencies. The ObjectRecognizer and the AudioPlayer, for example, are concurrent with no data dependencies between them. Second, several of these actors spawn remote actions on Things or cloud-based services and there can be several pending actions awaiting responses all active at the same time. The ObjectRecognizer, TextToSpeech, SpeechToText, and NaturalLanguage accessors, for example, may all use cloud-based services with RESTful APIs. Notice that neither of these forms of concurrency is about performance, about speeding the execution of software. Instead, concurrency is intrinsic in the distributed nature of IoT applications and the interaction of software with Things. If the application designer attempts to manage this concurrency using threads, chaos is likely to ensue, with unexpected nondeterminism and deadlocks a constant threat. The accessors framework instead provides a much more structured concurrent MoC, which we describe in this section. The key idea is to combine a discrete-event model of computation with asynchronous atomic callbacks.

A. Discrete-Event Systems

The diagram in Fig. 2 is an executable model given in a graphical syntax. Each icon represents a reactive piece of software, either an accessor from a reusable library or an application-specific script. We call these pieces of software "actors." Each actor has parameters and input and output ports. The "wires" connecting ports convey messages from one actor to another. In our semantics, these messages are events occurring at a logical time. An event has a time stamp, and the execution semantics ensures that an actor sees input events in time-stamp order. Moreover, if events with the same time stamp are received on multiple ports, our semantics ensures that the actor can see *all* such events in the same reaction. We thereby avoid a form of nondeterminism in which simultaneous events (those with identical time stamps) may be processed in nondeterministic order, as they would be in a more classical actor model.

Such discrete event (DE) systems have a long history and have been used primarily for simulation [31], [8], [15], [25], [7], [47]. In our case, we are using DE semantics not for simulation, but for run-time deployment, as has been done in Ptides [48], [13] and Spanner [9]. Ptides and Spanner extend the time-stamp semantics across networks so that time stamps have a global meaning in a distributed system. Accessors are compatible with Ptides, in principle, and therefore the semantics of DE can extend across a network to coordinate actions on several distributed hosts. Such an extension, however, is beyond the scope of this paper. We focus instead on how DE is combined with the highly asynchronous actions of Internet interactions.

As with any framework, there is overhead associated with abstraction. A DE scheduler needs to maintain a list of pending events to be processed sorted in time-stamp order. The core part of our implementation, which is shared among all hosts, supports defining, instantiating, connecting, and executing accessors and comprises only approximately 3000 lines of heavily-commented JavaScript code.

B. Asynchronous Atomic Callbacks

The Asynchronous atomic callbacks (AAC) concurrency pattern is used extensively in web programming, both on the server side (using for example Node.js[3] and Vert.x[4]) and on the client side, in browsers. On the server side, it has proven

[3] http://nodejs.org
[4] http://vertx.io

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 [43]) and embedded systems (e.g. TinyOS [28]).

A central feature of this pattern, which drives its scalability, is its dependence 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. For example, the callback function on line 15 in Fig. 4 will be invoked when a response from an HTTP server has been received. This callback function is passed as an argument to the module's get() function. This non-blocking behavior prevents programs from becoming unresponsive while waiting for responses from remote servers.

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. The same benefits can, in principle, be accomplished with threads, but the resulting programs are much less scalable, more difficult to understand, and vulnerable to the many nefarious bugs that multithreaded programs inevitably have [26], including unexpected nondeterminism and deadlocks. Properly written AAC applications do not use locks explicitly and cannot deadlock.

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 and compromising real-time performance. 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. Instead, they 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, 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. All of these extensions can, in principle, be used with accessors. But the combination of AAC with DE makes them much less necessary.

C. Combining DE with AAC

DE provides a streaming data model, suitable for many IoT applications, augmented with time stamps for improved determinism. AAC provides a mechanism for handling highly asynchronous delayed events. Both models are concurrent, are more disciplined than threads, and have achieved widespread acceptance. Accessors combine the two, getting the best of both worlds.

Consider again the simple accessor in Fig. 4. A trigger input that it receives occurs at a logical time and the output data occurs distinctly later than that logical time. Line 5 declares the output to be "spontaneous," which means that it does not have an immediate direct dependency on the input. The accessor host uses this information when analyzing data dependencies between actors to come up with a schedule that reacts deterministically to any particular set of time-stamped inputs.

On line 16, a callback function produces an output. This occurs asynchronously, but the AAC model ensures that it occurs atomically. No other actor can be in the middle of executing anything. This event can therefore be assigned a time stamp without risk of violating DE semantics by having events appear "in the past" or by having simultaneous events (those with identical time stamps) appear at some destination in two distinct reactions. The actual time stamp that is assigned to this "data" output is nondeterministic because it depends on the reaction time of some remote server and on network latencies. But once the time stamp is assigned, how the swarmlet reacts to this event is completely deterministic. This semantics makes behaviors more repeatable and testable. For example, we can write regression tests that check behavior given particular time-stamp assignments. The determinism of the model ensures that there is exactly one "correct" reaction to a set of time-stamped inputs.

