DREMS: A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems

Tihamer Levendovszky*, Abhishek Dubey* William R. Otte*, Daniel Balasubramanian*, Alessandro Coglio†, Sandor Nyako*, William Emfinger*, Pranav Kumar*, Aniruddha Gokhale*, and Gabor Karsai*

* ISIS/EECS, Vanderbilt University, Institute for Software-Integrated Systems, Vanderbilt University, 1025 16th Ave S, Ste 102
Nashville, TN 37235, USA
Email:{tihamer,dabhishe,wotte,daniel,snyako,emfinger, pkumar,gokhale,gabor}@isis.vanderbilt.edu

† Kestrel Institute, 3260 Hillview Avenue
Palo Alto, CA 94304
Email: coglio@kestrel.edu

Abstract

Architecting software for a cloud computing platform built from mobile embedded devices incurs many challenges not present in traditional cloud computing. Effective management of constrained resources, application isolation without adversely affecting performance are both needed. This paper describes a practical design- and run-time solution that incorporates modern software development practices and technologies along with novel approaches to address the challenges. Concievably, the patterns and principles manifested in our system can serve as guidelines for current and future practitioners in this field.

Index Terms

D.2.6.b Graphical environments, D.4.7.e Real-time systems and embedded systems, D.4.6.d Information flow controls

I. The emerging realm of mobile and embedded cloud computing

Mobile cloud computing infrastructures supporting the vision of Internet of Things (IoT) [1] provide services beneficial to our society. For example, a cloud of smart phones can run software that shares the sensing and computing resources of nearby devices, providing increased situational awareness in a disaster zone. A cluster of small collaborating satellites can provide increased reliability at reduced launch costs for scientific missions. For instance, NASA's Edison Demonstration of SmallSat Networks, as well as TanDEM-X, PROBA-3, and Prisma from the European Space Agency all use clusters of small satellites.

Unlike traditional computing clouds that draw a clear distinction between a cloud provider and user, these roles will be interchangeable in the participating resources in mobile clouds [2]. Additionally, the need to scale up on demand is often the motivation for using a traditional cloud, whereas a mobile embedded cloud is motivated by the need for on-demand collaboration. Table I presents associated requirements and challenges that are not fully addressed by existing cloud computing platforms. This paper describes an architecture called **D**istributed **RE**altime **M**anaged **S**ystem (*DREMS*) [3] that addresses these requirements. It consists of two main parts: (1) A design-time *toolsuite* for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components; (2) a run-time *software platform* for deploying, managing, and operating application software on a network of embedded,

TABLE I

SUMMARY OF ARCHITECTURAL DECISIONS IN *DREMS*

Requirement	Design Principle	Approach	Section
1. Rapid application development, software reuse	Component-based software engineering	Novel component model	Sec III
2. Multiple application interaction Semantics	Separation of concerns	Rich set of component interaction ports with operations scheduled independently	Sec III
3. Managed concurrency and synchronization for robustness	Single-threaded components	Concurrency managed by OS and middleware, not component business logic	Sec II and Sec III
4. Resource management and application isolation with performance guarantees	Spatial and temporal separation	Applications are run in isolated partitions	Sec II
5. Secure information flows	Multi-level security with multi-domain labels, temporal/spatial isolation, Mandatory Access Control	Architectural support for separation, MLS (based on label checking), and constrained information flows	Sec IV
6. Managed and secure application deployment and configuration	Modeling and automation	Model-driven middleware services to provide secure deployment and configuration	Sec V
7. Producible verified systems	Catch defects early in the development cycle	Model-based system design and generative development	Sec V

mobile nodes. The platform reduces the complexity and increases the robustness of software applications by providing reusable technological building blocks in the form of an operating system, middleware, and application management services (see Figure 1).

