A Rapid Testing Framework for a Mobile Cloud Infrastructure

Daniel Balasubramanian, Abhishek Dubey, William R. Otte, William Emfinger, Pranav S. Kumar, Gábor Karsai

ISIS / Vanderbilt University, Nashville, TN 37212
Email: {daniel.a.balasubramanian, abhishek.dubey, w.otte,
william.a.emfinger, pranav.s.kumar, gabor.karsai}@vanderbilt.edu

***Abstract*—Mobile clouds such as network-connected vehicles and satellite clusters are an emerging class of systems that are extensions to traditional real-time embedded systems: they provide long-term mission platforms made up of dynamic clusters of heterogeneous hardware nodes communicating over ad hoc wireless networks. Besides the inherent complexities entailed by a distributed architecture, developing software and testing these systems is difficult due to a number of other reasons, including the mobile nature of such systems, which can require a model of the physical dynamics of the system for accurate simulation and testing. This paper describes a rapid development and testing framework for a distributed satellite system. Our solutions include a modeling language for configuring and specifying an application's interaction with the middleware layer, a physics simulator integrated with hardware in the loop to provide the system's physical dynamics and the integration of a network traffic tool to dynamically vary the network bandwidth based on the physical dynamics.**

I. Introduction

Mobile clouds are an emerging class of distributed, real-time embedded systems comprised of multiple mobile nodes communicating over ad hoc wireless networks. These systems have two main characteristics. The first is the mobile nature of the hardware nodes, meaning that their physical location changes over time. The second is that the physical resources located on the hardware are shared between the nodes over the wireless network. There are several advantages that this type of architecture entails. By sharing resources among nodes, a hardware malfunction in one node can be mitigated by using the hardware provided by another node. This provides a fault tolerant platform ideal for long-term missions in remote areas.

Realizing the full potential of such mobile cloud systems necessitates a software application platform that supports secure and fault-tolerant sharing of resources: processors, storage, communication links and devices. The system must enable secure, on-demand collaboration between applications operated by different organizations. Clearly, the economic viability of these systems depends on the ability to rapidly assemble reliable distributed applications from reusable software components, including those sourced from various vendors.

Our team recently built the operating system (OS) and middleware layers for a particular type of mobile cloud platform: a distributed satellite cluster [**?**]. The individual satellites form the mobile hardware nodes, and individual units of hardware, such as cameras and sensors, are shared between the satellites over the wireless network. The OS provides a core set of low-level abstractions to user-level applications, such as temporal process isolation and security, and the middleware provides a higher level set of abstractions on top of the OS.

In order to validate that the OS and middleware layers provide the expected level of service to user-level applications, we needed a testing framework that was as close to the actual distributed satellite system as possible. In particular, we needed the ability to check whether mixed criticality applications running on the system all met their process scheduling deadlines. This was challenging for the following reasons: (1) We didn't have access to the actual hardware onto which the OS, middleware and applications would eventually be deployed, (2) The physical dynamics (the flight path) of the system affects the distance between nodes, which in turn affects network quality, such as bandwidth, and (3) Applications have to interact with and configure several aspects the middleware.

The first challenge above (not having access to the real hardware) meant that we had to find a suitable way to emulate the actual hardware. This raised question whether cost-effective software virtual machines (VMs) would be sufficient to accurately emulate process scheduling and network links, or if dedicated hardware would be required. The second challenge stems from the mobility of these nodes. Because the nodes (e.g., a cluster of satellites) form a dynamic physical system, the physical position of nodes will change over time depending upon the orbital mechanics and the resulting flight path. The changing distance impacts the quality of the radio signal, which affects the available bandwidth distributed applications can use. Therefore, a testing framework needs to incorporate a 'simulation' of the physical dynamics in order to test the applications and see if the fluctuating quality of the network affects the communication between applications, which in turn can affect whether they meet their scheduling deadlines. The third challenge, configuring the middleware, is an issue in any middleware-based platform for embedded systems. The configuration space for middleware can be very large and the configuration files themselves can be rather low-level. Hence, a testing framework needs a way to hide this complexity.

