RIAPS:Resilient Information Architecture Platform for Decentralized Smart Systems

Scott Eisele Istvan Madari Abhishek Dubey Gabor Karsai
Institute for Software Integrated Systems, Vanderbilt University Nashville, TN, 37211

***Abstract*—The emerging Fog Computing paradigm provides an additional computational layer that enables new capabilities in real-time data-driven applications. This is especially interesting in the domain of Smart Grid as the boundaries between traditional generation, distribution, and consumer roles are blurring. This is a reflection of the ongoing trend of intelligence distribution in Smart Systems. In this paper, we briefly describe a component-based decentralized software platform called Resilient Information Architecture Platform for Smart Systems (RIAPS) which provides an infrastructure for such systems. We briefly describe some initial applications built using this platform. Then, we focus on the design and integration choices for a resilient Discovery Manager service that is a critical component of this infrastructure. The service allows applications to discover each other, work collaboratively, and ensure the stability of the Smart System.**

I. Introduction

Emerging Trends: The emerging Fog Computing paradigm provides an additional computational layer consisting of distributed computation and communication resources that can be used to monitor and control physical phenomena close to the source. It can also be used for fine-grained data collection, and filtering before sending the data to a cloud service. Examples of these Fog Computation platforms include SCALE [1] and Paradrop[2]. However, while, the concept of Fog Computing is promising, a number of challenges exist that must be addressed. One of the foremost challenges is providing a stable environment application development and deployment despite the dynamism, heterogeneity, and increased failure potential of computing resources at the edge which do not operate in data centers or some other controlled environment.

A solution to this problem is a universal computing platform, which provides the core services necessary for a stable deployment environment. Services like time synchronization, distributed data management and coordination, service discovery, and mechanisms to deploy and remotely manage the distributed applications.

RIAPS: Our team is developing the core architecture, algorithms and programming paradigms for such a computing platform called RIAPS (Resilient Information Architecture Platform for Smart Systems) [3]. The pivotal concept of the Smart Applications is the distribution of intelligence throughout the infrastructure. For example, in the smart grid domain, increasingly companies, communities, and even some customers (or prosumers) are becoming managers of power. This requires monitoring, control, and management software applications at all levels to do their work. The centralized, control-room oriented paradigm is not sustainable, as it does not scale. Rather, a *decentralized* paradigm is required where interacting software programs deployed on devices across the network solve problems collaboratively. This is also true for distributed traffic control where each intersection controller must coordinate with other controllers based on contextual and local information [4].

Innovation: This paradigm is very different from what is being used today. In today's systems, data is collected locally and transferred to a central server or control room where control decisions are made and control commands are generated. These commands are then sent back to local controllers, and actuators. This architecture incurs long round-trip times, delayed decisions, and does not lend itself to the needs of future edge applications [5] like energy management [6]. The distinguishing characteristic of RIAPS is the completely decentralized computing model: software applications are distributed across a multitude of compute nodes on a communication network, and each node has access to local measurements and actuators. An application consists of components that run in parallel on a collection of nodes. The functionality of an application is realized by the network of interacting components managed by actors. This computation architecture is an extension of the F6COM computation model [7] and [8]. The specific extensions are related to the discovery services and the platform services that we discuss in the next section.

Contributions: The contribution of this paper is the architectural description of RIAPS (section II), a demonstration with a development version of the platform implementing a traffic control example (section II-B) showing some results comparing the effectiveness of traffic control with distributed coordination compared to no coordination. We also briefly discuss a microgrid application example on the platform. Using these examples we motivate the need for a robust decentralized discovery service (section II-D) as a critical service for the resilience of the platform. Thereafter, we discuss the design and implementation of the discovery service using a distributed hash table (section III). We finally present experimental results (section IV) on some discovery service metrics, and compare our work with the state of the art (section V).

II. The RIAPS Computation Architecture

The goal of the RIAPS run-time system (see Fig 1) is to provide a software foundation for building distributed applications. It relies on an underlying operating system and includes