The combination of DE with AAC was first realized by the authors by embedding a JavaScript interpreter in an actor and coordinating it with the DE director of Ptolemy II [40], which is implemented in Java. Lohstroh and Lee formalized this combination using interface automata [30].

D. Timing

Time stamps in a DE semantics are a logical concept, not a physical one. But the use of time stamps suggests a connection with the physical world. Indeed, in the IoT, physical timing of events can be quite important.

The most straightforward connection we can make between time stamps and physical time is to attempt to align them as much as possible. For example, the time stamp assigned to the asynchronous, spontaneous output on line 16 of Fig. 4 may be

taken from a physical clock on the host, and that clock may be synchronized with other clocks on the network. For this to be valid, the host needs to maintain a correspondence between the logical notion of "current time" and the time recorded on the physical clock. A simple way to do that is to delay handling of time-stamped events until the physical clock of the host reaches or exceeds the time of the time stamp. In a distributed system, clock drift will have to be taken into account as done for example in Ptides [48], [13] and Spanner [9].

Accessors will also need mechanisms to invoke actions in the future at specified logical times. These are similar conceptually to the callbacks in Fig. 4, but these future events will be assigned time stamps deterministically.

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 atomicity. 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 invoke the function at some time near the multiples of T milliseconds. When there are multiple such 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.

In [18], two of the authors (Jerad and Lee) show that these mechanisms can be given a stronger temporal semantics. For example, it is possible to ensure that if two calls to `setInterval`(F, T) and `setInterval`(G, T) are made with the same T, then the host can ensure that F and G are invoked together atomically and hence will appear to any observer as being simultaneous. Moreover, Jerad and Lee define labeled logical clock domains (LLCDs), within which islands of synchrony can be created asynchronously and coexist with a clean semantics.

These timing mechanisms have been integrated into our framework. They can be used, for example, to extend the AR application in Fig.2 with timed behavior, for example, to synchronize video and audio feedback to the user. Multiple clock domains could become useful in more complex applications where several concurrent islands of synchrony coexist.

Many callbacks, however, are untimed, like the ones in Fig. 4. Since callbacks are all atomic, during the invocation of the callback, each logical clock will have a specific "current time" that is frozen during the invocation of the callback. This makes it possible for such an asynchronous callback to then make a request for a timed callback that will be invoked logically simultaneously with other actions on the same logical clock domain. Referring again to the example in Fig. 2, this could enable audio stimuli to be synchronized with visual stimulus even if the onset of events that trigger these stimuli is asynchronous.

Ultimately, no strategy can guarantee that a timing goal is met. If a Thing fails, it fails! Our contribution is enabling detectability. A predictable, composable timing semantics is necessary to detect abnormal timing variability. The order of events is well-defined by time stamps, and any anomalous order that emerges at runtime is an indicator of a fault.

In addition, accessors are designed to be able to be run locally, close to the Things they interact with, in contrast with cloud-based services. This makes latencies more predictable, repeatable, and controllable. Some functions, however, are much easier to provide in the cloud than locally. Fig. 2 uses cloud services for speech synthesis and recognition and natural-language processing. For fault tolerance, we can use a "local first" architecture to treat cloud services as an enhancement instead of a requirement. It would be easy, for example, to augment Fig. 2 with a fallback user-interaction mechanism, such as pushbuttons on the screen. This means that the application can reliably deliver real-time behavior even if Internet connectivity is lost altogether.

E. Context Sensitivity

The Mutable accessor in Fig. 2 provides an example of a context-sensitive swarmlet, where the actual behavior depends on the Things that are locally available. Such mutation, if done in an ad hoc way, would amount to self-modifying code, which is usually not a good idea. Self-modifying code is notoriously difficult to understand and has even been found suitable as a code obfuscation technique [32]. The mechanism in the accessors framework is much more disciplined.

The Mutable accessor is a placeholder for an accessor that can reify it. Before being reified, the Mutable accessor has no functionality. It ignores input events and produces no output events, except for input events on the "accessor" input. In our implementation, when the "accessor" input receives an event, the Mutable accessor interprets this input event as a request to reify an accessor that is specified by the value of that event. The value of the event could be text similar to what is shown in Fig. 4. If the Mutable accessor has already reified an accessor, then it unreifies it and then reifies the new one. In the application in Fig. 2, this can be used to provide entirely different visual interfaces to Things in the field of view.

Note that, in principle, this dynamic substitution mechanism could be leveraged to optimize for locality or availability, as done in the example of Fig. 2, but also to cope with unexpected network outages, or for keeping up with firmware updates or unpredictable changes to remote APIs. A new accessor could be downloaded and instantiated at run-time to replace one that (for whatever reason) no longer optimally performs its function.

Discipline is important, however. The Mutable accessor has a specific role in this swarmlet, and not all accessors can satisfy this role. The role is specified by its input and output ports, "control" and "data." These ports have types, and the

reified accessor must have matching ports that conform to those types. A perfect match, however, is not required. We follow the type refinement schema for actors similar to that of Lee and Seshia [27, chapter 14]. An output data type of a reified accessor, for example, can be a subtype of the type of the corresponding output of the Mutable accessor. Conversely, an input type of the reified accessor can be a supertype of the corresponding input port of the Mutable accessor. In addition, the reified accessor need not match all the input and output ports present in the Mutable accessor. Any output port that is present in the reified accessor but not in the Mutable accessor will have its events ignored, and any input present in the Mutable accessor but not in the reified accessor will not be receiving events. Of course, a *useful* reification will have at least some input ports that match.