This paper describes the testing framework we developed for quickly prototyping applications running on our OS and middleware that addresses each of the challenges above. Our main contributions are (1) the description of our hardware infrastructure and how it integrates a physical dynamics simulator to emulate the movement of the hardware nodes, (2) integration of a network traffic shaper to account for the vary-

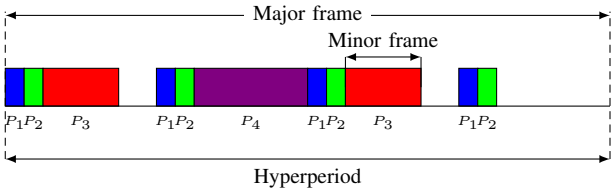

Fig. 1. A Major Frame. The four partitions (period,duration) in this frame are P_1 (2s, 0.25s), P_2 (2s, 0.25s), P_3 (4s, 1s), and P_4 (8s, 1.5s).

ing network characteristics, and (3) model-based methods for configuring an application's interaction with the middleware. We feel that these techniques are general enough to be reused by any distributed embedded system facing similar challenges.

The remainder of this paper is structured as follows. Section **??** gives background information on the OS, middleware and target platform. Section **??** describes our technical solutions. Section **??** demonstrates how the pieces of our testing framework interact using an example. Related work is presented in Section **??**, and we conclude in Section **??**.

II. Background

Recently, we built the operating system and middleware layers for a fractionated spacecraft system. Together, the operating system and middleware are referred to as DREMS: Distributed Real-time Managed System [**?**]. The operating system provides a unique temporal partitioning scheduler and a novel communication mechanism, which together help to ensure that mixed criticality processes can be scheduled deterministically and their performance and information flows can remain isolated. The middleware abstracts the lower-level features of the operating system by providing high-level interfaces and configuration mechanisms for application communication, including network quality of service (QoS).

Figure **??** shows an example of how the temporal partitioning for processes works. It provides temporal isolation between unrelated processes by using a periodically repeating fixed interval of the CPU's time exclusively assigned to a group of cooperating processes of the same application. A temporal partition is characterized by two parameters: *period* and *duration*. The period reflects how often the tasks within the partition will be guaranteed CPU allocation. The duration governs the length of the CPU allocation window in each cycle. Given the period and duration of all temporal partitions, an execution schedule can be generated by solving a series of constraints, see [**?**]. A feasible solution, *e.g.* Figure **??**, comprises of a repeating frame of windows, where each window is assigned to a partition. These windows are called *minor frames*. The length of a window assigned to a partition is always the same as the duration of that partition. The repeating frame of minor frames, known as the *major frame*, has a length called the *hyperperiod*. The hyperperiod is the lowest common multiple of the partition periods. This process scheduling model is similar to that described by the ARINC-653 [**?**] standard for avionics computing. While most tasks perform application functions, some tasks are critical for system management. To reduce the execution latency for critical system tasks, the platform supports grouping these tasks into a different criticality level, which are not subject to temporal partitioning

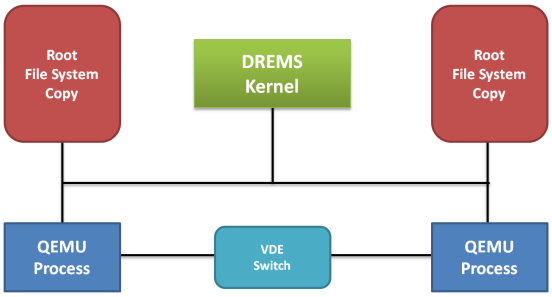

Fig. 2. Virtualized Simulation

and can execute as soon as possible. Lastly, the platform supports best effort tasks that are scheduled only when there are no runnable tasks from the partitions and critical category.

The network QoS requirements for an application specify the quantity of network resources an application requires during different periods of its execution. This can include metrics such as bandwidth (available bit rate) and latency (time interval for communication). A testing framework for this system must be able to validate that the network QoS requirements of each application are met while also satisfying the scheduling demands of both temporally partitioned and time-critical processes. With a static deployment where the network infrastructure is fixed, this is a fairly straightforward problem. However, the unique challenge here is the mobile nature of the system: the distance between the nodes changes over time. As the distance between the nodes changes, the radio signal degrades and it takes longer to transfer the messages, which lowers the effective bandwidth. This has a huge impact on the way in which applications are tested.