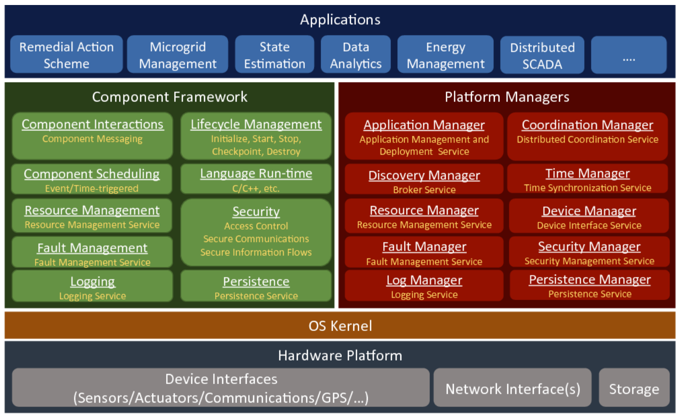

Fig. 1. RIAPS Run-Time System Architecture

two major ingredients: (1) a Component Framework, and (2) a suite of Platform Managers. The Component Framework is instantiated as a set of software libraries that are (dynamically) linked with the application components, while the Platform Managers are specialized operating system processes, implemented as daemons in the Linux systems. These two ingredients provide the services that will be used by a developer in supporting the implementation of the application logic. The Component Framework layer is where the implementation of the various middleware libraries reside. The goal of the Component Framework is to provide higher-level abstractions for building complex, resilient, distributed applications on the platform. The middleware libraries include the component scheduler (which implements the component execution semantics), the component interaction library (that enables publish/-subscribe and remote method invocation on the same node or across the network. Having a formal interaction semantics provide additional reasoning capabilities as shown in [9]). Furthermore, the framework provides support for lifecycle management support (that assists in remotely managing the software components), the language run-time libraries, the resource management support (to monitor computing platform resource utilization/availability), the fault management support (that detects and mitigates anomalies in software components), the security library (for secure communication), the logging library (to record component events), and the persistence library (to allow the persistent storage of data). These libraries are linked with the components used to create an application.

The Platform Managers layer includes the elements of the application framework: the various platform services that run as independent processes and implement system-level management capabilities. The services include the Application Manager (that enables remote installation and management of the applications), the Distributed Coordination Manager (that implements fault-tolerant distributed service like leader election, consensus, coordinated actions, etc.), the Discovery Manager (which determines available connections among components on the same node or other operating nodes), the Time Manager (that provides high-precision timing and time synchronization services), the Resource Manager (monitors computing resources to ensure components and Platform Managers are able to run concurrently), the Fault Manager (that provides node-level fault management services), the Device Manager (that supports access to and management of attached input/output devices), the Security Manager (that handles authentication and manages keys and digital signatures), the Log Manager (that serves as a single entry point to all log activity on a node) and the Persistence Manager (that provides non-volatile data storage facility).

The applications reside in the top layer (see figure 1) and they rely on the the services provided by the Component Framework and Platform Managers. One application consists of one or more application managers, called *actors*, which are deployed on computing nodes. Each actor hosts one or more application components that interact solely through the middleware interactions and rely on the available platform services. The advantage of packaging multiple components into one actor is that the cost of communication between components in one actor is much smaller than across actors running on the same node. The communication between actors running on different nodes is even more costly, as the messages have to go through a complex protocol stack and a (potentially unreliable) network.

A. *Component Architecture*

A RIAPS component is a reusable unit of software that implements a set of operations for manipulating its state, and ports through which it communicates and interacts with other components. A special port, called the timer port is also available. It enables time-based triggering of the component. The timing of the RIAPS component is controlled by the Time Manager service that provides high-precision timing and time synchronization services. This service is is not fully implemented yet and will be discussed in a future work.

The operation of a component is analogous to a typical computer process in the sense that each component is limited to a single thread of computation. This thread is managed by a trigger method which is provided by the developer of the component. The trigger method monitors the state of the component and launches operations when 1) the state of the ports change, 2) a timer expires, or 3) an operation is completed. These operations implement the application logic of the component. The ports on the component are determined by the desired communication patterns which include asynchronous request/response, synchronous client/server, and publish/subscribe. Ports are assigned a message type and when an application is deployed, the message types represent the services provided or requested by the corresponding port. A special component, called the device component has the same attributes as application components however it may have multiple threads of execution to handle interfaces to physical devices.

To run an application the components are deployed on computation nodes. The components on a particular compute node are managed by actors. An actor provides its components with the run-time code as well as the interfaces necessary to

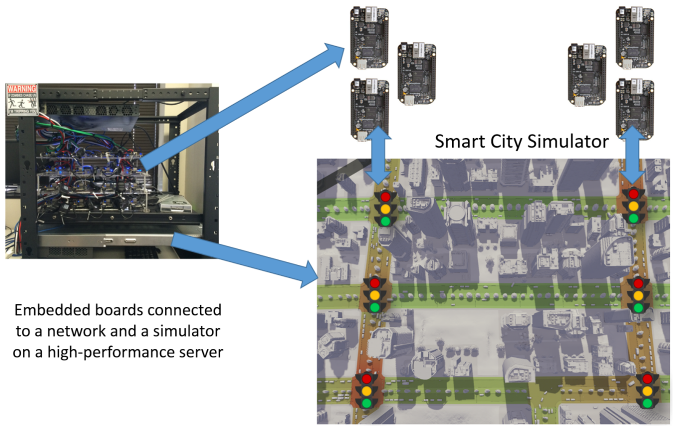
Smart City Simulator
Embedded boards connected to a network and a simulator on a high-performance server

Fig. 2. Testbed

access platform services. Additionally the actor provides the capabilities to control and configure its components remotely. This is required to ensure that all the components of an application can be installed and configured correctly. The actor is responsible for loading a component, setting up its configuration, and initializing its state.

