Towards an Architecture for Evaluating and Analyzing Decentralized Fog Applications

Scott Eisele, Geoffrey Pettet, Abhishek Dubey, Gabor Karsai
Dept of Electrical Engineering and Computer Science
Vanderbilt University, Nashville,TN 37235, USA

***Abstract*—As the number of low cost computing devices at the edge of network increases, there are greater opportunities to enable novel, innovative capabilities, especially in decentralized cyber-physical systems. For example, in an urban setting, a set of networked, collaborating processors at the edge can be used to dynamically detect traffic densities via image processing and then use those densities to control the traffic flow by coordinating traffic light sequences, in a decentralized architecture. In this paper we describe a testbed and an application framework for such applications.**

I. Introduction

The demand for Internet throughput can only increase as the number of Internet enabled devices continues to grow and this trend, according to [1] and other similar studies will continue. Currently many of these new devices are acting as dumb portals to a scalable cloud back-end but as the number of devices increases and the bandwidth available for each device diminishes we may reach a point where connecting to the cloud is no longer feasible. One way this could occur is if the data to be processed exceeds available bandwidth; for example an autonomous car may generate up to 1GB of data per second [2] but currently LTE can only handle a max speed of about 50MB/s in a 10MHz channel. The cloud is also not feasible for applications with stringent latency requirements, like the smart grid and many other industrial control systems [3]. The only options to resolve these problems are to increase the bandwidth to the cloud or to improve bandwidth utilization. Fog computing is one way to improve bandwidth utilization; by using devices that have local processing capability to handle some work, the amount of data sent to the cloud can be reduced. Since the devices are geographically distributed and thus only interfere with devices in their vicinity the system is more scalable.

Obviously fog computing doesn't solve the whole problem, there are still limits to the bandwidth available in a region. It also introduces a number of other challenges, including traditional distributed computing like synchronization, and application deployment and management(software updates)[4], as well as fog specific challenges. Some of these fog specific challenges include device discovery, determining application placement or workload distribution despite resource constraints, reduced reliability compared to cloud services (due to device failures, unknown availability, device mobility, etc), and device heterogeneity. Additionally it is well known that design errors become more costly to fix as the design progresses [5] and this is even more true for fog applications, due to the number of devices, particularly if a person needs to interact with them singly.

Recognizing the growing need for fog applications as well as the challenges, we are interested in developing a framework to aid in the development and testing of distributed fog applications. To handle distributed system problems, resource discovery, as well as app deployment and management we leverage our prior work RIAPS, a middleware developed for distributed smart grid applications but which can be used for any fog application. We discuss RIAPS in section II

RIAPS is useful in developing the application and deploying but we need additional tools to determine if an application is correct. To assist in this we focus on three additional areas. The first is network analysis. Since there will be congestion in the network we need to know how much congestion the application under development can sustain and still meet its performance specifications. There are many ways in which this could be accomplished, in this work we explore utilizing PRISM for this purpose which is in section III-A. Next we notice that the fog resources may not be sufficient to host an application, or handle all of the processing locally and so some work must be deployed to a cloud service, which incurs a cost. To determine this distribution of application workload we present a dataflow simulation which takes a workload as input and determines how to distribute it on a collection of fog and cloud resources. This is in section III-B. The final area presents our hard-ware-in-the-loop (HIL) testbed which lead us to recognize the need for the previous tools. This testbed is important in determining the network load of an application, which in turn is used to determine congestion limits and workload placement. Then the workload placement can be validated again on the HIL testbed. For the testbed we use single board computers hosting a *distributed traffic monitoring and control* application which collects traffic data from a city simulation and actuates the traffic lights. This is in sections IV and IV-A

Frameworks for traffic applications have been presented in other recent works [6] that address issues dealing with communication delay between autonomous cars and a traffic light. Their work is simulation based and they do not address how their protocol might be implemented. Their work also does not extend into the space of Fog applications; they do not address issues associated with distributed traffic control and intercommunication between intersections. In our work we include both hardware in the loop simulations as well as communication between traffic controllers.

The remainder of the paper is as follows, Section II discusses the platform for design and deployment of applications. Section III discusses simulation and network analysis, section IV presents our hardware testbed, section IV-A presents the case study and is followed by the conclusion in section V.