Dynamically reified accessors may be downloaded from the Internet as part of a discovery process. Hence, as we will discuss in Section VI, the accessor to reify is likely to not be completely trusted. Much as a browser controls the local resources that a web page can access, our hosts control the resources that the reified accessor can access. All access to resources is meditated by modules, like the http-client module that is required in Fig. 4. The module is implemented by the host and hence can be constrained in any way appropriate.

F. Hierarchy

The model in Fig. 2 is an instance of what we call a composite accessor. In that example, the composite accessor itself has no input and output ports, so it cannot be directly embedded in another swarmlet. But our accessor framework supports composite accessors with input and output ports, so models can be constructed hierarchically.

Even more interestingly, the swarmlet in Fig. 2 interacts with outside services through the network, for example by making HTTP requests. Those outside services could themselves be swarmlets, and they may have embedded within them an accessor designed for accessing the services of the swarmlet in Fig. 2.

This schema is illustrated in Fig. 5. In that figure, two networked hosts have each instantiated a swarmlet containing an accessor for the other swarmlet. When the accessor on host A receives an input event, it sends a message to the accessor on host B, which then produces an output event. The swarmlet on host B constructs a response and provides that response as input event for the accessor, which sends a message back to host A. Finally, the accessor on host A produces an output event with the response. This mechanism can be used to construct services that can then be easily instantiated remotely; the service (a swarmlet) provides an accessor that another swarmlet can instantiate.

Of course, once such peer-to-peer interactions exist, a new form of brittleness appears. One piece of a distributed application may be updated, for example, without being able to simultaneously update the other pieces. Some sort of coordinated deployment and update will have to be developed.

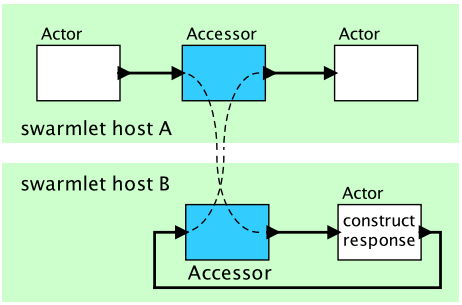

Fig. 5. Schema whereby swarmlets have accessors that can be instantiated in other swarmlets.

V. A Platform for Composing Things

Accessors are generic reusable components that can be composed in a common semantic domain with an actor-based discrete-event semantics furnished by a host implementation. As such, the host can be thought of as a platform in the sense of Platform-Based Design (PBD) [41]. The key goal of PBD is to separate functionality (the *what*) from architecture (the *how*) and be able to map a design (or parts of it) onto different architectures without having to change the design.

Platforms abound in IoT. A typical philosophy is to offer an application programming ecosystem deployed on a certain type of host, such as Node.js or a centralized cloud service. An application facilitates communication among Things using information streams, which can be acted on directly or scanned for events. Generally, the focus has been on supporting diverse host-to-Thing connections, with some success. An application developed on a particular platform is usually not transferable to another platform. This paradigm works well for a set of Things owned by the same entity and a community substantial enough to afford its own application designers.

Looking to the future, it is desirable to write an application once and deploy it on any host connected to the right Things. How should this application be written? JavaScript is an attractive candidate due to its widespread usage and compatibility with heterogeneous hosts, such as web browsers and Node.js.

One underlying problem is that JavaScript host environments differ, particularly in their mechanisms for allocating compute resources for large computations, providing permanent data storage, and global variable management. Pure JavaScript provides no such mechanisms, and hence, when such mechanisms are provided by a host, they are often provided in a host-specific way. Typically, these mechanisms are implemented in the host's native language, such as C, C++, or Java, and then provided to the JavaScript programs through modules that must be explicitly "required" by the program.

To solve this, the Accessor approach leverages an information hiding strategy. Accessor code has no direct access to platform-specific primitives. Instead, an accessor may declare dependencies on functionality contained in a host-provided module, like the http-client module in Fig. 4. A module ideally has a common API for all hosts but may have a host-specific

implementation. Thus, the host implementation details are hidden from the application developer.

An accessor can execute on any host that meets all its module dependencies. The binding between an accessor and the module(s) it relies on takes place upon instantiation of the accessor on a host. For instance, the Camera accessor in Fig. 1 can run on various hosts without assuming anything about the mechanisms used to access the camera other than what is defined in the camera module's API.

The ability to determine whether a host supports an accessor at run time provides advantages for IoT applications. Some Things execute on energy and cost-constrained leaf nodes, and therefore it may be desirable to reject accessors that push the system over budget. Accessor rejection may also be a security strategy.

Ideally, since accessors may contain untrusted code, it is preferable to execute them in a contained environment. The most straightforward way to do that is with a language interpreter or a virtual machine that provides a sandbox. Browsers, for example, already execute JavaScript in such a sandbox, and our browser host, described below, takes advantage of this extra measure of security.