B. Traffic Controller Example

In order to experiment with the RIAPS framework we developed a traffic controller example. The example involves a city simulation where the traffic lights in each intersection are controlled by a traffic controller application implemented with the RIAPS platform running on embedded single board computers. The simulation sends simulated "sensor data" to the intersection controllers consisting of the traffic density for the incoming road segments, as well as the current state of the traffic lights. Each intersection controller shares this information with its neighboring controllers, and each uses the information to estimate the traffic incoming on each segment. This information is used to change the state of the traffic light with the objective of improving the flow of traffic.

The testbed for this example can be seen in Figure 2. It consists of a 32 BeagleBone Black [10] cluster connected through an Ethernet switch to a computer running Cities:Skylines [11]. This game was chosen because it has the capability to simulate the movements of hundreds of thousands of citizens, and it has a rich game modification API with an active community. This allowed us to modify the game to be able to control the traffic lights with our embedded controllers.

The RIAPS application created for this test scenario includes an intersection controller, a light interface device, and a density sensor device whose implementation can be seen in listings 1, 2, and 3 respectively.

Listing 1. Intersection Controller Component
```
// Intersection Controller component.
component IC (parent="none"){
  timer clock 1000;
  sub densityPort : gameDensityMsg;
  sub lightPort : gameLightStateMsg;
  pub pubICPort : ICDensityMsg;
  sub subICPort : ICDensityMsg;
  req setLightsPort : (setLightReq, setLightRep);
}
```

Listing 2. Interface to Light device
```
device LightIF(rate=10, gameServerIP="host", parent
    ="none") {
  inside trigger /* default */;
  timer clock 1000;
  pub lightPort : gameLightStateMsg;
  rep setLightsPort : (setLightReq, setLightRep);
}
```

Listing 3. Interface to Density sensor device
```
device DensitySensor(rate=10, gameServerIP="host",
    parent="none") {
  inside trigger /* default */;
  timer clock 1000;
  pub densityPort : gameDensityMsg;
}
```

The controller has 3 subscriber ports, a publisher port, a request port and a timer. Two of the subscribers, *lightPort* and *densityPort* are for reading sensor data from the game. The *lightPort* has the message type of gameLightStateMsg and is connected to the publisher port with the corresponding message type in the light interface device when the application is deployed. The *densityPort* is essentially equivalent. Each intersection controller publishes the sensed density data at intervals determined by the firing of the timer, and subscribes to the density data published by the adjacent intersections. The controller implemented in this example is fairly simple in that the state switching occurs based on thresholds. The thresholds ensure that a light remains in a particular state for some minimum time but no longer than some maximum time, similarly there are minimum and maximum density thresholds.

These components and devices of the application are contained in an *actor*. The actor is responsible for managing the components and devices and providing the interfaces to platform services. As can be seen in listing 4 the actor contains both devices and a controller and is used to specify several parameters such as the IP address of the simulation machine. The actor registers the ports of the components with the Discovery Manager and if a matching message type is already known to the Discovery Service the client will retrieve that information and return it to the actor, which then sets up the connections between corresponding components.

Listing 4. Actor
```
actor Actor0 {
  local gameDensityMsg, gameLightStateMsg,
      setLightReq, setLightRep;// Local message
      types
  {
    ic : IC(parent="Actor0");// Intersection
        Controller
    LIF : LightIF(rate=2, gameServerIP
        ="192.168.0.107", parent="Actor0");// Light
        interface device
    dsnsr : DensitySensor(rate=2, gameServerIP
        ="192.168.0.107", parent="Actor0");//
        Density sensor device
  }
}
```

20s Timer

	IC0	IC3	IC2	IC1
Seg0(S) means:	21.31	16.37	11.62	30.89
Seg1(N) means:	17.23	15	8.41	24.22
Seg2(W) means:	14.84	13.3	16.09	12.33
Seg3(E) means:	15.54	10.06	18.65	24.77
Intersection avg	17.23	13.68	13.69	23.05

Densities

	IC0	IC3	IC2	IC1
Seg0(S) means:	15.69	15.93	10.84	20.09
Seg1(N) means:	15.47	11.81	9.6	18.74
Seg2(W) means:	11.15	9.31	11.22	8.43
Seg3(E) means:	8.89	6.2	8.44	16.72
Intersection avg	12.8	10.81	10.03	15.99

Share Data

	IC0	IC3	IC2	IC1
Seg0(S) means:	19.09	18.38	11.78	22.47
Seg1(N) means:	14.12	12.59	7.59	16.61
Seg2(W) means:	9.39	5.68	7.04	8.6
Seg3(E) means:	10.28	6	9.06	15.23
Intersection avg	13.22	10.66	8.87	15.73

Fig. 3. Mean traffic densities for each segment of each intersection controller.