III. Testing Framework

This section presents our technical solutions to the challenges described above in three parts. The first part describes the testing hardware/infrastructure. The second part describes how we integrated the physical system dynamics and network quality control. The third part describes our model-based methods for configuring the middleware.

A. Simulation hardware

In order to provide a flexible and low cost (i.e., free) environment in which developers could test and evaluate both infrastructure (i.e., the operating system, middleware, and deployment infrastructure) and applications, we developed a virtualized simulation platform, shown in figure **??**, that could be run on a normal developer's workstation. This virtualized environment was built using the Quick EMUlator (QEMU) [**?**], a free and open source hosted hypervisor that emulates a wide variety of processor architectures. QEMU requires as parameters a binary kernel image and a root file system, which contains all of the necessary executables, libraries and configuration scripts necessary for the operating system to boot. As the root file system will likely be modified by the running DREMS OS, each QEMU instance requires a private copy of the root file system. Using QEMU, we were able to configure our build system to launch a configurable number of instances of the DREMS OS that are able to communicate via network links provided by Virtual Distributed Ethernet

(VDE) [**?**], a software defined networking infrastructure that allows virtual network topologies.

This virtualized infrastructure proved to be an extremely convenient and effective tool for developers to quickly evaluate the functionality of the various pieces of the infrastructure and application programs. Moreover, this allowed us to easily and inexpensively set up automated continuous integration services that automatically build and ran a test suite for the entire DREMS stack. However, we quickly noticed that the price for this convenience and flexibility was a tremendous lack of determinism: testing and debugging timing critical software gave very inconsistent results. This resulted from two sources. First, we did not have precise control of when the host operating system would schedule the QEMU instances to run, and how the execution would be interleaved — especially of multi-node tests/experiments — would change from one execution to another. Second, there was a wide disparity of host hardware used by infrastructure and application developers: problems that would manifest for a developer running the simulation environment under Linux hosted on bare metal might not manifest for a developer running the same under Linux hosted inside a VM.

As a result of the non-deterministic behavior of the virtualized simulation infrastructure, we created a multi-node hardware simulation infrastructure. This multi-node environment consisted of several fan less single board computers with a 1.6 GHz Atom N270 processor and 1 GB of RAM each. Each node had a single Compact Flash (CF) card that acted as the hard disk drive. These computers were connected via a private gigabit Ethernet switch and utilized software traffic shaping, described in Section **??**. In order to configure these systems, the CF card was initially imaged with the same root file system and kernel image generated for the virtualized infrastructure. This is a lengthy process that requires manually manipulating the CF card, but fortunately can be avoided (presuming that the node is capable of booting) by copying over future software updates using SSH.

B. Network

For accurate simulation of distributed systems and their applications, dedicated hardware must provide not only accurate, deterministic timing, but also accurate emulation of the expected network behavior that the system will experience. This section will explain (1) why accurate network emulation is important, (2) what makes network emulation challenging and (3) our solution to system prototyping with accurate network emulation.

Because mobile clouds, and more generally, embedded systems, are increasingly utilizing mobile ad-hoc networks, the communication of both the applications and infrastructure is subject to the dynamics and variations of the network, which is further influenced by the node mobility. As the network becomes an increasingly important resource for systems and their applications, accurate reliable emulation of the network must be incorporated into the system simulation. Without this integration, the system simulation would be incomplete and would provide a partial perspective on the system's performance. Further, an application's timing and execution are increasingly dependent on the stimuli provided by information from the network. Just as real hardware must be used to accurately simulate application processor timing characteristics, so too must a real network be used to accurately simulate application network timing and execution performance.