Browsers and server-side infrastructure such as Node.js and Vert.x provide powerful JavaScript interpreters, but they are not lightweight enough for installation in many leaf devices. The situation may improve in the future, as a number of small JavaScript kernels have appeared recently in open-source form, such as Duktape. Some of these can execute without much operating system support, and hence may be suitable for deployment in quite constrained environments. In the remainder of this section, we outline the hosts we have prototyped, thereby demonstrating that the accessor architecture is deployable on a large variety of platforms.

A. CapeCode

The CapeCode host is a Ptolemy II configuration that provides a GUI for composing, executing, and deploying composite accessors. Ptolemy II is a Java-based open-source software laboratory that supports experimenting with actor-oriented design [14]. CapeCode provides a graphical user interface for building swarmlets. The name of this host is derived from Cape Cod, Massachusetts, where much of the development has occurred.

Models that consist solely of accessors may be code generated into composite accessors that may be executed by any accessor host that implements the modules used by the accessors. CapeCode can execute the generated composite accessors either locally or deploy them remotely on the Node host, which we discuss in the next section. These composite accessors can also be loaded and executed in the Browser host by embedding a reference in a web page. Developing composite accessors using the CapeCode GUI and then deploying them to possibly less powerful remote machines is an effective development strategy. Further, CapeCode can execute models combining accessors with other Ptolemy II actors, thereby providing a rich library of predefined actors.

A drawback of using a graphical syntax to express a more complex or elaborate design is that it tends to result in exceeding the Deutsch limit [12] of fifty graphical elements, which can make the model unwieldy and difficult to understand. Ptolemy II uses hierarchy to decompose a design into comprehensible pieces. To scale up to very large numbers of actors that are widely distributed in networks, we can consider application builders like Chisel [5], which provides programmed construction of digital circuits. It leverages higher-order functions featured in the Scala programming language. In Chisel, rather than directly instantiating and connecting components, a designer writes a program that instantiates and connects components when it executes.

B. Node

The Node host is meant for use with Node.js. Node.js is a popular cross-platform server-side JavaScript runtime environment. Because of the popularity of Node.js, we anticipate that the Node host will have the greatest impact of all of the hosts. In particular, we feel that providing well-defined temporal semantics to the Node environment could increase robustness and reliability.

The Node host is available via the Node Package Manager (npm) as the @terraswarm/accessors module.

C. Browser

The Browser host allows users to inspect and execute accessors in a web browser. Any web page may load an accessor by including a <script> tag pointing to the Browser host script plus a reference to the accessor(s). It is assumed that some web server is available to provide these files. A demonstration server is available publicly[5], and it features an interactive tutorial[6] for writing accessors that executes in the browser host.

The JavaScript engine in a modern browser is designed to safely execute third-party code in the local environment. Therefore, the Browser host does not typically have direct access to the local file system or to machine hardware, leaving it slightly underpowered compared to other hosts. One advantage of the Browser host is ease of deployability. Any device with a web browser can download and execute accessors simply by pointing to a URL.

An example of the Browser host is an accessor that uses a JavaScript implementation of OpenCV [42] to recognize faces in an image[7].

D. Cordova

Smartphones provide access to a wealth of sensors, offer an uplink to a LAN or the Internet, and are mobile, which means that their environment is subject to change as they are carried around by their owners. These three aspects combined make smartphones a very appealing deployment platform.

[5] http://accessors.org/hosts/browser
[6] http://accessors.org/hosts/browser/demo/tutorial/tutorial.html
[7] http://accessors.org/hosts/browser/demo/faceDetector/faceDetector.html

Apache Cordova[8] provides a development toolchain amenable to a variety of smartphone operating systems. Apps are constructed much like an ordinary web page, using HTML, CSS, and JavaScript, and can be deployed to different targets, including Apple iOS and Google Android. Superior to a browser-hosted web application, Cordova's plugins expose platform-dependent functionality such as geolocation and a file system to the application's JavaScript environment. Hence, unlike the browser host, which can also be deployed on mobile platforms, the Cordova host can selectively bypass the browser's security restrictions, giving access to platform-specific functionality.

Apache Cordova is itself a platform-based design tool, where Cordova plugins offer a uniform API while hiding Android-specific or iOS-specific implementation details. This points to the scalability of PBD, as the PBD approach supports a series of platform mappings; here, from accessors to Cordova, then Cordova to Android or iOS.

E. Duktape

The Duktape accessor host uses the Duktape JavaScript engine[9] to deploy accessors on small embedded systems.

As a proof of concept, we deployed a composite accessor to a Maxim Integrated MAX32630, which is a Cortex-M4 with 512K RAM and 2Mb flash. Our simple accessor was an accessor that produces integers connected to a display accessor. The executable had the following sizes in bytes:

- text: 291,848 - program code in flash
- data: 2,964 - initialized data in RAM
- bss: 8400 - uninitialized data in RAM

This shows that accessors can be deployed on deeply embedded platforms.

To use the Duktape host on a composite accessor would require implementing in C/C++ the modules used by the accessors. For example, to support the accessor in Fig. 4 requires implementing the http-client module, which would most likely be implemented in C or C++ using low-level networking primitives.

