A Vision of Swarmlets

Elizabeth Latronico, Edward A. Lee, Marten Lohstroh,
Chris Shaver, Armin Wasicek, Matthew Weber

{beth,eal,marten,shaver,arminw,matt.weber}@eecs.berkeley.edu
University of California, Berkeley

***Abstract*—“Swarmlets” are applications and services that leverage networked sensors and actuators with cloud services and mobile devices. This paper offers a way to construct swarmlets by composing “accessors,” which are wrappers for sensors, actuators, and services, that export an actor interface. We propose that an actor semantics provides ways to compose accessors with disciplined and understandable concurrency models, while hiding from the swarmlet the details of the mechanisms by which the accessor provides sensor data, controls an actuator, or accesses a service. This architecture can leverage the enormous variety of mechanisms that have emerged for such interactions, including HTTP, Websockets, CoAP, MQTT, and many others. Recognizing that these standards have emerged because of huge variability of requirements for bandwidth, latency, and security, accessors embrace heterogeneity instead of attempting to homogenize.**

I. Introduction

The rapid growth of networked smart sensors and actuators today presents enormous challenges and opportunities. Called the Internet of Things (IoT), Industry 4.0, the Industrial Internet, Machine-to-Machine (M2M), the Internet of Everything, the Smarter Planet, TSensors (Trillion Sensors), or The Fog (like The Cloud, but closer to the ground), the vision is of a technology that deeply connects our physical world with our information world. Many technologies have emerged to address the diverse concerns of IoT, a few of which are listed in Table I. These mechanisms provide ways to discover and communicate with devices and services. Many leverage established technologies that were originally developed for ordinary Internet usage.

Providing truly universal interoperability in order to build applications, compose services, and manage data is quite difficult, however. A key challenge is the enormous diversity of requirements. Power, bandwidth, latency, and assurance requirements can differ dramatically. For example, at one extreme, an energy harvesting device cannot support elaborate protocols and does not require much communication bandwidth. At the opposite extreme, real-time visual feedback from a flying drone to its remote controller may require very high bandwidth and very low latency. Both extremes, and everything in between, are useful.

Established Internet technologies fit many cases, but can encounter trouble at the extremes. Many IoT devices today, such Internet-controllable light bulbs, thermostats, and home security systems provide an interface using a Representational State Transfer (REST) architectural style [1] via the HTTP protocol. However, a tiny resource-constrained device likely cannot run an entire TCP/IP stack, and it is unlikely that

Acknowledgments: This work was supported in part 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.

Name	Category
Alljoyn	discovery https://www.alljoyn.org
Contiki	operating system http://www.contiki-os.org
CHROMOSOME	middleware http://www.fortiss.org/en/research/projects/chromosome
CoAP	connectivity protocol http://wikipedia.org/wiki/Constrained_Application_Protocol
Krikkit	API http://eclipse.org/proposals/technology.krikkit
MQTT	connectivity protocol http://mqtt.org
Node Red	software integration platform http://nodered.org
OceanStore	data storage http://oceanstore.cs.berkeley.edu
OpenRemote	software integration platform http://www.openremote.org
Open Sound Control	networking and communication http://opensoundcontrol.org
OpenWSN	networking and communication http://openwsn.org
POMI	networking and communication http://pomi.stanford.edu
SensorAndrew	connectivity protocol http://sensor.andrew.cmu.edu:9000/wiki
SmartHome	software integration platform http://eclipse.org/smarthome
sMAP	connectivity protocol http://www.cs.berkeley.edu/~stevedh/smap2
The Thing System	software integration platform http://thethingsystem.com
Thread	wireless mesh network http://www.threadgroup.org
UPnP	discovery http://www.upnp.org
XMPP	connectivity protocol http://xmpp.org

TABLE I
Some IoT Resources

HTTP can deliver adequate bandwidth or latency for the communication between a drone and its controller. Hence, any solution for IoT will have to embrace heterogeneity.