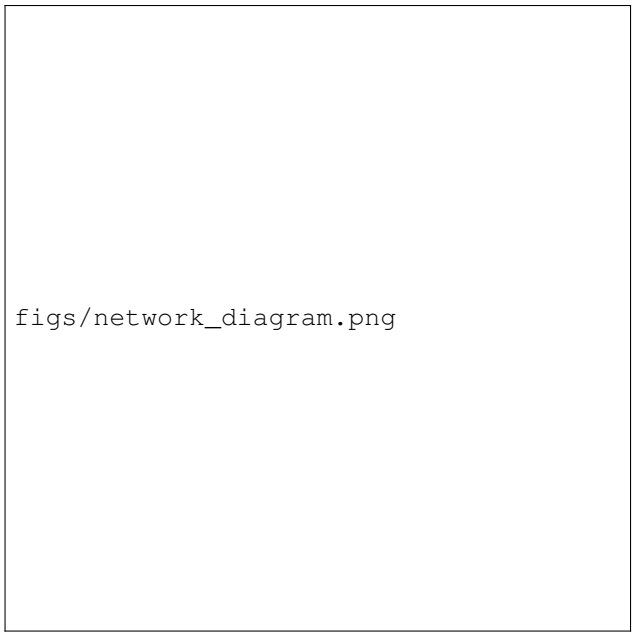

Fig. 3. Physical and virtual connections between the system simulation nodes, the network emulation node and the physical dynamics simulation node.

Simulating the system's network is a complex task for two main reasons: (1) the complex physical dynamics of the system itself, and (2) the configuration of the network control. One of the key factors in determining the network resources available at a point in time is the positions of the nodes, which vary with time. By incorporating the physical dynamics along with the physical characteristics of the network transceivers into a simulation engine, part of the problem is solved. However, the network characteristics must still be simulated, and the information from this simulation must be fed into the node network emulation hardware. There are several choices for simulating the dynamics and also for network control, and thus the integration of the two is not simple.

We evaluated several options for configuring the network emulation: (1) network emulation in a (programmable) switch or router connecting the nodes, (2) network emulation on each node itself or (3) network emulation on a dedicated network emulation node. Network emulation on the switch or router is an excellent choice for performance reasons, as it has the least processing overhead. However, the configuration of the switch or router is not easy to set up and is not easy to control remotely or dynamically in a programmatic fashion. Further, the addition of nodes to the network may require the use of more switches which must also be configured, increasing the complexity. Network emulation on each system node is a valid distributed approach to simulating the network and is easier to configure than a switch, but may be difficult due to the requirement of porting the network emulation software to the hardware and the infrastructure that the system uses. Additionally, network emulation in this manner will again share processing time with the application code and therefore

impact the application's timing accuracy.

Using a dedicated network emulation node combines the ease of configuration of network emulation on each node with the performance and application independence of running the network emulation on a programmable switch. The node configuration for this setup merely requires assigning to each node: (1) a physical IP address to use on the interface, (2) a virtual IP subnet that the applications will use that is not assigned to any interface, (3) configuring each node to use the network emulation node as its default gateway. The network emulation node is then configured to act as a router and run network address translation (NAT) from the virtual IP addresses to the physical IP addresses. This NAT-ing ensures that all application traffic must pass through the network emulation node.

To actually perform the network emulation on the node, each network link in the system must be controlled according to the system's physical and network dynamics. This traffic control requires both the use of special network emulation programs and the integration with the physical dynamics simulation. For providing the network emulation, we evaluated the following options: (1) using Linux with its built-in functionality such as the Traffic-Cop (TC), (2) using Linux with Dummynet - a special network emulation program, and (3) using BSD (FreeBSD) with Dummynet[1] built-in. Using Linux's built-in traffic control features proved non-ideal simply because of the difficulty in mapping the application-level system network characteristics, *i.e.* bandwidth, latency, and packet loss, into the parameters exposed by the tool. Dummynet, which exists for both Linux and BSD, solves this problem by exposing a clean, easy to use interface for configuring and controlling network links based on their link bandwidth, the link latency and the link packet loss ratio. However, the Dummynet port to Linux is incomplete and is not completely integrated with the operating system. Therefore, we configured our network emulation node with FreeBSD because Dummynet (and its underlying IP firewall framework) is built into the operating system and is easier to configure. Figure **??** illustrates the set up.