1) Experimental Results: The traffic controller implementation was run comparing the densities of the segments when running the controller with 1)only timer switching logic, 2)each controller checking its own density data and 3)each each controller sharing data with its neighbors. The average segment densities for these tests can be seen in Figure 3.

We see from these initial experiments that having the controllers aware of their own densities decreases segment density, and sharing that information improves the situation slightly. In this study the densities were collected from the game by querying the road segments surrounding the traffic lights. In an actual implementation this will not work unless some sensor is installed at each road segment. Another option to obtain the data is for the cars themselves to publish their positions and routes to the cars around them and that information is then shared to the the intersection controller. This way the controller does not have to guess how much traffic is coming on each of its segments. In addition if vehicles are publishing their data, emergency vehicles may also publish this information and controllers switch to prioritize the emergency vehicle. In order for such a system to be realistic it is critical that the vehicles and lights are aware of each other. This motivates the need for a discovery service which is able to quickly capture and share ingress and egress information to intersection controllers as vehicles come and go.

C. Microgrid Example

Another application for the RIAPS platform that is currently under development is a decentralized controller for microgrids As power requirements change or faults occur segments of a connected grid may break off and become islands. It is necessary that this information propagates quickly to the controllers to ensure smooth operation and continuous service. Similarly, when the islands re-connect to the main grid it is necessary for the components to discover each other and share resource information. The objective is to handle the transients between these states in clusters of controllers rather than at a centralized location as this will improve scalability as well as fault tolerance.

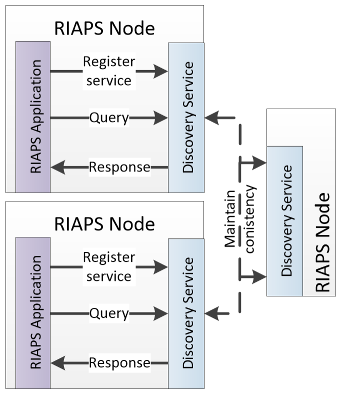

Fig. 4. RIAPS Discovery Service Infrastructure

D. Requirements for the Discovery Service

In both use cases the set of member nodes can change over time. For example, in a microgrid application, homeowners can choose to disconnect themselves from their local photovoltaic grid and transfer themselves to the main utility grid. Similarly, in the traffic controller example the lights can get disconnected due to failures. Furthermore, the same system can be extended to create a traffic priority system, where the emergency vehicles entering an area can communicate with the controller and can disengage when they exit the area. Given these two use cases it is easy to see that the network of communicating entities must be able to (a) know when new nodes join the group and (b) know when nodes leave the group. Furthermore, they must know when applications (and their components) come and go - the Discovery Service is expected to keep track of the state of the applications' services and message types. This service has to be distributed and fault tolerant. A centralized implementation is insufficient, as it does not scale and it can be a single point of failure. Fault tolerance is needed as any node or communication link can fail unexpectedly. These local failures must not result in system-wide collapse. Hence, the Discovery Service must be available on each node, and these instances need to share their state - as needed - across the network.

III. Discovery Service

RIAPS aims to provide modular, decentralized solutions for each service comprising the platform, so they can be used in other applications. Therefore, the *Discovery Service* runs on each node as an independent process and listens for messages from the local RIAPS applications, and from other nodes with a Discovery Service. Figure 4 shows the main features of the service discovery: RIAPS applications register app services by providing the related details to the discovery service(message types, communication protocols, IP addresses and ports). The Discovery Service stores the app service details and forwards the new information to the neighboring nodes. When a RIAPS application requests an app service, it queries the local Discovery Service and the results are asynchronously sent to the RIAPS application.

The Discovery Service relies on OpenDHT [12] to store, query, and disseminate the service details through the network. OpenDHT is a fast, lightweight Distributed Hash Table (DHT) implementation. The dissemination does not mean full data replication on all nodes, OpenDHT stores the registered value locally and forwards it to a maximum of eight neighbours. Note that usage of distributed hash table for service discovery does not distinguish the nodes, (i.e. there are no "server" or "client" nodes) – nodes are peers and each operates with the same rules. If a node disconnects from the network, the Discovery Service on other nodes is still able to register new services or run queries. If a new node joins the cluster, the values stored in the network are available to the new node. For this approach to flexibly handle node ingress and egress it is necessary for the Discovery Services to find each other. Note that there are two major cases for network configuration: (i) the nodes are on the same local subnet or (ii) the nodes are on different subnets of the network.

To find the available Discovery Service managers on the same local subnet the RIAPS framework uses UDP-beacons. Periodically, each Discovery Service instance announces (via IPv4 UDP broadcast) its network address and listens for incoming beacons. These UDP packets are sent and received asynchronously and the Discovery Service managers maintain the list of known addresses. Before a UDP packet is processed by the Discovery Service the received beacons are filtered to remove the non RIAPS-specific UDP messages. These messages function as a heartbeat. If no messages are received from a known node during two time periods, then the Discovery Service removes the silent node from the list of peers. When a UDP beacon arrives from a new node the Discovery Service stores the address of the new node in OpenDHT, which then adds it to the list of known nodes.