VI. Security

Accessors are untrusted code that serve as proxies for sensors, actuators, and services. Inspired by the web, accessors are therefore executed in a virtualized environment that controls access to resources and data. Such encapsulation provides a starting point for ensuring security and privacy, but it is not sufficient by itself. In particular, the execution environment will have to grant access to physical resources such as sensors and actuators in order to realize IoT applications. How should it authenticate the IoT applications (e.g., whether an application running remotely is modified from the original program components)? Moreover, how can we make sure we only grant permission to legitimate IoT applications to access certain resources (authorization or access control)?

[8]https://cordova.apache.org
[9]http://duktape.org

While it is crucial to provide appropriate authentication and authorization for the IoT applications, it is also challenging to do so. Many IoT applications can run on a variety of software/hardware platforms, possibly in a distributed fashion. Some of them run under a constrained energy budget or with restricted communication capability. It would be unreasonable to expect those types of applications to incorporate traditional security measures for the web, such as SSL/TLS (Secure Socket Layer/Transport Layer Security), based on a Public Key Infrastructure (PKI), because it requires a frequent use of power-hungry public-key cryptography. Leveraging the Kerberos authentication system [37] is also difficult because it requires a constant stable connection to the authentication server and an interactive prompt for users to enter passwords. Passwords are not appropriate for Thing-to-Thing interaction.

In addition, IoT applications tend to operate in open (or even hostile) environments and thus are at higher risk of being compromised or subverted. As an example of this, Ghena *et al.* [16] demonstrated an attack on traffic controllers on the streets of Ann Arbor, Michigan, by leveraging access to wireless communication used by traffic controllers. Therefore, it is sorely important to monitor the behavior of IoT applications and revoke access to safety-critical resources as soon as a security breach has been detected. Due to the scale of IoT applications, in terms of both the number of applications and the volume of data traffic, it is not a feasible strategy to solely rely on digital certificates because a PKI with tens of billions of certificates will quickly become unmanageable.

Our open-source toolkit, SST (Secure Swarm Toolkit) [20], provides a set of accessors for bringing authentication and authorization to the IoT while addressing above-mentioned challenges. SST uses a local authorization entity called Auth [21] deployed on edge computing devices that act as local gateways to the Internet for IoT applications. SST employs a locally centralized, globally distributed [19] approach, which has two key benefits: (1) dependency on a reliable connection to Cloud servers is limited, which improves robustness to network failures, and (2) better scalability can be achieved by distributing the workload for authentication and authorization among Auths. Various security configuration alternatives supported by SST also embrace heterogeneous security requirements and resource availability in the platforms for IoT applications.

Fig. 6 illustrates a part of an extended version of the augmented reality example in Fig. 2, secured by one of the accessors provided by SST, SecureCommClient. A stream of output data from the Mutable accessor is encrypted and sent to a cloud server via the SecureCommClient accessor. Let us assume there is another IoT application, namely SensorAnomalyDetector, running on a remote cloud server and programmed using another accessor in SST, called SecureCommServer. SensorAnomalyDetector takes streams of data from the distributed augmented reality applications reporting their sensor data, executes a machine learning algorithm on collected data, and sends feedback to the applications when any sensor data anomaly is detected. When a client application receives feedback on a detected anomaly from the server, the feedback

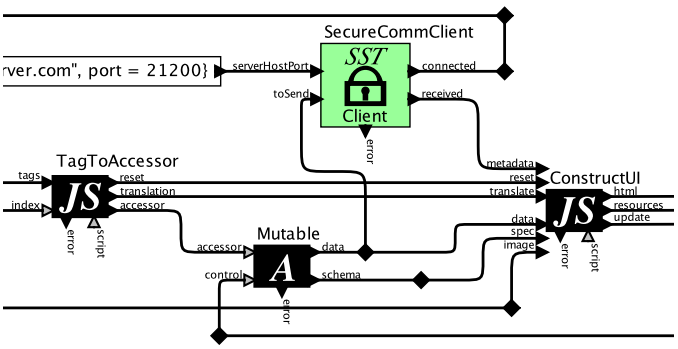

Fig. 6. Modified part of the example in Fig. 2 with a SecureCommClient accessor for additional security.

is sent to the graphic overlay's additional input port, *metadata*, to indicate the anomaly as part of the overlay.

In this extended example, the main function of these security accessors is to access the authentication and authorization services required to establish a secure channel between two swarmlets, each of which are associated with their own (possibly different) Auth. It is assumed that a trust relationship between the involved Auths exists. After being authenticated and authorized by their respective Auths, SecureCommClient and SecureCommServer establish a secure communication channel between each other similar to that of a client-server connection via SSL/TLS. But with SST, we have the option to choose the underlying network protocols (e.g., TCP or UDP) or the cryptographic protocol. We also do not have to maintain a centralized certificate authority (CA). Adding SST accessors provides additional security guarantees including confidentiality and message authenticity, preventing network-based attackers from eavesdropping or staging a man-in-the-middle attack. In addition to the two aforementioned accessors, SST provides accessors for constructing IoT applications based on a publish-subscribe communication style using accessors called SecurePublisher and SecureSubscriber.