II. Information Architecture Platform

The resilient information architecture for smart systems (RIAPS) is a decentralized middleware platform comprising a set of communication, application deployment and platform management services. These services include time synchronization, distributed coordination, and a discovery service. An application in RIAPS is a collection of actors (processes) that are deployed on computing nodes of a network. The actors manage components that provide the functionality of the application. An actor provides an interface to RIAPS services for the hosted components. The components interact with one another using well-defined interaction semantics alluded to in Figure 1. Furthermore, the component architecture, an extension of our prior work [7], ensures that the applications are free from a number of concurrency issues such as deadlocks and race conditions. All these features are designed with the intention of reducing the overall management costs, which we expect to be higher given that the devices are going to be distributed in the physical environment and will often require physical travel to the device location for maintenance.

A. Actor and Component Model

Component-based design of complex software has several advantages that have been recognized by the industry and various large-scale systems. Components encapsulate functionality in a reusable unit that can be composed with other such units to form an application. This architectural principle is used in many applications today, including the Android platform [8] for smart phones and the AUTOSAR standard [9] for Embedded Control Units (ECUs) in cars. The component model defines what a component is, how it can be customized, deployed, executed, and how it interacts with other components to form applications. A component model also defines a component framework: a middleware software layer that implements common services needed by application components. Although component-based software engineering has traditionally been used to develop enterprise applications, a number of prior efforts [10], [11], [12], [13] have also used it for real-time and embedded applications. These component models for real-time and embedded applications focus on assuring one or more of the non-functional properties (e.g., timeliness, reliability, and security), satisfying resource limitations, and handling uncertainties in operating environments.

The RIAPS platform supports component-based applications to obtain these advantages. While a number of component models exist, they are either too complex or have too much management overhead to be used in Fog environments.

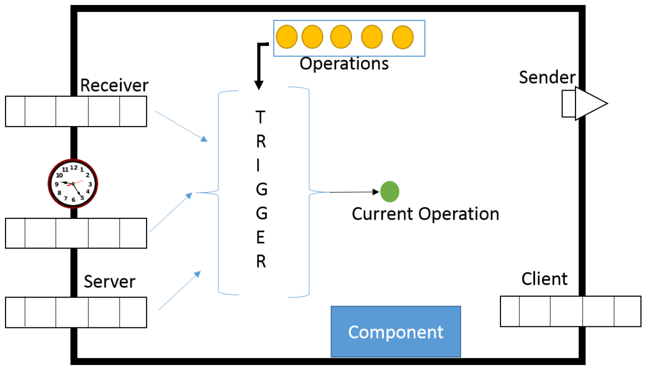


Fig. 1: RIAPS Component Model

For this purpose, we have defined a minimal, yet expressive, component model for RIAPS. A RIAPS component is a reusable unit that has a set of operations that manipulate the component's state and that can interact with other components of the application via *ports*. As shown in Figure 1, a RIAPS component supports several port types: *sender*, *receiver*, *client*, *server*. Sender is unique because it is used to only send messages outside of a component. The receiver port is exclusively used to bring messages into the component. Both client and server are special ports in that they can be used to send messages and receive messages. This is required to implement the common pattern of request/response, where the "client" can send a request message via a client port, which is then received by the server port of some other component, which then sends a "response" back to the component that originated the request. Client, server, and receiver ports are individually buffered, i.e. the messages are guaranteed to be held in the port until a component operation has consumed the message or unless the port's buffer is full. An additional construct is *timer*: it can be armed and used to produce messages that record the time of timer expiry and that trigger the invocation of a component operaion. The computation aspect of a component is single threaded and is managed by a *trigger* method provided by the component developer. The trigger method can evaluate arbitrary conditions (e.g. the state of the component) and events related to the ports and use the result to select an operation that executes the core business logic of the component. The trigger fires (1) when the state of ports of the components change, (2) when a timer expires, or (3) when an operation is completed.

Logically, multiple components are grouped together to create applications. Assuming that the components for these applications have dependencies (e.g. hardware, location, etc) that cannot be satisfied on a single RIAPS node, the components of the application are distributed among the available nodes.

An actor acts as a container for components and it allows the component to interact with other services that the system provides including time synchronization, resource discovery and distributed coordination. The actor also provides a standard interface for remote management of the components that it hosts. This is required to ensure that all the components of an application can be installed and configured correctly. The actor also provides process abstraction and resource containers for the components. This makes it possible to limit hardware resources available to individual components, as well as limits the resources used by a group of components within an actor.