Unfortunately the nodes in another subnet cannot be discovered by UDP broadcasting; the remote addresses must be passed explicitly to the Discovery Service. In this case, we rely on designated gateways running the Discovery Service with IP addresses of the other subnet (assuming that routing is available between the subnets).

A. Handling the ingress and egress scenarios

In the previous section we mentioned that the DHT-based service discovery forms 8 node clusters to share application service registration data, but we did not discuss how the stored data is used in RIAPS. When a RIAPS application starts, it registers its services in the Discovery Service. The Discovery Service stores this information in the DHT, and the DHT propagates the new information through the cluster.

In the startup phase a RIAPS application not only announces the provided services, but subscribes to needed services. If a compatible service is already in the DHT, the Discovery Service sends a notification to the requesting RIAPS application. The application processes the newly arrived notification and connects to the service. If the desired service is not available the Discovery service will issue a callback to the requesting application when it becomes available.

OpenDHT does not provide an API to remove a service from the DHT. Instead a service may be removed by setting an expiration value (the default being 10 minutes). After this time the service is removed from the DHT. It also means that value must be renewed periodically by the Discovery Service if the application is running. Therefore, when a service stops responding the manager does not remove it from the DHT until the current registration expires.

B. Fault Tolerance

The Discovery Service is responsible for renewing the registration of application services in the DHT. Renewal is necessary, as the stored values are otherwise removed. Before renewal the Discovery Service must check that the RIAPS component service to be renewed is still running and available. This means that the Discovery Service must handle the case when an application service leaves the cluster abruptly, e.g. it stops without sending a message.

We are currently implementing the next version of the discovery service in which the service information is paired with the *Process ID* (PID) of the actor. Namely, when an application component registers a service then the PID of the parent actor is also registered with a time-stamp in the Discovery Service. The list of service/PID pairs are verified periodically by checking if the PID is still running. If the process has stopped, the Discovery Service removes the pair and does not renew the registration at the next DHT refresh point.

The components must be resilient as well, since the Discovery Service could stop unexpectedly. If the Discovery Service fails, the components and actors continue, but cannot receive notifications about new services, and new actors cannot be started (until the Discovery Service restarts).

Since the components are managed by actors, the components do not implement any discovery checking algorithm. The connection with the Discovery Service is maintained by the actors. The approach for the actor checking Discovery Service liveness is the same as Discovery Service checking components. The actor knows the PID of the Discovery Service and maintains a time stamp. If the PID of the Discovery Service is not in the list of the running processes, the actor starts a re-initialization process. Reinitialization means, that the actor re-registers the running RIAPS services in a new Discovery Service instance and subscribes to the services needed. If the discovery service dies, the actors are informed and they re-register to recreate the state within the discovery service.

IV. Tests for the Discovery Service

To test the discovery service we ran a few tests. The first was to initialize all but one node as subscribers. Once ready,

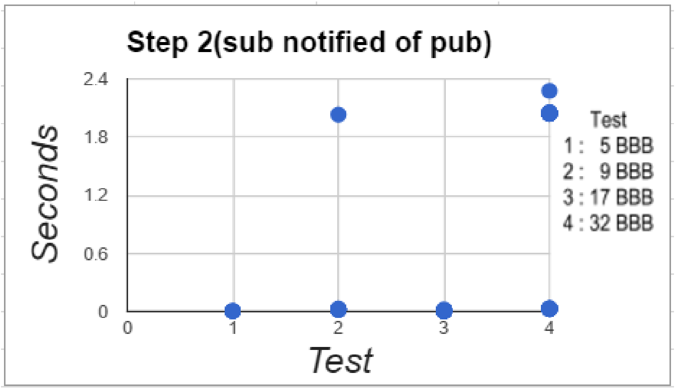

Fig. 5. This figures shows the time between when the publisher registers with the Discovery service and the time when the subscribers receive notification of that service for the tests involving 5, 9, 17, and 32 beaglebones. Several data points were very close to each other, and so overlap in the plot. The ranges of the overlapping nodes are Test 1: 4 nodes(1-7ms), Test 2: 7 nodes(18-32ms) 1 node(2.032s), Test 3: 16 nodes(2-23ms), Test 4 : 12 nodes(27-35ms), 18 nodes(2.041-2.056s), 1 node(2.276s)