Composing services presents further problems. Suppose you want to connect your light bulbs to your home alarm system? Smartphone apps are the closest we have today to a *de facto* standard means of accessing IoT devices, yet they are difficult to compose. This problem has created an opportunity for aggregators. On the software side, for example, IFTTT (if this, then that), supports construction of "recipe" rule sets that accept input from Internet-accessible services and can issue responses using Internet-accessible services. All-in-one hubs like Revolv, The Thing System (TTS), and SmartThings provide hardware and/or software that consolidate access to various smart-home devices into a single app. Lacking standards, services and devices operate using proprietary APIs and mechanisms, so at any given time, aggregators "support" only a bounded set of third-party devices and services. This can slow innovation by hindering new entrants to the market, whose products are not supported.

The Industrial Internet Consortium (with the Object Management Group and affiliates) and the IEEE Standards Association are leading efforts to promote interoperability standards. The goal of this paper is to offer a particular approach to such interoperability that we believe can scale well, and can enable millions of creative minds to invent new IoT technologies by composing existing ones. Following [4], we will call applications that integrate networked sensors and actuators "**swarmlets**," a bow towards Rabaey's phrase "the swarm at the edge of the cloud," which hints at the potential vastness (and ominousness) of these devices [7]. The goal of this paper is to shift the focus away from standardizing over-the-network communication and APIs, and towards mechanisms by which diverse, proprietary, and secure communication protocols and APIs do not hinder interoperability of devices and services.

II. A Scenario

Consider a scenario where a startup company produces a robot called C4PO that will wander a space, such as a factory floor, that already contains a variety of networked sensors, security devices, and other robots made by other vendors. Those vendors have never heard of the company providing C4PO and offer no support for its interfaces, and the vendor of C4PO has no knowledge of the over-the-network APIs provided by those third-party devices. Upon removing the robot from the box, the user configures it with credentials for accessing the local wireless network, perhaps using a smartphone app and optical communication, as done by Electric Imp.Once it is on the local network, it sends a multicast discovery packet, in the style of UPnP,for example. It receives responses providing IPv6 addresses from which it can obtain XML files defining **accessors**, which are the main focus of this paper. The robot can download an accessor and execute it in a manner similar to the way a browser today downloads HTML5 with JavaScript and executes it.

C4PO begins by filtering the accessors it has discovered to select those providing proximity data. Specifically, it looks for accessors that, when triggered, output a measure of the physical distance from C4PO to the device to which the accessor provides access. C4PO need not know exactly how this distance measure is obtained. All it needs to know is that the accessor indeed provides a distance measure, and that it is capable of executing the accessor (it is a suitable **accessor host**). For example, such an accessor may declare that it "requires" a Bluetooth API to be provided by the accessor host (C4PO) so that it can provide a measure of distance using a BLE beacon technology such as iBeacon. C4PO has Bluetooth capability, so it is compatible with the accessor and can execute it. Another accessor may declare that it requires a GPS API, and since C4PO has no GPS capability, it is unable to execute that accessor. C4PO collects all compatible accessors that it discovers on the local network, and begins executing them.

Some of the accessors begin providing output data, and some do not (some devices are out of range). Those that do, provide (noisy) distance measures to a device such as a video camera, a motion sensor, or a lock on a door. C4PO wishes to construct a map of the local environment, but noisy distance measures do not provide enough information. But since C4PO can move, and using dead reckoning can estimate its motion, it can combine multiple measurements to convert noisy distance information into relative location estimates. To do this, C4PO leverages another accessor that provides a cloud-based machine-learning service that, behind the scenes, uses particle filtering to estimate location, and an optimization algorithm to recommend motion vectors to the robot.

The ML and optimization algorithms are quite computationally heavy and memory intensive, and are beyond the energy budget of C4PO, so offloading them to the cloud makes sense. However, the cloud-based service may aggregate data from multiple robots that are using its accessors that happen to be measuring distances to the same devices, as identified by their GUIDs. The possibility for such data aggregation, in fact, is a key advantage of using the cloud-based service instead of performing computation locally. This may be even more important than energy savings.