Another benefit of using SST accessors comes from encapsulation of cryptography operations and cryptographic key management. As Myers and Stylos [36] point out, design of APIs is critical for software security, especially in the sense that misuse of APIs can lead to serious security problems. With SST accessors, all software developers need to do is to specify configuration parameters and set up initial credentials (e.g., generate a public-private key pair and register the public key with an Auth). Even developers with moderate knowledge in security need not worry about internal cryptographic operations and encryption key management for accessors once the accessor design is correct. Specifying security configurations can be further simplified by using the given profiles as suggested in [20].

Combined with actor-oriented modeling semantics where actors communicate only through input and output ports, isolation of cryptographic keys and operations in SST accessors can enhance security when supported by OS-level or architecture-level security mechanisms. By sandboxing the execution of SST accessors, a swarmlet host can restrict the privilege of accessors to read from or write to arbitrary files or network ports, preventing credentials from being leaked or being used maliciously (e.g., by an attacker to spoof the device). If the host is equipped with an architectural security mechanism such as Intel's SGX (Software Guard Extensions) [33], the credentials can be protected even when other processes on the host or the host's middleware or hardware components are compromised. Like other accessors, SST accessors also require some modules to be available on the host. These modules include the crypto module and the TCP or UDP modules.

VII. Privacy

Fundamentally, accessors are a disciplined form of mobile code. An accessor can be dynamically downloaded and instantiated on a wide variety of platforms, including deeply embedded ones. This has a potential advantage for preserving privacy because computation can be easily moved to a data source rather than the more common scenario where the data is fed into a cloud service for processing.

As Jaron Lanier explains in his book "Who Owns the Future?" [23], a key aspect of the business models of Silicon Valley tech companies that provide such cloud services (and typically do so free of charge) is the extraction and accumulation of information from and about its users. Furthermore, cloud services that make use of virtual resources such as those provided by Amazon AWS[10] or Microsoft Azure Cloud[11] may not be designed to spy on its users, but can still be vulnerable to side channel attacks such as Meltdown [29] and Spectre [22] and potentially leak sensitive data.

A shift away from a centralized cloud-based paradigm towards mobile code and local computation could thus be an effective strategy to foster better privacy on the IoT. Accessors can reduce the need to transport data over the open Internet and allow for designs that carry out analysis locally and therefore rely less on the cloud, or they can anonymize data before sending it into the cloud for further processing.

VIII. Related Work

A number of projects adopt an actor-oriented approach similar to ours for IoT system development. The closest are probably Calvin [39], Node-RED[12], and NoFlo[13], which use a dataflow concurrency model for interactions between services that are using AAC. Also reasonably close, although very different in syntax, is Rx (Reactive Extensions), which combine callbacks with streams [34]. We believe that our use of discrete-event semantics is unique and offers a solid foundation for deploying timing-sensitive IoT applications.

Also related are publish-and-subscribe services such as MQTT, DDS, and ROS. MQTT, for Message Queue Telemetry

[10] https://aws.amazon.com
[11] https://azure.microsoft.com
[12] https://nodered.org
[13] https://noflojs.org

Transport, is an ISO standard (ISO/IEC PRF 20922) intended for embedded applications where small footprint code is required and/or network bandwidth is limited. The Data Distribution Service (DDS) is an Object Management Group (OMG) machine-to-machine standard for real-time communication using publish-and-subscribe pattern. ROS, the Robot Operating System, an open-source software framework originated by Willow Garage, is widely used for building robot applications. These can be used by accessors for vertical communication with Things and services, and hence are complementary to our work. We have used MQTT in accessors for wireless sensor-network devices and ROS in accessors that control robots. But similar to accessors, these services provide a communication fabric that can stitch together components. Unlike accessors, they use a publish-and-subscribe pattern, where a data producer generates messages for a "topic" and data consumers that are subscribed to the topic will be notified, typically by a centralized broker that mediates the communication. None of these use time stamps to provide a deterministic interaction semantics, however, so applications are less testable and harder to deploy in timing-sensitive scenarios. Time stamps help by controlling the order in which events are handled, thereby ensuring that given the same inputs, the application always delivers the same behavior.

IFTTT (IF This Then That) is a free web-based service originally created by Linden Tibbets and Jesse Tane that enables composing Things and services using a simple imperative style with chains of conditional statements. It is fundamentally a cloud-based approach, and it excels at integrating with other cloud-based services such as email, Facebook, and Pinterest. Accessors, in contrast, need not run in the cloud. An application built with accessors would usually run in a host that is much closer to the Things it is interacting with. The AR application in Fig. 2, for example, is designed to run on a phone, tablet, or head-mounted AR display that is in the same room as the Things it is interacting with. When feedback control is involved, local execution can be critical because latency can strongly affect behavior.

Accessors can, in fact, leverage the considerable ecosystem around IFTTT. Like the accessors in Fig. 2 that use cloud-based services for language and speech, accessors could be easily designed to interact with IFTTT.