B. *Isolating the Physical System Interaction*

Fog applications are primarily cyber-physical in nature. That means the applications must be able to interact with physical sensors and actuators. This is challenging because the devices are heterogeneous and do not share the same interfaces. In order to address this RIAPS includes a device interface service whose goal is to *encapsulate* specific I/O devices so that application components can (a) access them using a *unified* interface and (b) precisely time interactions with devices and other components. To achieve these goals, the service must contain the necessary drivers, resource arbitration methods, and a real-time scheduler. This is one of the most comprehensive platform services and it is tightly integrated with the Time Synchronization Service for executing device interactions on a globally synchronized timescale.

III. Network Analysis and Simulation

A. *Network Analysis Framework*

The purpose of a network analysis tool in our framework is to determine the congestion levels that the developed application can tolerate given its own message size and communication rate and still meet its requirements. We elected to model the 802.11a protocol in PRISM a probabilistic model checker intended for formally modeling and verifying systems that have probabilistic, or nondeterministic behavior. The formal specifications can be expressed in the PRISM [14] property specification language, which can express statements from PCTL, CSL, LTL and PCTL*.

In our first application of this tool we check how long we should expect to wait for a station to be successfully send a message if there is random interference. This kind of interference can result from sources that do not implement the 802.11a protocol or if there are hidden nodes. For fog applications this interference can be modified to model the background congestion.

We see the result of this experiment in Figure 2. The x axis is the probability that there will be interference at a given time step, where that probability is $\frac{x}{1000}$. The y axis is the expected time to wait in μs. We see that with 2% random utilization on average it should take $0.1s$ for the station to be able to send a message. However, the trend is exponential: as the probability that the medium is randomly accessed by uncontrolled stations grows linearly the expected time for a station following the 802.11a protocol to be able to send a message grows exponentially. This trend is likely because the interruptions are a Poisson process. This experiment does show us that PRISM may be used to determine how congestion impacts message delivery. By including a minimum message transmission time in the application specification we can

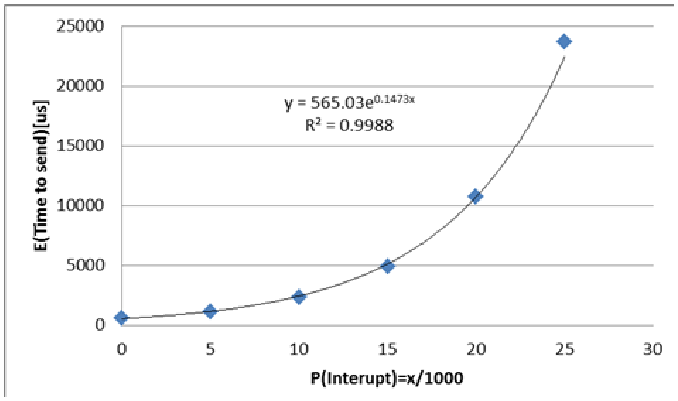


Fig. 2: The expected time to send a message vs. the probability of interruption by uncontrolled external sources

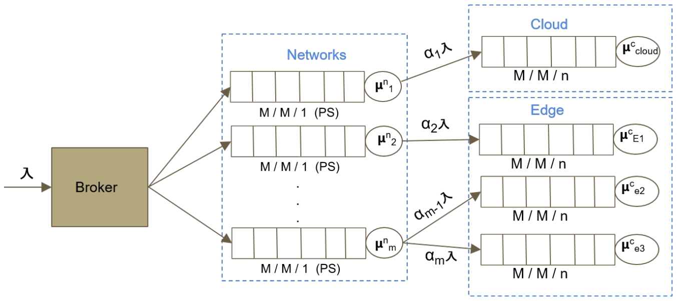


Fig. 3: Diagram of the Workload Simulation Model

monitor when the controller is no longer able to perform is function.

B. *Dataflow Simulation*

One of the key difficulties when creating a Fog application is determining the optimal placement of workloads across edge devices and the cloud. This placement will depend on the needs of the application, including latency demands, availability of edge resources, and economic constraints. Rather than determing the distribution of the workloads via trial and error on an actual deployment, which would be costly, a simulation of the system can be used to optimize their placement as shown in our prior work [15]. However, due to the nature of Fog computing we need to consider a multi-tier resource approach, where the resource sharing across tiers incur different costs, including network latency and direct financial costs. Rather than being strictly defined, tiers are associated with different resources.

The simulation used in this framework uses the MATLAB SimEvents tool: a discrete-event simulation engine that includes a component library for analyzing event-driven system models [16]. Workloads are represented using the built-in 'entity' objects, with each entity representing an individual task to complete. The model used in the framework consists of the following components shown in Figure 3, explained below.