By running Dummynet on FreeBSD, we can use simple scripts to set up the network by using Dummynet to configure all of the subnet's traffic to be NAT-ed and then apply link profiles to each virtual link of the system. This configuration separates the NAT functionality from the network emulation functionality while keeping the relevant parameters in simple to use configuration files. However, the network emulation is incomplete without including the information from the physical system dynamics simulation. Using the Orbiter space-flight simulation[2] to simulate a cluster of satellites, we can run a script on the network emulation node to periodically query the satellite positions from Orbiter. This script can then calculate the appropriate link characteristics between each node (based on the inverse-square law and configurable parameters) and apply them to the network traffic through Dummynet. Based on the calculations of our system's capabilities, the system's network bandwidth on each link varies from 10 kbps to 3 Mbps, which are data rates easily emulated by Dummynet on the gigabit Ethernet links connecting the nodes.

[1]http://info.iet.unipi.it/ luigi/dummynet/

[2]http://orbit.medphys.ucl.ac.uk/

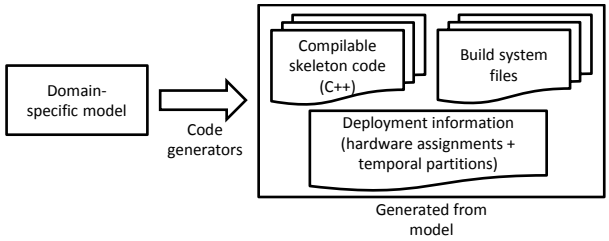

Fig. 4. Overview of the artifacts automatically generated from models.

Using this simulation system (shown in Figure **??**), extending the network with an additional node requires only minor modifications to a small number of configuration files. For our setup, a corresponding satellite needs to be added to the Orbiter simulator so that its dynamics are simulated, and the configuration file on the network emulation node which lists the nodes in the system needs to be updated. Because our design and development infrastructure automatically creates the OS images and configures the network for each node, the user does not need to edit any configuration parameters on the nodes themselves.

C. Model-based methods for configuration

Applications running on our distributed satellite platform must configure several aspects of both the OS and middleware. This includes tasks like configuring the operating system so that it knows about the temporal partitions of each process and configuring the middleware to establish the required network communication paths. This configuration is largely comprised of tedious, low-level tasks, such as writing XML configuration files. To facilitate faster application coding/debugging cycles, we created a model-driven development environment for specifying several parts of an application and its configuration, including its software architecture and communication links, its temporal partition schedule and its mapping onto physical hardware nodes.

Our model driven environment relies on a domain-specific modeling language and a set of software generators that are used to produce various configuration files as well as the necessary glue code that enables the application developers to focus on business logic. These tools also generate automation scripts for system integrators to launch the testing framework. The modeling language has a graphical syntax that not only allows domain experts to quickly define models, but also enables other stakeholders to intuitively understand the concepts and ideas represented by the models.

The software architecture of an application is defined using *components* that run on top of the middleware. Component-based programming is well-suited for distributed computing platforms in part because it provides location transparency: an application that communicates with or uses functionality provided by an external component does not need to know the location details of the external component. Instead, the application developer defines a link between the components inside a model, and a code generator creates the appropriate settings in the generated configuration file.

Figure **??** shows a high-level overview of what is generated from a domain specific model. The generated deployment

information includes a description of both the application to hardware mapping as well as a list of the temporal partitions needed to configure the OS and middleware. The compilable skeleton code includes a programmatic description of the communication interfaces exposed and used by software components. The generated build system files relieve the developer from manually specifying makefiles to compile the application.

In our experience, we found that using a high-level modeling language along with code generators to automatically produce low-level configuration files greatly increased the speed at which applications could be designed and configured reconfigured.

IV. Example

This section presents an end-to-end example showing how mixed-criticality applications can be tested and their requirements validated using our framework. The scenario consists of a cluster of 3 satellites in orbit, each running two types of applications: a critical platform application cluster flight application (CFA), and a regular CPU-intensive image processing application (IPA), which is scheduled in its own temporal partition. In short, the CFA periodically observes sensor readings and manages the flight dynamics of the satellite while logging the relative positions of all other satellites. Concurrently, the IPA records images from camera modules attached on each satellite and then perform CPU-intensive calculations and transformations on the observed data.