In the end, C4PO forms a map of the relative locations of objects in the environment. It can then start using that map for navigation through the space, intruder detection, or asset tracking, for example. It may leverage additional accessors provided by devices it discovers, such as video cameras that can provide it with images of its own surroundings, or sensors that provide temperature, air quality, light levels, etc. And it may instantiate accessors to leverage cloud-based services to, for example, perform image classification, face recognition, natural language understanding, etc. Most importantly, because of the actor interface provided by accessors, C4PO can compose accessors, building more complex services out of simple ones.

This scenario illustrates that if we endow devices with the ability to download and execute accessors in a manner similar to how browsers today download and execute HTML5 with JavaScript, then we can change the IoT infrastructure from one that is dominated by closed proprietary systems that manifest themselves as *platforms*, the interoperability of which is negotiated in business deals, to one that embraces heterogeneity and encourages innovative new technologies to emerge from the composition of independently designed *components*. That way, there is no need for vendors to support all other vendors' over-the-network APIs, a strategy that discourages innovation and new entrants, because each new entrant will be at best incompletely supported by already deployed systems. Moreover, the cost of interoperating with all existing deployed systems will be prohibitive, precluding small, energy efficient devices, and increasing barriers to entry into the market.

III. An Actor Model

The underlying principle behind accessors is a design pattern called **actors**, whereby concurrent components send each other messages via ports. Actors are a concurrent model of computation (**MoC**) originally proposed by Hewitt [2]. The interface of an actor, input ports that receive messages and output ports that send messages, contrast with a more typical imperative API, which defines procedures that can be invoked. Actors are intrinsically concurrent, consistent with many views of dataflow, process networks, interaction, and rendezvous-based concurrency. The Ptolemy Project has shown that these various approaches to actors can be unified under an **actor abstract semantics** [3]. The Ptolemy II open-source software framework [6] demonstrates that these various approaches can be usefully combined in heterogeneous designs.

We describe a way to construct swarmlets by composing **accessors**, which are actors that wrap sensors, actuators, and services. A **swarmlet host** instantiates an accessor, treating it as a local actor. It sends inputs to the ports of the accessor, invokes one or more of a small set of action methods, and receives outputs via the output ports of the accessor. The swarmlet host need not necessarily trust the accessor because the host executes the accessor in a sandbox to constrain potentially malicious code, similar to how a browser executes JavaScript on a web page.

The mechanism by which the accessor provides access to a sensor, actuator, or service can be any of those in Table I, but it can also be something entirely custom and proprietary. In effect, we shift the standardization problem from trying to find commonality among the huge variety of protocols in Table I to using an interfacing schema (actors) that has already been demonstrated capable of handling many of the required concurrent interaction patterns. Moreover, this architecture makes it much easier to compose separately developed accessors to create new accessors and services. Actor compositions have well-studied and desirable formal properties, which helps yield predictable behavior under composition. The architecture thus facilitates creative composition of multi-vendor IoT devices and services.

IV. Accessors

An **accessor** is a component class that a swarmlet instantiates to access a device or service. The pattern is illustrated in Figure 1. An accessor has **inputs**, by which the swarmlet makes requests, and **outputs**, by which the service issues responses. The responses need not be synchronous with the