- **Workload Generation**: Entities are generated at some rate λ, have a size that impacts network utilization, and a complexity that impacts computation time. Each of these properties can be constant or sampled from a distribution. After being generated the entities are sent to the Broker.

- **Broker**: The Broker determines which computational device to send each entity as it is generated and then directs it to the appropriate network connection. It stochastically averages the workload across these devices based on both the devices' computation speed and the characteristics of the networks connecting them. It accomplishes this by optimizing the rate at which it sends to each device via the DIRECT algorithm below.
- **Network**: Each network is represented as an M/M/1(processor sharing) queue [17], which models entities being transfered as processes to be computed. The shared processing (ps) property allows multiple entities to be processed at a time, which simulates the ability to send multiple files over a network simultaneously. The time required to transfer an entity depends on its size.
- **Computation Devices**: The edge devices and the cloud are represented as M/M/1 and M/M/N queues respectively [18]. The M/M/N queue represents the ability of the cloud to scale indefinitely, while edge devices are limited to their local resources. Each entity is processed one at a time on each device, after which it reports the amount of time it took to be processed and exits the model. Currently the simulation does not support resending computation from the edge to the cloud: once the computational device is chosen, the entity must be processed there.

This framework is flexible enough to support any dataflow application where data does not need to travel between tiers. It can also support many optimization parameters. For example, the average latency can be determined by examining how long it takes entities to travel through the system, and monetary costs can be associated with the computational resources (to simulate cloud computation costs, for instance). After exposing the constraints for the application, the parameters can be optimized using the DIRECT algorithm, described next.

1) DIRECT Optimization: The DIRECT algorithm [19] is a modification to the Lipschitzian optimization algorithm [20]. It solves the optimization problems using sampling, which is useful when the objective function is opaque. It is guaranteed to converge to the optimal solution, although it might come at the cost of an exhaustive search in the worst case.

A function is Lipschitz continuous on R^1 if the following holds: *Let M ⊂ R^1 and f: M ⇒ R. The function f is called Lipschitz continuous on M with Lipschitz constant α if*

$$|f(x) - f(x')| \leq \alpha |x - x'| \forall x, x' \in M \tag{1}$$

Shubert's algorithm uses this property to find the minimum of an objective function $f(x)$ with Lipschitz constant α. It divides the domain into regions and estimates each region's minimum f value using the Lipschitz constant: the slope of the function will never be greater than α. It then narrows its search where the estimated function value is lowest and continues until the actual minimum value is found [19].

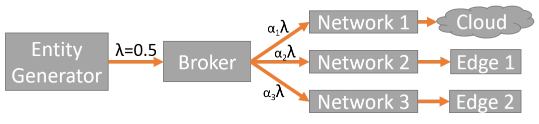


Fig. 4: Diagram of Simulation Example

Unfortunately this has a few drawbacks, both of which are addressed by the DIRECT algorithm. The first is that the endpoints of the regions discussed do not generalize well into higher dimensional space. DIRECT instead samples the midpoints of search regions, which do generalize to higher dimensions. The second issue is that it is not guaranteed that these problems are Lipschitz continuous. Even if they are, a poor choice of the Lipschitz constant can lead to poor optimization time. DIRECT addresses this by 'using all possible values to determine if a region of the domain should be broken into sub-regions during the current iteration' [19], which does not require Lipschitz continuity. While it implies using more values, it can cut down on the number of iterations to complete.

The DIRECT algorithm is implemented in MATLAB and can interface directly with the system model described in section III-B1. All the user needs to do is to construct the appropriate constraints and an optimization function. Once run on the model, the algorithm outputs the optimal workload distribution across the edge devices and the cloud. This step can be done before implementing the application in RIAPS to a) ensure that the application will work well with the given resources, and b) find the optimal workflow distribution across RIAPS resources before implementation.

We created an example network to test the simulation framework, shown in Figure 4. A new Entity is created every other discrete step. Each of the networks have identical transmission speeds of 0.2 entities per step in a shared processing queue. Each network transmits entities to either the scalable cloud or one of two edge devices. Edge 1 has a processing time of 1 entity per step while Edge 2 processes 0.5 entities per step. The Direct optimization algorithm found that a minimum average processing time of 5.0 steps per entity was obtained when α_1 = 0.5, α_2 = 0.3889, and α_3 = 0.1111.