The goal of testing these applications together is to ensure that the CFA still has an adequate response time even when running beside the CPU-intensive IPA. This requires that (1) each CFA receives sensor data from the CFAs running on the other nodes despite dynamically varying distances between the nodes, (2) the CPU resource is strictly measured and allocated for the vendor applications (like the IPA), (3) The presence of such application activity does not interfere with the safe and timely operation of critical tasks such as CFA. This is especially necessary as trajectory changes that are imposed by the CFA are safety-critical and must be responded to with low latency.

A. System setup

Figure **??** shows the basic workflow for this example. The cluster flight application consists of four software components: *OrbitalMaintenance*, *TrajectoryPlanning*, *CommandProxy*, and *ModuleProxy*. The ModuleProxy behaves as the interface to the satellite hardware. The OrbitalMaintenance component uses the ModuleProxy to obtain the satellite state vector. On observing fresh sensor data, each satellite's OrbitalMaintenance publishes this data to every other satellite in the cluster. This is done by a group publish subscribe interaction between all Orbital Maintenance components across all nodes. The CommandProxy exposes an interface that is used to receive remote commands from a ground control. These commands can include queries on data structures, and even highly time-critical trajectory-specific commands. A *scatter* command received on the CommandProxy, if handled correctly, results in a coordinated change in cluster orientation. This is especially important when the satellites need to switch orbits to avoid collision with space debris. The CFA is a system-level critical application that is handled by critical threads that operate as required.

Additionally, four image processing applications (IPA) are deployed as application tasks. These IPAs were written so that we can configure the percentage of CPU cycles consumed by them. The four IPAs are isolated from each other and periodically use the satellite camera feed for processing. The IPAs are handled by application threads of lower priority than CFA. These IPA threads, when scheduled, take up a large part of the CPU.

Once these applications are modeled using our modeling tools, the integrated code generators generate much of the necessary code to complete the software development. The modeling tools establish the (1) component structure, (2) communication interfaces, (3) assembly of components, (4) interaction patterns through component ports, (5) scheduling policy, and (6) integration with the assumed hardware. The modeling tools do not, however, model the business logic of the software components. Therefore, the code generators generate the application and infrastructure code that captures this general structure. Development is greatly simplified by this code generation capability as the only bit of code that the developer needs to write is business logic of the components (the application logic of the CFA).

The code generators also generate a *deployment plan* that describes the application's structure, the hardware resources and the software configuration required by the infrastructure to deploy the applications. A system-level process called the *Operations Manager* uses the deployment plan and manages the safe and correct deployment of application-specific processes. As part of this management, the necessary software packages are uploaded on the hardware devices and the plan is started. Once the applications are deployed on each node, run-time management includes dynamic changes to scheduling policies, dynamic reconfiguration of application assembly and structure, monitoring facilities for the application processes, and termination of the executing plan.

In a mixed-criticality scenario such as this, one of the many important run-time challenges is validating the timing requirements of the software components being deployed. Each software component in the application has real-time deadlines to service operations. Time delays incurred due to (1) execution of operations, (2) temporal partitioning and (3) network latencies directly affect the timeliness of the application. Large deviations in critical response times (e.g., the latency between the reception of a scatter command from the ground station and the activation of the satellites' thrusters) affect the safe and reliable operation of the satellites. To mitigate these challenges, both design-time analysis and run-time measurement and monitoring facilities have been integrated into the framework. Monitoring component activity with a logging framework, network properties of the system can be measured for analysis.

To test whether or not the CFA can achieve its required response time to the scatter command, the latency of the scatter response itself was measured. This response latency is the time between the cluster's reception of the scatter command from the ground station and the activation of each satellite's thrusters. Multiple software components make up the