II. RUNTIME SOFTWARE PLATFORM: OS AND MIDDLEWARE

DREMS provides a runtime platform for applications in the form of an operating system (OS) and middleware. The *DREMS* OS – a set of extensions to the Linux kernel – provides all the critical low-level services to support resource management (including spatial and temporal partitioning of the memory and the CPU), actor management (discussed below), secure information flows (labeled and managed, as discussed in next section), and fault tolerance.

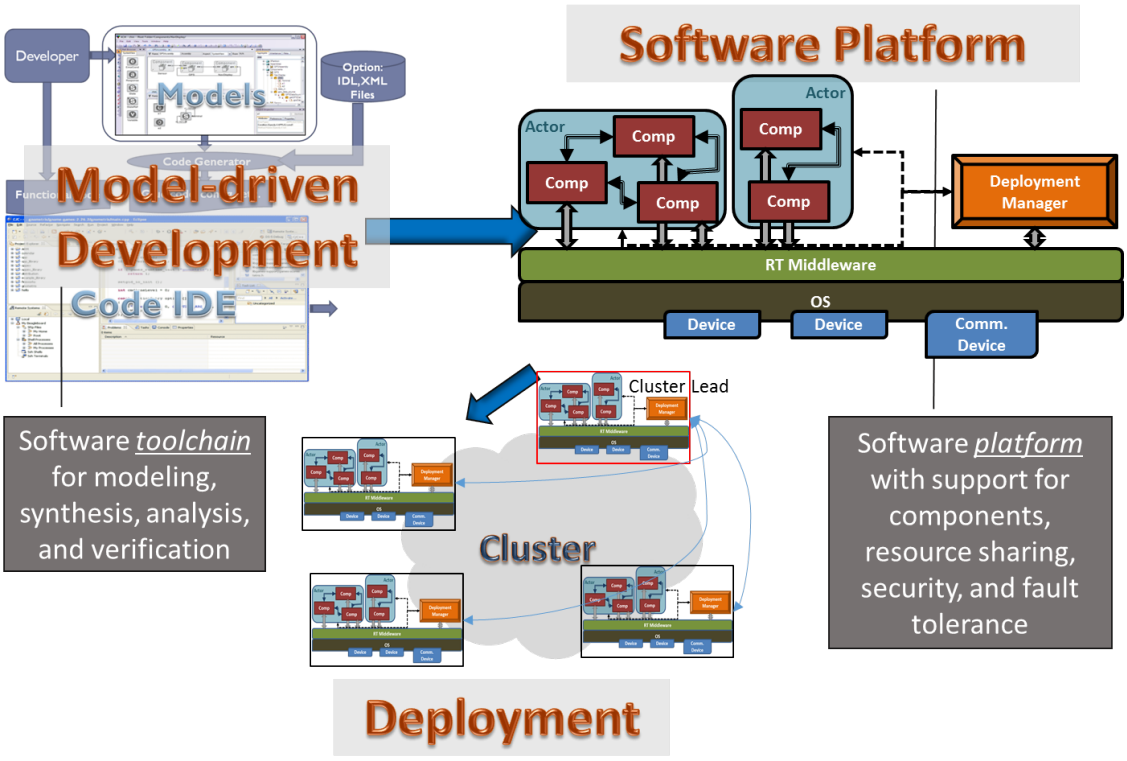

Fig. 1. *DREMS* architecture. The top-right portion shows the internals of one node.

Software applications running on *DREMS* are distributed. To facilitate isolation (Requirement 4), the components that make up an application are encapsulated in process-like containers called *actors* that run concurrently (on the same node) or in parallel (on different nodes). This is similar to the notion of concurrent communicating objects described in [4].

Actors are specialized OS processes; they have a persistent identity that allows their transparent migration between computing nodes. They also have strict limits on the resources that they can use. There are two main types of actors: *application actors* and *platform actors*. Application actors are built for specific applications, while platform actors provide system-level services.