In IFTTT, for the most part, each Thing requires its own "channel," the name for the mechanism that IFTTT uses to interact with the Thing. Accessors, similarly, can be created independently for each Thing. An application that is built with such "channels" or Thing-specific accessors will not work if the Thing is replaced by a variant from another manufacturer. However, as illustrated by the Mutable accessor in Fig. 2, accessors offer the intriguing possibility of more vendor-independent applications. The application in Fig. 2 will work with any device for which there is an accessor that provides JSON-formatted data and (optionally) accepts JSON-formatted commands. There are many such devices. We need no prior agreement on what particular JSON structures are used nor even on what communication mechanism is used to talk to the Thing. The accessor may access the Thing via the Internet, Bluetooth, or ZigBee, for example, and it may use protocols such as HTTP, WebSockets, MQTT, TCP sockets, or datagrams, as long as the host provides modules supporting these technologies. IFTTT has no such mechanism for this level of vendor neutrality.

One approach to achieving vendor neutrality is represented by Home Connect, a protocol and a corresponding app from BSH Bosch and Siemens Hausgerate that controls multiple brands of home appliances. This is superficially similar to accessors in that it integrates Things from diverse vendors, but it is really quite different. First, it does not directly support Thing-to-Thing interaction, although its protocol is supported by IFTTT, and hence, using IFTTT, it is possible to build Thing-to-Thing interaction with Home Connect appliances. But more importantly, Home Connect standardizes the communication protocol between the Thing and the app. It requires the Thing to use a specific communication protocol, such as HTTP over the Internet, and it dictates the format that commands and data must have. Accessors, in contrast, standardize the interface between an accessor and its host, like web browsers, which standardize the interface between a downloaded JavaScript program and the computer on which the browser runs. Accessors do not standardize the mechanism that is used to communicate with the Thing.

Accessors, fundamentally, are stitched together by a coordination language, which has a syntax and a semantics. 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 [6], and Akka, codify programming models that manage parallel computing resources, scalable workloads, and/or long network latencies. A common thread in the new languages is to embrace elements of functional programming, particularly to make functions first-class objects in the language. Functional programming can be used to realize design patterns such as asynchronous atomic callbacks and structured parallelism such as map-reduce [11]. A common thread in the frameworks (Spark, Orleans, Akka) is support for stream computation based on actors [17], [1]. Our work overlaps with these by embracing functional programming and stream-based computation, but our work appears to be unique in its adoption of discrete-event semantics.

Another recent trend that pays more attention to timing is the focus on *real-time* data analytics. The IoT promises a flood of sensor data. Many organizations already are collecting but not effectively using vast amounts of data. The research and consulting firm Forrester defines "perishable insights" as "urgent business situations (risks and opportunities) that firms can only detect and act on at a moment's notice." Fraud detection for credit cards is one example of such perishable insights. This has a real-time constraint in the sense that once a fraudulent transaction is allowed, the damage is done. In CPS, a perishable insight may be, for example, a determination of whether to apply the brakes on a car, where a wrong or late

decision can be quite destructive.

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[14], are getting a lot of attention. Examples of algorithmic innovations for real-time streaming data include adaptations of machine learning and optimization algorithms [2], [3] and adaptations of formal methods [4] to operate on streams. We believe that our component architecture could contribute quite a bit to such projects by integrating Things.

IX. Conclusion

IoT services are intrinsically an amalgam of heterogeneous and distributed components. It is naive to assume that any single standard will emerge for interaction between Things, services, and users. The accessors framework described in this paper provides a number of key properties not found (at least not all together) in any IoT framework that we know of today:

- The use of proxies for Things, where proxies execute in a host-controlled environment, similar to web pages with scripts;
- Embracing heterogeneity by not standardizing the means by which Things and services communicate with applications;
- An actor-oriented streaming model of computation for interactions between components;
- Time-stamped messages with deterministic interleaving semantics;
- Use of functional programming concepts, particularly asynchronous atomic callbacks, to hide network latencies;
- Deterministic timed interactions between components; and
- Integration of state-of-the-art security including encrypted communication, authentication, and authorization.

A great deal of work remains to be done. Most interesting to us is the possibility of leveraging the time-stamped interactions between components to build more deterministic *distributed* real-time applications. These could be based on the semantics of Ptides [48], [13] and Spanner [9], but also will require more support for dynamically changing component interactions and for large numbers of components. The mechanisms used in Orleans [6] look particularly promising, where a distributed registry of actor instantiations together with multiplexing of streams through host-to-host communication channels facilitates scalability to very large numbers of actors and hosts.

Acknowledgments

This work was supported 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; by the National Science Foundation (NSF) award #1446619 (Mathematical Theory of CPS); by the iCyPhy Research Center at UC Berkeley, supported by the following companies: Denso, Ford, IHI, National Instruments, Siemens, and Toyota; and by the Fulbright Scholar Program, a program of the United States Department of State Bureau of Educational and Cultural Affairs.

[14] https://rise.cs.berkeley.edu

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] I. Akkaya. Data-driven cyber-physical systems via real-time stream analytics and machine learning. Report, EECS Department, University of California, Berkeley, October 25 2016. PhD Thesis, Technical Report No. UCB/EECS-2016-159.