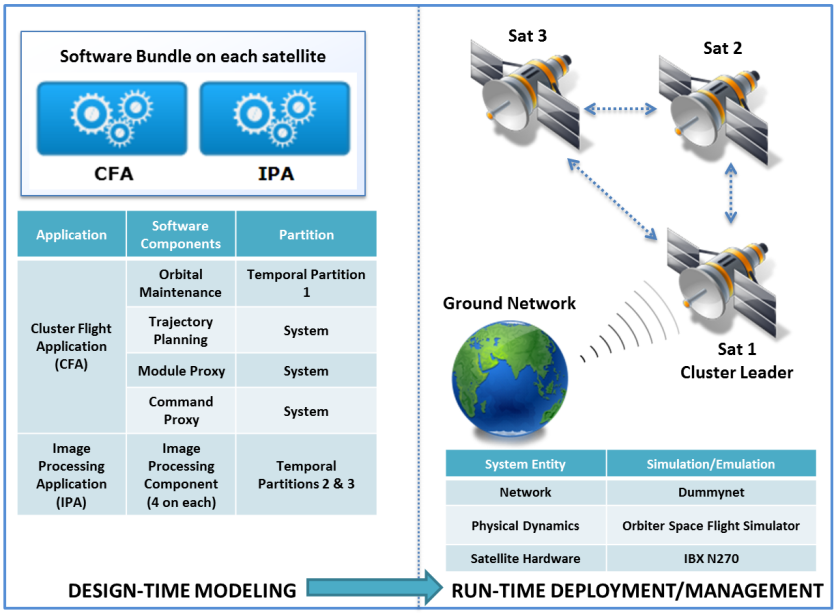

Fig. 5. System setup with the Cluster Flight Application (CFA) and Image Processing Application (IPA). Each satellite has an instance of both applications.