The OS guarantees performance isolation between actors of different applications (Requirement 4). This is done by (a) providing separate, protected address spaces per actor; (b) enforcing that a peripheral device can be accessed by only one actor at a time; and (c) facilitating temporal isolation between actors by the scheduler. The temporal isolation is provided via ARINC-653 [5] style partitions - periodically repeating fixed intervals of the CPU's time exclusively assigned

to a group of cooperating actors of the same application. The scheduler guarantees that actors in distinct temporal partitions cannot inadvertently interfere with each other via CPU usage. Readers are referred to [3] for further details on spatial and temporal isolation, both of which are standard mechanisms.

III. Component model: Building blocks for application development

To address Requirement 1, *DREMS* uses a component-oriented approach for application development [6]. It is accepted that component-based software development promotes rapid application development and reuse [7]. Components have identity, state, support various operations, and interact via ports. A *DREMS* component supports four basic types of communication ports providing a range of interaction semantics (Requirement 2): **Facets** that are collections of operations (interfaces) provided by a component and **Receptacles** that are collections of operation required. These two ports can be used to implement synchronous and asynchronous point to point interactions. In addition, **Publisher** and **Subscriber** ports provide a way for components to interact in a global data space defined over **Topic**s. Conceptually this is similar to the OMG CORBA Component Model (CCM) [8].

However, there are some key differences. The *DREMS* component model provides ports for accessing I/O devices and timers. Ports are implemented using connectors [9] that enable the use of a variety of communication mechanisms, including CORBA and DDS. Furthermore, security using labeled communication (Section IV) is a fundamental part of all component interactions. Another key distinction is the threading model: *DREMS* meets Requirement 3 by enforcing that component activities are scheduled by the midleware as non-preemptible, single-threaded operations that necessitate no synchronization code from the developer. Note that components do run concurrently.

IV. Secure Transport: A secure actor to actor communication channel

DREMS provides a security architecture (Requirements 4 and 6) based on (1) spatial and temporal separation among the actors, (2) fine grained actor privileges that control what system services can be used by an actor, (3) ensuring that only one actor actively controls a device at a time, and (4) a novel communication mechanism among nodes called ‘secure transport’, which supports the exchange of messages among actors according to a Multi-Level Security (MLS)

policy. The combination of separation and MLS guarantee, for example, that an erroneous or malicious actor cannot read information at a higher classification level than its own.

To enforce these rules systemwide, application actors are not permitted to either create new actors or configure secure transport – these activities are performed by the trusted *platform actors*.

A. Endpoints and flows

Actors interact only in controlled ways, which is especially important when they belong to different organizations (e.g. countries). To exchange messages, actors do not reference each other directly. They reference local *endpoints* through which messages are sent and received. An endpoint is analogous to a socket handle in traditional networking systems.

Endpoints in different actors are connected by *flows*, i.e. "pipes" through which messages are transferred (Figure 2). A flow is a connectionless logical association between endpoints: unicast flows connect a source endpoint to a destination endpoint; multicast flows connect a source endpoint to multiple destination endpoints. Both endpoints and flows are created and assigned (only) by trusted platform actors.

Performing message exchanges via endpoints and flows (instead of addressing actors directly) has the following advantages:

- It supports fine-grain communication constraints: two actors can communicate only if there are suitable endpoints and flows.
- It increases decoupling between senders and receivers, which only operate on their local endpoints, without explicit knowledge of the flows attached to those endpoints. For example, the flow connecting a client to a failed server can be switched over to an alternative server transparently to the client.

B. Multi-Level Security (MLS) policy

MLS [10] is a well-established concept. It is based on linearly ordered hierarchical classification levels (e.g. Unclassified $<$ Confidential $<$ Secret $<$ Top Secret) and non-hierarchical need-to-know categories (e.g. mission identifiers). Each organization defines its own levels and categories, i.e. its own labeling domain. In typical systems, which operate in a single labeling domain, a label is a pair LC where L is a level and C is a set of zero or more categories, e.g. in