[3] I. Akkaya, S. Emoto, and E. A. Lee. PILOT: An actor-oriented learning and optimization toolkit. In *Second International Workshop on Robotic Sensor Networks (RSN), part of Cyber-Physical Systems Week*, 2015.

[4] R. Alur, D. Fisman, and M. Raghothaman. Regular programming for quantitative properties of data streams. In *European Symposium on Programming Languages and Systems (ESOP)*, volume LNCS 9632, pages 15–40. Springer, 2016.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC)*, pages 1216–1225. ACM/EDAC/IEEE, 2012.

[6] P. A. Bernstein and S. Bykov. Developing cloud services using the Orleans virtual actor model. *IEEE Internet Computing*, 20(5):71–75, 2016.

[7] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[8] A. Cataldo, E. Lee, X. Liu, E. Matsikoudis, and H. Zheng. Discrete-event systems: Generalizing metric spaces and fixed point semantics. Technical Report UCB/ERL M05/12, EECS Department, University of California, April 8 2005.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(8), 2013.

[10] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, and Q. Zhu. A next-generation design framework for platform-based design. In *Design Verification Conference (DVCon)*, pages 239–245, 2007.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150. USENIX Association, 2004.

[12] P. Deutsch. Comp.lang.visual FAQ. ftp://rtfm.mit.edu/pub/usenet/news.answers/visual-lang/faq. Accessed: 2017-06-27.

[13] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):45–59, 2012.

[14] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[15] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.

[16] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman. Green Lights Forever: Analyzing the Security of Traffic Infrastructure. In *The 8th USENIX Workshop on Offensive Technologies (WOOT '14)*, San Diego, CA, Aug. 2014.

[17] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363, 1977.

[18] C. Jerad and E. A. Lee. A JavaScript extension providing deterministic temporal semantics for the Internet of Things. Technical Report UCB/EECS-2017-136, EECS Department, University of California, Berkeley, Aug 2017.

[19] H. Kim and E. A. Lee. Authentication and authorization for the Internet of Things. *IT Professional*, 19(5), September 2017. to appear.

[20] H. Kim, E. A. Lee, E. Kang, and D. Broman. A toolkit for construction of authorization service infrastructure for the Internet of Things. In *Int. Conf. on Internet-of-Things Design and Implementation (IoTDI)*, pages 147–158. ACM/IEEE, April 2017.

[21] H. Kim, A. Wasicek, B. Mehne, and E. A. Lee. A secure network architecture for the Internet of Things based on local authorization entities. In *The 4th IEEE International Conference on Future Internet of Things and Cloud*, pages 114–122, Vienna, Austria, Aug. 2016.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[23] J. Lanier. *Who Owns the Future?* SIMON & SCHUSTER, 2013.

[24] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of swarmlets. *IEEE Internet Computing*, 19(2):20–28, 2015.

[25] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[26] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[27] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. MIT Press, Cambridge, MA, USA, second edition, 2017.

[28] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[30] M. Lohstroh and E. A. Lee. An interface theory for the Internet of Things. In *International Conference on Software Engineering and Formal Methods (SEFM)*, volume LNCS 9276, pages pp. 20–34. Springer, 2015.

[31] E. Matsikoudis, C. Stergiou, and E. A. Lee. On the schedulability of real-time discrete-event systems. In *International Conference on Embedded Software (EMSOFT)*. ACM, 2013.

[32] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679 – 691, 2011.

[33] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 10:1–10:9. ACM.

[34] E. Meijer. Your mouse is a database. *ACM Queue*, 10(3):20, 2012.

[35] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[36] B. A. Myers and J. Stylos. Improving API usability. 59(6):62–69.

[37] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos network authentication service (V5). RFC 4120, July 2005.

[38] G. Papadopoulos and F. Arbab. *Coordination Models and Languages*, volume 46, pages 329–400. Academic Press, 1998. According to Arbab, a comprehensive survey.

[39] P. Persson and O. Angelsmark. Calvin — merging cloud and IoT. In *6th International Conference on Ambient Systems, Networks and Technologies (ANT)*, 2015.

[40] C. Ptolemaeus. *System Design, Modeling, and Simulation Using Ptolemy II*. Ptolemy.org, Berkeley, CA, USA, 2012.

[41] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of IEEE*, 95(3):467–506, 2007.

[42] S. Taheri, A. V. Veidenbaum, A. Nicolau, and M. R. Haghighat. OpenCV.js: computer vision processing for the web. Technical Report CECS TR 17-02, University of California, Irvine, June 2017.

[43] T. von Eicken, D. Culler, S. Ooldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, 1992.

[44] P. Wegner. *Coordination as constrained interaction (extended abstract)*, pages 28–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[45] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[46] T. Zachariah, J. Adkins, and P. Dutta. Demo: Browsing the web of things with Summon. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 481–482, New York, NY, USA, 2015. ACM.

[47] B. Zeigler. *Theory of Modeling and Simulation*. Wiley Interscience, New York, 1976. DEVS abbreviating Discrete Event System Specification.

[48] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 259 – 268. IEEE, 2007.