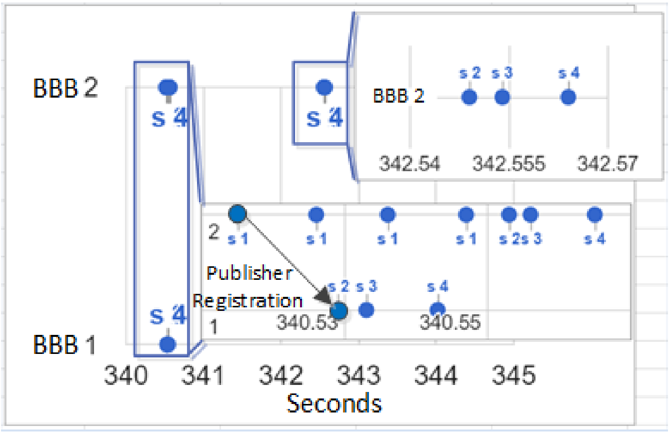

Fig. 6. The plot here has two Beaglebones on the y axis. The points denote the events (s1-s4) for each Beaglebone when BBB2 rejoins the cluster. The two zoom in boxes show the events in detail as they happen on timescales on the order of milliseconds. At the beginning of this test two nodes were started. After some time BBB2 was rebooted. In the zoomed box between times 340 and 341, the first event shown is s1 for BBB2 when it registers its publisher service. The s2 event for BBB1 is when BBB1 receives the notification of BBB2's service 14ms after BBB2 registers it. 4 and 10ms later BBB1 notifies it's requesting component and that component connects to the publisher respectively. Towards the end of the first zoom in box we see s2, 3, and 4 events when BBB2 receives notification of its own service. The second zoom in box between 342 and 343 seconds shows BBB2 again receiving notification of its own publisher. Covered by the second zoom in box at 344.5 seconds BBB2 finally receives notification of the publisher on BBB1. There is some behavior where a new node receives notification of the first service it discovers twice.

the final node was added as a publisher service. This provides events for measuring service propagation time through the cluster. The node clocks are synchronized with NTP, making measurements across the cluster meaningful as their timestamps are within 300ms of each other. This test is relevant to the traffic light example because as vehicles move through the city they enter and exit new clusters which collectively transmit their density to the traffic controller allowing the controller efficiently route traffic.

There are several measurements taken for this test. The first is the “Register service" step (s1) where each node informs the Discovery Service of which services it provides and which it requires. As mentioned this occurs first for all of the subscribers, then the publisher. The Discovery Service stores which service the subscriber is interested in, and when that service is available the Discovery service sends a message to the component's actor, this is the second measurement (s2). We assume the difference between this time and the publisher service registration time to be the time needed to propagate a new service through the DHT. The third time-stamp (s3) is when the actor notifies the corresponding component of the new service. The final time-stamp (s4) is when the subscriber is connected to the publisher. These events can be visualized using Figure 4 as a reference. The duration of steps 1 and 2 are the time stamp of the publisher registration and the subscriber s1 and s2 timestamps respectively. The duration of step 3 is s3-s2 and the duration of step 4 is s4-s3.

The test was run utilizing Zopkio [13]; a testing framework. We can see the result for step 2 in Figure 5. The time between the first subscriber registration and the publisher registration (step 1) is linear as nodes increase due to the implementation of deployment in Zopkio as a sequential process. Steps 3 and 4 take between 5 and 12ms.

As the accuracy of the clock between nodes is about 300ms the 5, 9, and 17 node tests are indistinguishable, however the jump in time for the 32 node experiment suggests further tests are needed to verify scalability. The times for steps 3 and 4 after notification are not relevant for this service propagation test and are of short duration as we see in the next test.

The second test consists of two nodes each requesting and providing a publisher service. This means that each node subscribes to itself and all others. The test is to have one node exit and later rejoin the cluster. This is to verify that nodes recover and provide services reliably after egress events. The result of this test is shown and described in Figure 6

From this test we do see that when BBB2 rejoined all services were re-established.

V. Related Literature

Resource discovery is a critical aspect of distributed applications. Zookeeper, which was originally developed to provide a distributed, eventually consistent hierarchical configuration store [14] has been adapted to serve as a resource discovery service. The infrastructure of Zookeeper includes sending notifications to clients, so service discovery has been implemented with it. However it is difficult to deploy and maintain, it also prioritizes consistency over availability [15]. This means that in a network with nodes joining and leaving new applications will be blocked waiting for resources while the service discovery waits for consistency before allocating resources. For highly dynamic systems, like the traffic example, a paradigm that prioritizes consistency is fundamentally flawed when attempting service discovery. Responding to this need Apache released Apache Helix which resolves some problems [16] but the fundamental issues are still present.