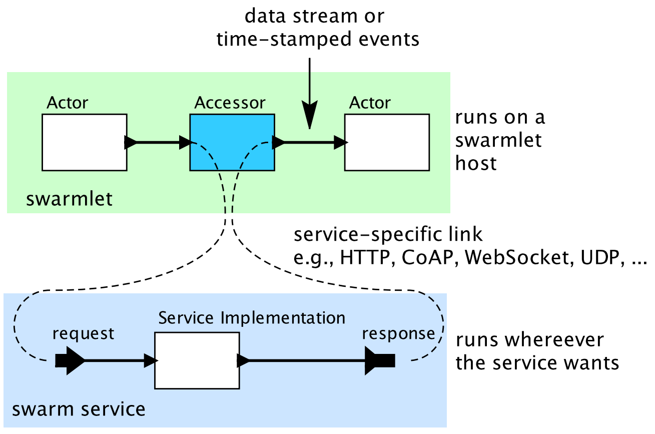

Fig. 1. Design pattern of accessors.

```
<?xml version="1.0" encoding="utf-8"?>
<class name="Hue"
       extends="org.terraswarm.JSAccessor">
  <requires name="http" .../>
  <input name="bridgeIPAddress" ... />
  <input name="lightID" ... />
  <input name="brightness" ... />
  ...
  <output name="success" .../>
  <output name="error" .../>
  ...
  <documentation type="text/html"> ... </documentation>

  <script type="text/javascript">
    // <![CDATA[
function initialize() {...}
function fire() {
   var command = '{"on":false,';
   if (get(on)) {
      command = '{"on":true,';
   }
   command = command
         + '"bri":' + get(brightness) + ','
         + '"hue":' + get(hue) + ','
         + '"sat":' + get(saturation) + ','
         + '"transitiontime":' + get(transitiontime)
         + '}';
   var response
     = httpRequest(url, "PUT", null, command, timeout);
   if (...response is an error...) {
       send(response[0].error.description, error);
   }
}
function wrapup() {...}
    // ]]>
  </script>
</class>
```

Fig. 2. Sketch of an accessor definition file for a Philips Hue lightbulb.

requests. They can be callbacks or "push" notifications. An accessor need not even have any inputs; it can instead spontaneously produce outputs.

A. *Examples of Accessors*

To help make the idea concrete, consider a simple accessor sketched in Figure 2. This defines a class that extends a base class called `JSAccessor`, provided by organization `terraswarm.org`. This base class provides a JavaScript environment in which the accessor executes.

The accessor in Figure 2 has several inputs, some of which specify a bulb and bridge (a gateway providing Internet access to the bulb), and some of which specify commands to the bulb. C4PO, the robot described above, might use such an accessor to illuminate its surroundings to make another accessor, providing images from a video camera, more effective. The accessor specification, the XML text in Figure 2, is downloaded from an accessor library by a **swarmlet host**, which might be C4PO, a computer, a handheld device, an embedded device, or a virtual machine. The swarmlet host must trust the base class, `JSAccessor`, which is implemented locally, but it need not trust the accessor itself. The base class will sandbox execution of the script.

The script inside the `<script> ... </script>` element defines three functions, one of which is shown. The `fire` function reads commands from the accessor inputs, constructs a JSON data structure, the structure of which is specific to the Philips Hue, and sends it to the gateway using an HTTP PUT. Although the use of HTTP PUT is standard in RESTful interfaces, the syntax of the command is not, and using accessors, this syntax need not be standardized.

The swarmlet host **instantiates** the accessor, provides it with inputs, and **invokes** it whenever it chooses. In this case, "invoking" the accessor means executing the specified `fire` function in a JavaScript context that provides a way to read to the accessor inputs (the `get` function in Figure 2), a way to send outputs (the `send` function), and other facilities such as the `httpRequest` function. The use of such a function is given as part of the accessor interface by the `requires` entry. A swarmlet host needs to provide this function to be compatible with this accessor.

In this example, the particular underlying RESTful service does not require authentication, and neither the request nor the response is encrypted. Yet both authentication and encryption may be required by an accessor-server pair. The mechanism by which the accessor communicates with a server is up to the service provider and only factors into the interface between the accessor and the swarmlet host through "required" capabilities, which may be very low level (network access using UDP, bluetooth radio, HTTP access to the Internet, etc.). This separation of concerns is central to the concept of accessors.

There are many interesting variants of this example. An accessor could spontaneously provide outputs whenever the data satisfy some criterion, e.g. whenever a sensor value moves by a specified delta. Or it could provide historical data over some time range. The data provided by an accessor or commands to it could be time stamped, for example to leverage synchronized clocks. Accessors could provide database accesses, time series data, data analysis, actuation and control, and interactive services, at least.

B. *Implementation*

An accessor is instantiated and executed by a **swarmlet host**. A simple swarmlet will consist of only one such host, but a more interesting one may have many such hosts that interact through swarmlets (they function as a distributed swarmlet).

The example in Figure 2 deliberately uses industry standard notations and mechanisms for untrusted code (XML and JavaScript). The base class, `JSAccessor`, which provides the context in which to parse the XML and execute the JavaScript, is installed on the swarmlet host, and is presumably trusted. It can provide, for example, a sandbox environment for executing the JavaScript, much as done in modern browsers.