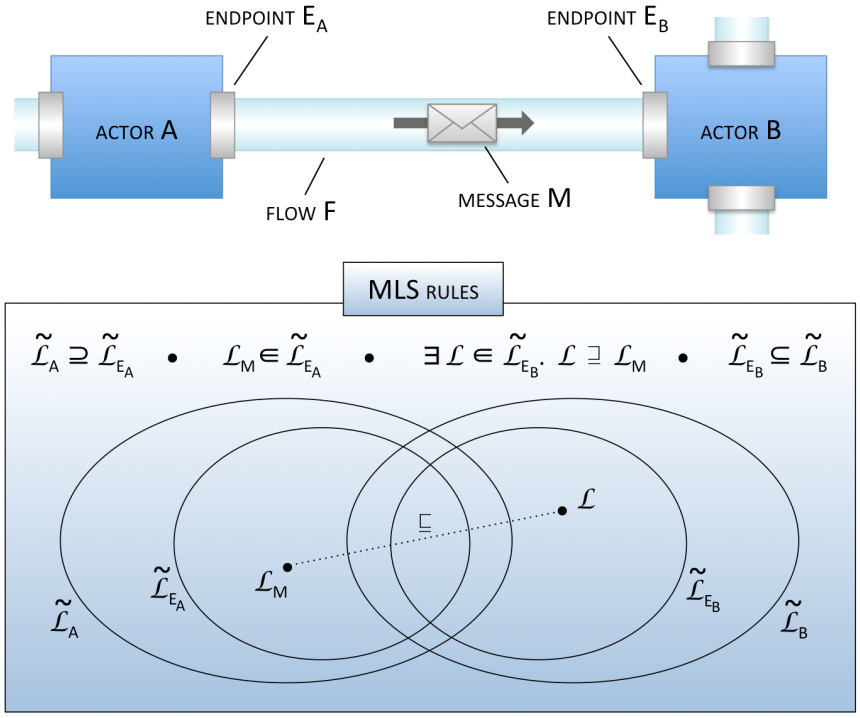

Fig. 2. Transfer of a message via secure transport. The message goes through a flow that connects an endpoint of the sending actor to an endpoint of the receiving actor. The rules on labels and label sets of actors, endpoints, and messages, guarantee the satisfaction of the MLS policy. The MLS rules are illustrated using Venn diagrams.

the US domain, the label $\mathsf{TS}\{\mathsf{x},\mathsf{y}\}$ consists of the level Top Secret and identifiers for missions X and Y.

To support communication among actors from different organizations that can share the common embedded system infrastructure, *DREMS* uses the novel concept of multi-domain labels [11]. A multi-domain label has the form $[D_1]L_1C_1 \ldots [D_p]L_pC_p$, where $D_1, \ldots, D_p$ are $p \geq 1$ distinct (identifiers of) domains and each L_iC_i is a label (as defined in single-domain systems) in domain D_i. For example, the label $[\mathsf{US}]\mathsf{TS}\{\mathsf{x}\}[\mathsf{NATO}]\mathsf{CTS}\{\mathsf{x}\}$ is used for data that is both US Top Secret and NATO Cosmic Top Secret for joint mission X.

The *DREMS* secure transport security policy follows the standard MLS requirement [10] that information can only flow "up", according to the dominance relation. For example, a principal with Top Secret clearance can read Unclassified messages but not vice versa. Data exchanged among different organizations carries labels with levels and categories from all the organizations'

domains. Formally, a label $\mathcal{L}$ dominates a label $\mathcal{L}'$, written $\mathcal{L} \sqsupseteq \mathcal{L}'$, if and only if $\mathcal{L}$ has at least all the domains of $\mathcal{L}'$ (and possibly others) and, for each common domain, the level L in $\mathcal{L}$ is at least as high as the level L' in $\mathcal{L}'$ (i.e. $L \geq L'$) and the category set C in $\mathcal{L}$ contains the category set C' in $\mathcal{L}'$ (i.e. $C \supseteq C'$).

Each actor has an immutable set of labels, which describe the clearance of the actor, i.e. which information the actor is allowed to read and write. The label set is assigned to the actor by (only) trusted platform actors.