Another tool developed specifically for resource discovery is Consul from Hashicorp [17] which is the industrial state-of-the-art. Consul provides several higher level features such as health checking and distributed configuration management. It relies on servers to store and replicate the resource discovery data. It is recommended to have several consul servers for fault-tolerance. With several servers one is elected as a leader, using Raft-based consensus, in order to guarantee consistency between servers. This means that Consul too has the problems associated with prioritizing consistency over availability because in a dynamic system a device chosen as a server may leave resulting in a lack of quorum. To combat this, it is possible to make every node a server but in high device density, or resource limited situations there are issues with data replication since every node will replicate the key/value store. An alternative is Serf, another tool from Hashicorp which is not as fully featured as Consul, but rather than using an always consistent model it has an eventually consistent model, prioritizing availability. Serf does not have a central server, making it more resilient. The problem with Serf is that it was developed for node discovery rather than service discovery and so would need extensions to provide the service discovery. In [15] the authors create a docker container for Serf calling it Serfnode and using it for service discovery.

Hoefling et al. in [18] present some extensions to the C-DAX [19] middleware. C-DAX is a middleware developed to be a cyber-secure and scalable middleware for the power grid. The authors do not discuss the security aspects of C-DAX but rather reference papers demonstrating these features. Scalability in C-DAX is achieved using a cloud and broker based publish/subscribe mechanism, making it more scalable than client/server patterns. The extensions presented by Hoefling are to address weaknesses in the C-DAX middlware with respect to interoperability with legacy applications such as SCADA, and low latency applications such as synchrophasor-based Real-Time State Estimation of Active Distribution Networks (RTSE-ADN). SCADA relies on bidirectional communication, so the authors implement a new client which consists of both a publisher and a subscriber to communicate with IP-based applications like SCADA using a tunnel-adapters and virtual network interfaces. The clients communicate using the C-DAX middleware. For low latency applications such as synchrophasor-based RTSE-ADN the authors present a method of connecting publishers directly to subscribers, without a broker reducing network traffic and the number of network hops required. The problems with this approach are those that impact all cloud-based systems. As devices increase so does cloud traffic, and latency. Therefore in the end edge computing will be necessary.

In our work on RIAPS, we are interested in similar issues regarding latency, however rather than relying on cloud based centralized databases and resolvers we use a Distributed Hash Table (DHT) to track the participants in the network and have the nodes discover one another. The removal of the cloud allows us to achieve single hop connections between publishers and subscribers as done in [18] but rather than needing to 1) send a join message to a Designated Node in the cloud which 2) queries the Resolver (the look up) for the address of the topic specific database, then 3) have the Designated node connect to the database and request the subscribers or publishers for the topic so that 5)the publisher or subscriber can update its connection rules we can simply 1)look up in our DHT Discovery Service for the message type we are interested in, 2)receive the address and 3)connect.

Data Distribution Service (DDS) is a “middleware protocol” and API open standard for data-centric connectivity" published by Object Management Group (OMG) [20]. There are many implementations of the DDS standard which vary according by developer. In order to promote interoperability between DDS implementations OMG introduced the Real-Time Publish-Subscribe (RTPS) protocol which was designed for DDS. The main features of RTPS[21] include fault tolerance, plug-and-play connectivity (allowing for dynamic ingress and egress with automatic discovery), capability to implement trade-offs between reliability and latency, scalability, modularity allowing constrained devices to run a subset of the standard[22] and still communicate with the network, and type-safety to prevent mismatched endpoints from connecting. However, the tremendous complexity of the DDS discovery service due to the several QoS options make it very difficult to use. Furthermore, the discovery service is tightly integrated with DDS and is not suitable for other platforms.

In [23], Cirani et al. present work on global and local service and resource discovery for the Internet of Things. To handle resource discovery the authors present an *IoT Gateway*. For local networking there are two ways a device can join a network. If it is aware of a IoT gateway it can join and send its resource information to it, or it can wait for a message broadcast from the gateway alerting the device to its presence. The addresses and resources of devices are added to the gateway which acts as a service look-up to the other devices in the local network and a service provider to external IoT gateways. For IoT gateways to discover and communicate with each other the authors present two P2P overlays. The first is the distributed geographic table (DGT). This table is similar to a distributed hash table but rather than the replicated data being based on hash value assignments the storage is based on the geographic location. This makes it deterministic. For a device to make itself known on a network it contacts a known gateway and shares its location to the DGT. The DGT shares this among the peers and informs the device of other gateways. Once the gateways are known, requests can be made for lists of services that can be accessed and this information is added to the distributed location service which is the other P2P overlay.

In [24] the authors present several important issues in IoT systems including standardization, mobility, networking and Quality of Service support. To address these issues they present an architecture which combines DDS with software defined networking (SDN). DDS is responsible for providing discovery and communication between heterogeneous devices within a domain. However DDS is for local networks. SDN is used to allow communication outside of the local network. By decoupling the control plane from the forwarding plane a SDN controller can provide network interfaces to the local network and when requests can not be filled locally it allows for forwarding rules to be defined, to pass messages to other networks. It is not clear from the paper how the SDN nodes find each other.

Compared to these solutions, one of the key benefits of our discovery service is that it is completely isolated and compartmentalized from the rest of the framework. In fact, we have seamlessly moved from an earlier Redis based discovery service (not described in this paper) to the DHT based discovery service mentioned in this paper as a drop-in replacement. This is due to the abstraction of register, query, and response interfaces as described in figure 4.