IV. Hardware in the Loop Simulation

In order to test our applications we constructed a hardware in the loop test bed seen in Figure 5. The testbed is composed of a cluster of Beagle Bone Black(BBB) single board computers that are networked via routers which allow us to define a control plane to creating a virtual network. A high performance computer is also connected that is responsible for running the simulations.

A. Example: Adaptive Traffic Controller

An adaptive traffic controller considers the current and predicted traffic density in connected blocks of a neighborhood and maximizes the net traffic flow. This application requires

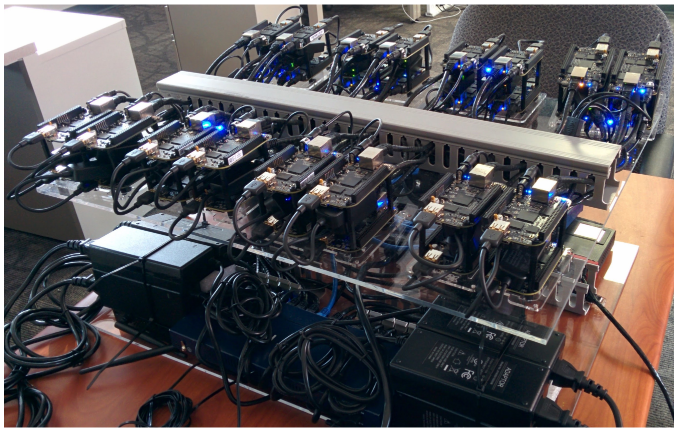


Fig. 5: Hardware Test Bed

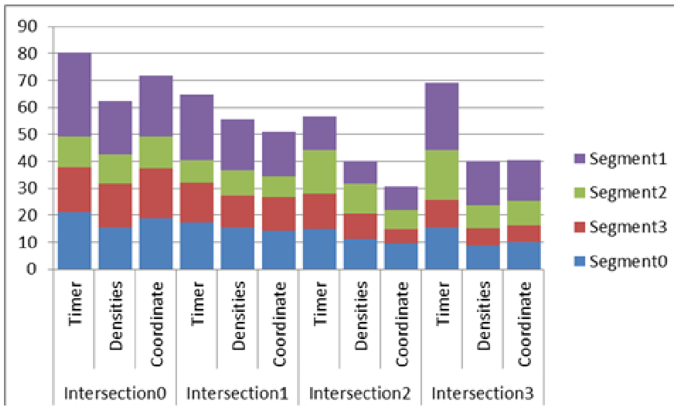
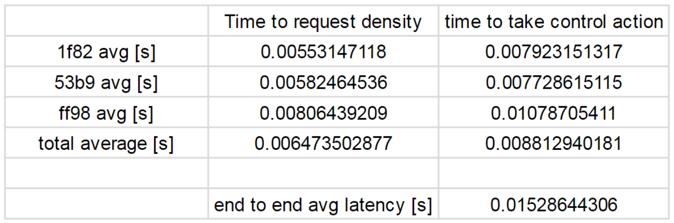


Fig. 6: Compare densities at each intersection using the various controllers



	Time to request density	time to take control action
1f82 avg [s]	0.00553147118	0.007923151317
53b9 avg [s]	0.00582464536	0.007728615115
ff98 avg [s]	0.00806439209	0.01078705411
total average [s]	0.006473502877	0.008812940181
	end to end avg latency [s]	0.01528644306

Fig. 7: End to end latency for receiving density data from Cities: Skylines, taking control action and actuating the traffic light.

streams of raw data from sensors associated with street segments in real-time, which is then used to estimate traffic density and make control decisions. Clearly, this application requires coordination, time synchronization and low-latency computation among traffic controllers associated with different intersections.

We developed this example on 4 beagle bone blacks, where each traffic controller (per intesection) was deployed on a BBB, which acts as a RIAPS node. The traffic dynamics and physics were simulated by a game called Cities Skylines [21].

The BBB interfaces with the simulation via UDP messages. Our hardware testbed allows creation of virtual connections between nodes. Since the controllers are mapped to intersections in the simulator we can set the network topology to restrict communication to neighboring intersections.

We deploy three traffic controller variants. One where the light switched on a timer, one where switching depends on the timer and sensed traffic density, and one where the controllers publish the their sensed density to neighboring intersections. The algorithms are cumulative. The first switched after some time period, the second added switching after crossing a threshold, and the third modified that density value, adding a portion of neighboring density in anticipation of future traffic.