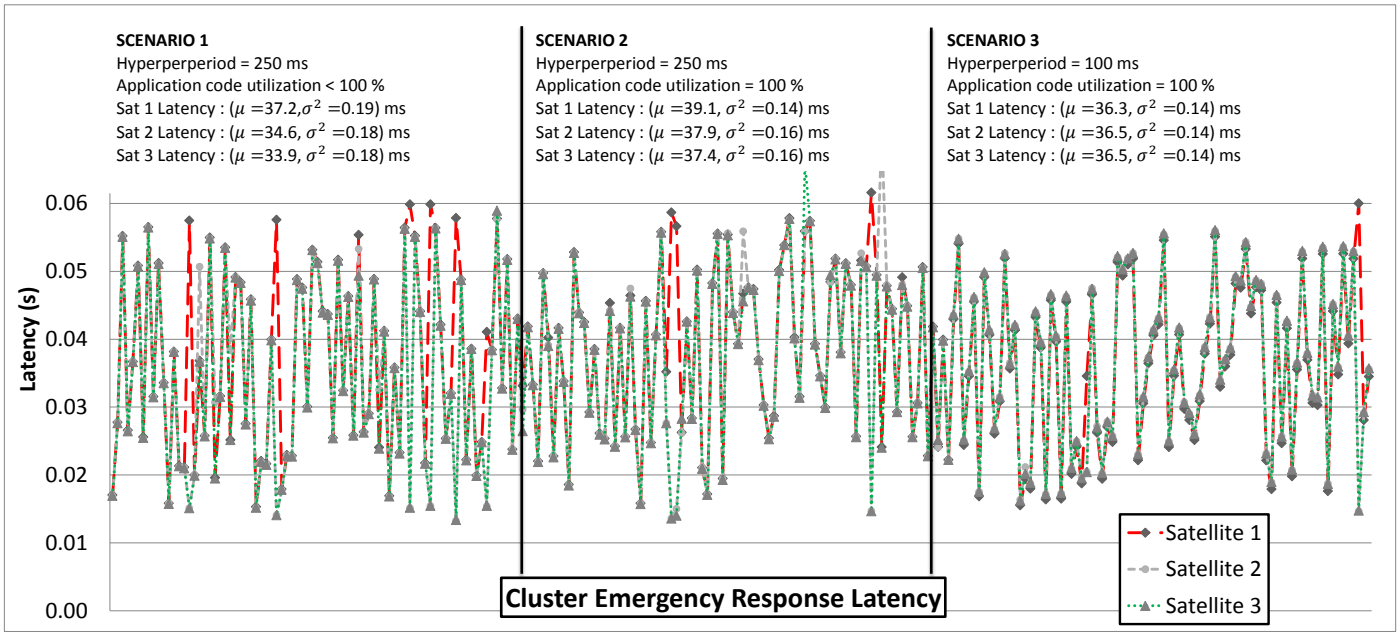

Fig. 6. This is the time between reception of the *scatter* command by satellite 1 and the activation of the thrusters on each satellite. The three regions of the plot indicate the three scenarios: (1) IPA has limited use of its partitions the system's hyperperiod is 250 ms, (2) IPA has full use of its partitions and the system's hyperperiod is 250 ms, and (3) IPA has full use of its partitions and the system's hyperperiod is 100 ms. The averages and variances for the satellites' latencies are shown for each of the three scenarios. The x-axis indicates different sampling points.

CFA and interact to disseminate the command and coordinate thruster activation, so the response time is dependent on the scheduling of the components and the network latency. Using our testbed, we can ensure that the response time is resilient to the combined effects of the component scheduling and the fluctuating network resources that vary with the distance between the satellites.

Figure **??** shows the latency results for each satellite over three scenarios. In each case, Sat 1 received the scatter command from the ground. The latency was measured by invoking the logging facility of the middleware from the application directly before the scatter command was sent and just after the satellite thrusters were activated. This logging framework allows us to rapidly and easily gather timing data from the system which we can analyze against application constraints. These results show that the latency experienced by the CFA is independent of both the system's partition schedule and the resource utilization of the IPA.

V. Related Work

Performance analysis of applications is a well-studied problem, especially in the area of web services and cloud computing. The potential of a model-based resource provisioning method in these environments is demonstrated in [**?**], where simplified analytic models of server memory, storage I/O rate, storage response time and service response time are used to capture the application performance for an informed policy-driven resource allocation vector for complex resource management challenges. An approximate layered queuing model has also been used in [**?**] and [**?**] to capture different performance characteristics and resource contention with the help of a function approximation method while serving a request among multiple tiers. These efforts typically model an entire tier as a queue. Such models are often service-aware, which allows system management decisions involving components and services to be executed. While this approach works well for predictive workloads and closed environments, it can have difficulty scaling in distributed ad hoc computing environments.

Evaluation of large scale distributed systems, particularly on dynamic network platforms, is a non-trivial problem that requires not only a correct model of both the application under test and the network platforms, but also the relationship between the two. Most tools provide simulation support for either distributed service oriented architectures [**?**] or for the network [**?**], but not both in combination. Additionally, related efforts for real-time embedded systems such as [**?**] often describe benchmark results from single node experiments, whereas our framework is aimed at accurate testing and simulation for multi-node configurations.

The ideas behind virtual prototypes [**?**] and related references are very close to the concepts described in this paper. In [**?**], the authors describe the benefits of examining the safe functionality of complex systems early in the design phase. Virtualization technologies such as [**?**], [**?**], [**?**] provide a mechanism to test and develop applications on single nodes. The work described in this paper focuses on a framework to test and evaluate applications under varying network conditions distributed across a cluster of these nodes.

VI. Conclusions

In this paper we have presented a testing framework for the rapid evaluation of distributed applications running on a mobile cluster of nodes with ad hoc networking, where the network characteristics change over time. The framework includes real (or realistic) hardware elements for the computing nodes and an emulated network where the performance of individual network links can be modulated over time. This latter function is driven by a physics simulation engine that simulates the movement of the nodes and calculates the pairwise distance between the nodes, which in turn determines the momentary performance of the network links. We have successfully used this framework for testing and experimentation where the application software has to either satisfy stringent response time requirements or has to be adaptive to network conditions.

The framework offers interesting opportunities for further research and development. Here we list only a few. One is to provide a direct link between the measured momentary network performance and application level adaptation (as a response to changing network conditions). Second, the complete elimination of one or more network links allows the testing of fault tolerance mechanisms in the application software. Third, as the network emulation is agnostic of higher level protocols, it allows experimentation with various IP-based protocols (e.g. SCTP). Arguably, such a combined network emulation/physical system simulation framework offers unique opportunities for developing cyber-physical systems.

Acknowledgment

This work was supported by the DARPA System F6 Program under contract NNA11AC08C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or USAF/AFRL.