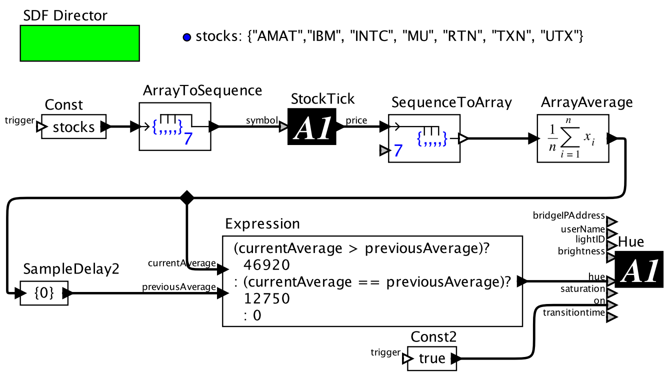

Fig. 3. Composition of accessors in Ptolemy II.

The simplest implementation of a swarmlet host that could invoke the accessor in Figure 2 is an ordinary browser, provided, via a web page, with a JavaScript implementation of the swarmlet host. We have prototyped such a host. It instantiates accessors and constructs a web page where their inputs are rendered as entry boxes in an HTML form. A "fire" button provided on the HTML page fires the accessor, upon which any output is displayed on the HTML page using standard AJAX mechanisms.

We have also prototyped a swarmlet host in Node.js (also JavaScript, but independent of a browser), and in Ptolemy II [6], which is written in Java. Ptolemy II provides a graphical editor for composing accessors and operating on their input and output data. Figure 3 shows a whimsical example of a swarmlet realized in Ptolemy II. This swarmlet uses an instance of another StockTick accessor to read a set of stock values from a cloud-based service (top A1 icon labeled "StockTick"). It then averages the prices, and if the average is greater than the previous average, it sets the color of a Hue light bulb to green using an instance of the accessor in Figure 2. (bottom A1 icon labeled "Hue").

A key issue is the semantics of composition of accessors. In Figure 3, the composition semantics is a particularly simple variant of dataflow known as synchronous dataflow (SDF), executed periodically with some specified period. A more intelligent swarmlet would use a time-based MoC such as discrete events (DE) to activate the light only when markets are open, and use a third accessor to activate the light only when the room in which the light is located is occupied. Note that the occupancy sensor, the light bulb, and the stock price service could be provided by three distinct vendors and use three distinct protocols for access to their respective devices or services. But they are easily composed, and the swarmlet designer chooses the composition semantics according to his or her needs.

The server side for the accessor in Figure 2 is a gateway box on a network for which an accessor instance must have

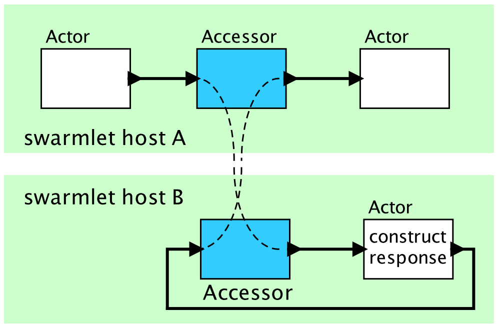

Fig. 4. Peer-to-peer accessors.

an IP address. Alternatively, the server side could itself be implemented as an accessor, as shown in Figure 4. Here, swarmlet host B provides the "service" for the accessor on swarmlet host A. An input on A is an output on B, and vice versa. When the accessor on A receives an input, it sends a request to B (by whatever mechanism the accessor defines, e.g. an HTTP get). The accessor on B produces this request as an output. The Actor on B constructs a response to this request and sends it back to its accessor as an input. The accessor then sends the response back to A (by whatever mechanism, e.g. a response to the HTTP get), whose accessor produces that response as an output. This **peer-to-peer** style of accessor interaction suggests that swarmlets can be constructed as large networks of interacting accessors, and that gateways need not be vendor-locked, an untenable situation that will lead to a large proliferation of gateways. The peer-to-peer nature of this architecture contrasts starkly with the centralized composition mechanism provided by IFTTT, for example.