This test bed allowed us to compare the effectiveness of the the three variants. Figure 6 compares the densities at each intersection using each of the controllers. The coordinated controller has a 28% reduction in overall traffic density compared against the simple traffic timer. This shows that we can indeed quickly prototype and test alternate control mechanisms in the framework.

Another useful metric of this test is the time it takes for the controller to obtain state information from the simulation and send a command to change the state. Our controller samples traffic density every 10ms and took on average 6.5ms to query the game, while the controller runs every second. In the case where the controller needs to actuate the lights it takes on average 9ms to send a signal to the game and receive a response that the change has occurred. The total time necessary to respond to a change in the simulated environment in this test bed then is approximately 15.5ms, which is adequate for controlling traffic.

These tests brought to our attention the need for additional tools. The hardware-in-the-loop simulation demonstrated that, indeed traffic performance improves when the lights can coordinate, but there is a cost in increased message traffic. How much background wireless congestion will cause this performance to degrade? This testbed will allow us to deploy applications and determine the message size and rate needed for the application to be effective. But it does not allow us to evaluate the effect of congestion. This is the reason for working on the network analysis aspect of the framework in III-A. Additionally in this simulation the traffic data used for the control logic was provided by the city simulator. In a real deployment, this information may only be available through processing of images. Since the BBBs are resource limited devices this may not be possible on board and the workload may need to offloaded to other local devices or the cloud. This is the justification for the workload placement aspect of the framework in section III-B.

V. Conclusions

In this work we have presented a framework for the development, deployment and testing of Fog applications. It includes simulation, network analysis and a HIL testbed, all of which are supported by the RIAPS platform. As the applications are further developed the framework will continue to provide feedback on the correctness of the applications designed with it.

Acknowledgment: The 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 of Siemens, CT.

REFERENCES

[1] S. Lucero, "IoT platforms: enabling the internet of things."

[2] L. Mearian. Self-driving cars could create 1gb of data a second | computerworld. [Online]. Available: http://www.computerworld.com/article/2484219/emerging-technology/self-driving-cars-could-create-1gb-of-data-a-second.html

[3] M. Weiner, M. Jorgovanovic, A. Sahai, and B. Nikolie, "Design of a low-latency, high-reliability wireless communication system for control applications," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, pp. 3829–3835. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6883918/

[4] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," vol. 3, no. 6, pp. 854–864. [Online]. Available: http://ieeexplore.ieee.org/document/7498684/

[5] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle." [Online]. Available: https://ntrs.nasa.gov/search.jsp?R=20100036670

[6] B. Zheng, C.-W. Lin, H. Liang, S. Shiraishi, W. Li, and Q. Zhu, "Delay-aware design, analysis and verication of intelligent intersection management."

[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. Developers, "What is android," 2011.

[9] Autosar GbR, "AUTomotive Open System ARchitecture," http://www.autosar.org/. [Online]. Available: http://www.autosar.org/

[10] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich, "Components for Embedded Software: the PECOS Approach," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2002, pp. 19–26.

[11] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, vol. 3291. Agia Napa, Cyprus: Springer-Verlag, Oct. 2004, pp. 1520–1537.

[12] T. Bures, J. Carlson, S. Sentilles, and A. Vulgarakis, "A Component Model Family for Vehicular Embedded Systems," in *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*. IEEE, 2008, pp. 437–444.

[13] 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.

[14] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*. Springer, pp. 585–591. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-22110-1_47

[15] N. Roy, A. Dubey, A. Gokhale, and L. Dowdy, "A capacity planning process for performance assurance of component-based distributed systems," in *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11. New York, NY, USA: ACM, 2011, pp. 259–270. [Online]. Available: http://doi.acm.org/10.1145/1958746.1958784

[16] "Mathworks SimEvents," https://www.mathworks.com/products/simevents.html, 2008.

[17] E. G. Coffman Jr, R. R. Muntz, and H. Trotter, "Waiting time distributions for processor-sharing systems," *Journal of the ACM (JACM)*, vol. 17, no. 1, pp. 123–130, 1970.

[18] L. Kleinrock, *Queueing systems, volume 2: Computer applications*. wiley New York, 1976, vol. 66.

[19] D. E. Finkel, "Direct optimization algorithm user guide," *Center for Research in Scientific Computation, North Carolina State University*, vol. 2, 2003.

[20] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian optimization without the lipschitz constant," *Journal of optimization Theory and Applications*, vol. 79, no. 1, pp. 157–181, 1993.

[21] "Cities skylines," http://www.citiesskylines.com/.