VI. Discussion and Conclusions

Fog computing provides new opportunities for distributed applications and analytics. However as the domain becomes more complex, tools are necessary to assist developers in creating applications by handling the implementation details. One of these details is service discovery.

Service discovery is an essential aspect of fog and edge computing particularly in dynamic environments. The current mechanisms for handling dynamic discovery are generally ill-equipped as they rely on a central server resulting in increased latency and a single point of failure, or they prioritize consistency over availability which can prevent application deployment if there is a fractured quorum. For highly dynamic discovery, which prioritizes utility over consensus, the only options do not include discovery of services. This means that service discovery would be an addition. From our initial experiments using distributed hash tables to provide a dynamic discovery service we see that it is tolerant to egress/ingress scenarios and is able to scale to at least 32 nodes.

We have shown some example applications on the RIAPS platform and demonstrated our prototype discovery service which allows for fault tolerant dynamic discovery. There is additional work to be done to verify and improve the capabilities of the platform but the outlook is promising.

Acknowledgment: The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000666 and in part by a grant from Siemens, CT. The views and opinions of authors expressed herein do not necessarily state or reflect those of the US Government or any agency thereof or Siemens, CT.

References

[1] K. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. DâĂŹarcy, D. Hoffman, M. Makai, J. Stamatakis et al., "Scale: Safe community awareness and alerting leveraging the internet of things," IEEE Communications Magazine, vol. 53, no. 12, pp. 27–34, 2015.

[2] D. Willis, A. Dasgupta, and S. Banerjee, "ParaDrop: A Multi-tenant Platform to Dynamically Install Third Party Services on Wireless Gateways," in Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture. ACM, 2014, pp. 43–48.

[3] "Riaps," https://riaps.isis.vanderbilt.edu/redmine/projects/riaps.

[4] S. P. Lau, G. V. Merrett, A. S. Weddell, and N. M. White, "A traffic-aware street lighting scheme for smart cities using autonomous networked sensors," Computers & Electrical Engineering, vol. 45, pp. 192–207, 2015.

[5] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2831347.2831354

[6] EPRI, "Transforming smart grid devices into open application platforms," Electric Power Research Institute Report 3002002859, July 2014. [Online]. Available: http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?productid=000000003002002859

[7] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13), Paderborn, Germany, Jun. 2013.

[8] A. Dubey, G. Karsai, and N. Mahadevan, "A Component Model for Hard Real-time Systems: CCM with ARINC-653," Software: Practice and Experience, vol. 41, no. 12, pp. 1517–1550, 2011.

[9] ——, "Formalization of a component model for real-time systems," 04/2012 2012. [Online]. Available: http://www.isis.vanderbilt.edu/sites/default/files/ISIS-12-102-TechReport.pdf

[10] BeagleBoard.org - black. [Online]. Available: http://beagleboard.org/black

[11] "Cities skyline," http://www.citiesskylines.com.

[12] "Opendht," https://github.com/savoirfairelinux/opendht.

[13] Zopkio: A functional and performance test framework for distributed systems. [Online]. Available: https://github.com/linkedin/Zopkio

[14] Service discovery overview. [Online]. Available: http://www.simplicityitself.io/getting/started/with/microservices/2015/06/10/service-discovery-overview.html

[15] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode." IEEE, pp. 34–39. [Online]. Available: http://ieeexplore.ieee.org/document/7217926/

[16] Apache helix - service discovery. [Online]. Available: http://helix.apache.org/0.7.0-incubating-docs/recipes/service_discovery.html

[17] Introduction - consul by HashiCorp. [Online]. Available: https://www.consul.io/intro/index.html

[18] M. Hoefling, F. Heimgaertner, and M. Menth, "Advanced communication modes for the publish/subscribe c-DAX middleware," in Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP. IEEE, pp. 1309–1314. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7503009/

[19] Cyber-secure data and control cloud for power grids. [Online]. Available: http://www.cdax.eu/

[20] What is dds? [Online]. Available: http://portals.omg.org/dds/what-is-dds-3/

[21] O. M. G. Specification, "RTPS specification."

[22] K. Beckmann and O. Dedi, "sDDS: A portable data distribution service implementation for WSN and IoT platforms," in Intelligent Solutions in Embedded Systems (WISES), 2015 12th International Workshop on. IEEE, pp. 115–120. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7356992/

[23] S. Cirani, L. Davoli, G. Ferrari, R. Leone, P. Medagliani, M. Picone, and L. Veltri, "A scalable and self-configuring architecture for service discovery in the internet of things," vol. 1, no. 5, pp. 508–521. [Online]. Available: http://ieeexplore.ieee.org/document/6899579/

[24] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications," vol. 53, no. 9, pp. 48–54. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7263372/