V. Compatibility of Accessors and Hosts

To be useful, a swarmlet host needs to be able to use and compose a variety of accessors. However, accessors can get quite sophisticated, some swarmlet hosts will be constrained devices, and the particular security implications of running swarmlets may require different gradations of trust. Hence, a given swarmlet host may not be be able to instantiate all accessors. A well-defined interface will ensure that compatibility can be statically checked.

A. Accessor Interfaces

An accessor interface defines the inputs, outputs, and required capabilities (such as network access). Such an interface enables search, discovery, and service composition. In Figure 2, the interface is given in XML, and tools checking for compatibility need only check the XML. In particular, the body of the `<script>` element need not be checked, because if, at run time, it attempts to invoke capabilities that it has not declared "required," the swarmlet host can reject the execution.

Here is where standardization would become valuable. We envision a very basic standard that is extensible through "requires" tags and references to ontologies for sensors, actuators, and data. Many issues will need to be addressed in a standardization process, including the type system used for inputs and outputs and version management, for example.

B. Component Accessors

A **component accessor** has an interface and a script, as shown in Figure 2. The script defines one or more functions that are invoked by the swarmlet host. For example, in Figure 2, the script defines a `fire` function. In this case, "standardization" is provided through definition of the accessor base class, which is `JSAccessor` in Figure 2. That base class supports JavaScript scripts, and invokes specific procedures defined in the scripts, like `initialize`, `fire`, and `wrapup` in Figure 2. A different base class might use a different scripting language and/or a different set of functions that it invokes.

We envision other base classes, e.g. `PythonAccessor`, but in our experiments, we have focused on JavaScript because of its well-established use in browsers and its relatively well-understood security risks. We also envision component accessors whose scripts require specialized extensions in the swarmlet host, for example to access local hardware resources. Such accessors may require that the swarmlet host "install" the extension, where "installation" is an explicit expression of trust. An installation process allows for a careful assessment of the origin and function being installed, much like installation of software on personal computers today.

A component accessor is executable on any swarmlet host that includes an implementation of the accessor's base class and specified required extensions. The base class effectively defines a standard for a family of accessors. It also significantly reduces security risks by binding particular functions and variables usable by the script, as done today in browsers.

C. Composite Accessors

A **composite accessor** is an an accessor composed of other accessors on a single swarmlet host (in contrast to the peer-to-peer architecture of Figure 4, where a swarmlet is distributed across swarmlet hosts). The composition connects outputs of one accessor to inputs of another and defines an execution policy for the component accessors. Specifically, when the composite accessor is invoked, how should the component accessors be invoked? Whatever mechanism is used, the composition must itself define an accessor. One possibility is for the swarmlet host to implement a **model of computation** (MoC) that governs the execution and communication between accessors. The swarmlet host realized in Ptolemy II is already capable of this, using Ptolemy directors [6]. A second possibility is for the accessor to script the composition of accessors in some scripting language (e.g. JavaScript) where the capabilities of the language are sufficiently restricted to ensure safe execution. In this case, the accessor itself defines the model of computation that binds the component accessors.

Possibilities for MoCs that can be implemented by either of these mechanisms (directors installed on the hosts or scripts provided by the accessors) include at least the following:

- Imperative programming, giving a sequence of actions.
- Pipe-and-filter, for example a dataflow model.
- Process networks (deterministic Kahn networks, MPI, or Scala-style nondeterministic message passing).
- Publish-subscribe, of which there are many variants.

- Event notification (like the callbacks in Node.js).
- Discrete events (with locally defined time stamps, or with globally defined time stamps relying on clock synchronization).
- Synchronous-reactive models and their asynchronously executed variants.

Note that for untrusted accessors, it may be important to choose a less expressive MoC, where a composition can be statically analyzed for memory usage and deadlock, for example. For a discussion of why less expressive mechanisms improve security, see [8].

VI. Models of Computation

Composability is essential to the concept of accessors. But what should be the semantics of composition? The interface we have chosen for accessors is deliberately designed to enable disciplined concurrent composition. Specifically, the accessor interface we define conforms to the **actor abstract semantics** [3], which generalizes the classical notion of actors [2] to embrace a richer set of concurrency models. In particular, such interfaces have been shown to be compatible with dataflow, process networks (PN), publish-and-subscribe, discrete-event (DE) systems, synchronous-reactive (SR) systems, and rendezvous-based models. All of these MoCs have been implemented using the actor abstract semantics in the open-source Ptolemy II system [6].

A key point is that *the same accessor definition* may be useful in multiple MoCs, a concept that we call **domain polymorphism** [6]. The notion of "invoking" an accessor is a key part of the actor abstract semantics. The script given in Figure 2 defines an **action method**, `fire`. In dataflow, this method is invoked when there is sufficient input data for the accessor. In discrete events, a firing occurs when one or more inputs of the accessor have an event with an earliest time stamp. In synchronous-reactive models, `fire` is invoked at ticks of a conceptual global clock. In process networks, actors fire asynchronously, each in a separate thread of control.

The actor abstract semantics defines a few other action methods. The `initialize` method will be invoked when the swarmlet starts executing (or if it gets reinitialized during execution). The `wrapup` method is invoked when the swarmlet stops executing due to either normal termination or an unhandled exception. In addition, we will need a mechanism for an accessor to request a future firing. Our `JSAccessor` base class provides a `timeout` function in the JavaScript context for this purpose.

Using a timed concurrent model of computation such as discrete events to run accessors offers some interesting possibilities for composition of accessors. It is common when using a callback style of concurrency, such as in Node.js, to use timeouts to manage concurrency [5]. However, timeouts are difficult control, because they have weak semantics in JavaScript. If you want two actions to be performed later in a specific order, setting a longer timeout for the second action does not ensure that it executes later. Moreover, there is no notion of simultaneity nor causality, so it is difficult to ensure that two actions are executed in time to simultaneously provide inputs to a third action. One may achieve these effects with large differences in timeouts, but there are no guarantees. A timed MoC like discrete events, however, provides deterministic timed concurrency [6]. Even more interesting is to leverage network clock synchronization to get distributed services with deterministic composition semantics.

The key is that for the composition of accessors, what concurrency model to use, if any, and whether to use a timed model, is determined by the swarmlet, not the accessors. The provider of an accessor, which may be the vendor of a IoT device or a third party, need not have any idea that the device will be used in a distributed real-time system that exploits a sophisticated coordination mechanism. And there is no need to standardize on that coordination mechanism, because many swarmlets will not need it.

VII. Discussion

This paper offers a starting point for a very rich mechanism. Because of the ability to download and compose accessors, create new accessors, and construct swarmlets out of peer-to-peer communicating accessors, the usage of code mobility is more sophisticated than conventional client-side code evaluation in a browser; web applications do not have a general mechanism for composition.

This raises some interesting and challenging research questions, many of which we left untouched. For instance, how should accessors be published, registered, searched for, and discovered? Some accessors may need to be signed to establish trust. To what extent can we use static analysis to ensure some measure of safety regarding accessor scripts? How constrained does the context in which scripts are evaluated need to be? What services should be provided for encryption and authentication? Most accessors require interaction with a persistent process or service. How should the accessor handle discovery of that process or service? What should the life cycle of those persistent processes look like? How do we deploy a swarmlet in the Cloud? Could a swarmlet migrate from one host to another? How should versioning of accessors be managed? A great deal of work remains to be done.

References

[1] Fielding, R. T., and Taylor, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology 2*, 2 (2002).

[2] Hewitt, C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence 8*, 3 (1977).

[3] Lee, E. A. et al. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers 12*, 3 (2003).

[4] Lee, E.A. et al. The swarm at the edge of the cloud. *IEEE Design and Test of Computers To Appear* (2014).

[5] Pasquali, S. *Mastering Node.js*. Packt Publishing, 2013.

[6] Ptolemaeus, C., Ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[7] Rabaey, J. M. The swarm at the edge of the cloud: The new face of wireless. In *Proceedings Symposium on VLSI Circuits* (2011).

[8] Sassaman, Len, e. a. Security applications of formal language theory. *IEEE Systems Journal 7*, 3 (2013).