Each endpoint $\mathrm{E_A}$ also has an immutable set of labels $\widetilde{\mathcal{L}}_{\mathrm{E_A}}$, which must be contained in the label set $\widetilde{\mathcal{L}}_{\mathrm{A}}$ of the (unique) actor A that owns the endpoint (i.e. $\widetilde{\mathcal{L}}_{\mathrm{E_A}} \subseteq \widetilde{\mathcal{L}}_{\mathrm{A}}$). The label set is assigned to the endpoint by (only) trusted platform actors.

Each message sent via secure transport has an immutable label, which describes the sensitivity of the message. The label is assigned by the actor that creates and sends the message. An actor A can send a message M with label $\mathcal{L}_{\mathrm{M}}$ through an endpoint $\mathrm{E_A}$ with label set $\widetilde{\mathcal{L}}_{\mathrm{E_A}}$ only if $\mathcal{L}_{\mathrm{M}} \in \widetilde{\mathcal{L}}_{\mathrm{E_A}}$.

Figure 2 shows all of these MLS rules. These rules follow the standard MLS policy [10], adapted to secure transport. When actor A attempts to send message M with label $\mathcal{L}_{\mathrm{M}}$ through endpoint $\mathrm{E_A}$, secure transport checks that $\mathcal{L}_{\mathrm{M}} \in \widetilde{\mathcal{L}}_{\mathrm{E_A}}$. When M is received through endpoint $\mathrm{E_B}$ of actor B, secure transport checks that $\mathcal{L} \sqsupseteq \mathcal{L}_{\mathrm{M}}$ for some label $\mathcal{L} \in \widetilde{\mathcal{L}}_{\mathrm{E_B}}$.

C. Networks

When a flow connects endpoints on different nodes, secure transport uses IPv6 [12] to transfer messages across the network, which may involve various wireless networking devices. Without proper protection, messages traveling through the network could be seen or modified, defeating the MLS policy. IPsec [13] and other measures are used to protect the confidentiality of messages (and their labels).

V. Model-Driven application development, integration, and deployment

To simplify development and promote producible and verified systems (Requirement 7), we have developed a model-based framework for *DREMS* for developing and integrating applications. This approach uses *models* to represent the software, the hardware platform, and the mapping between the two. The validation of well-formedness constraints over the models

makes the early detection of integration errors possible. Code generators then translate the validated high-level models into low-level artifacts, such as program code and deployment plans to configure the system.

System integration and deployment (Requirement 6) are also simplified with this approach. Once individual application models are combined, the global system configuration can be generated the same way as a single application configuration. Global system properties, such as timing, can be checked using the integrated models. The graphical modeling language as a technique, along with reusability via templates in the modeling language, also addresses rapid application development (Requirement 1).

Parts A and B of Figure 3 summarize the model-driven development process. During steps 1 and 2, data types are created and used to define the structure and interfaces of individual software components. Multiple implementations of the same component type can co-exist, providing the application developer with alternative implementations. The behavioral logic of the components is entered in step 3 that utilizes model-generated skeleton files. Once a component has been implemented, it can be reused across different applications across different projects. Applications are defined by wiring instances of different components together (step 4).

After all applications are modeled, the system integrator performs steps 5 through 7 (described in part B of Figure 3). Well-formedness (Requirement 7) is ensured by a design constraint checker that analyzes the models and reports violations, including details about the constraints violated and the modeling elements involved.

The *deployment plan* describes all aspects of the application, including the binary libraries required for each component and the meta-data describing those libraries, the secure transport configuration, and the component interactions. This plan is provided to the runtime platform's deployment and configuration service that is responsible for deploying and activating the application on the distributed platform (see example in part C of Figure 3).

VI. Example

To demonstrate the *DREMS*, a complex, multi-node experiment was conducted on a testbed of fanless computing nodes, each containing an Intel Atom N270 clocked at 1.6 GHz and with 1 GB of RAM. The nodes are connected via a private subnet which has a network control node

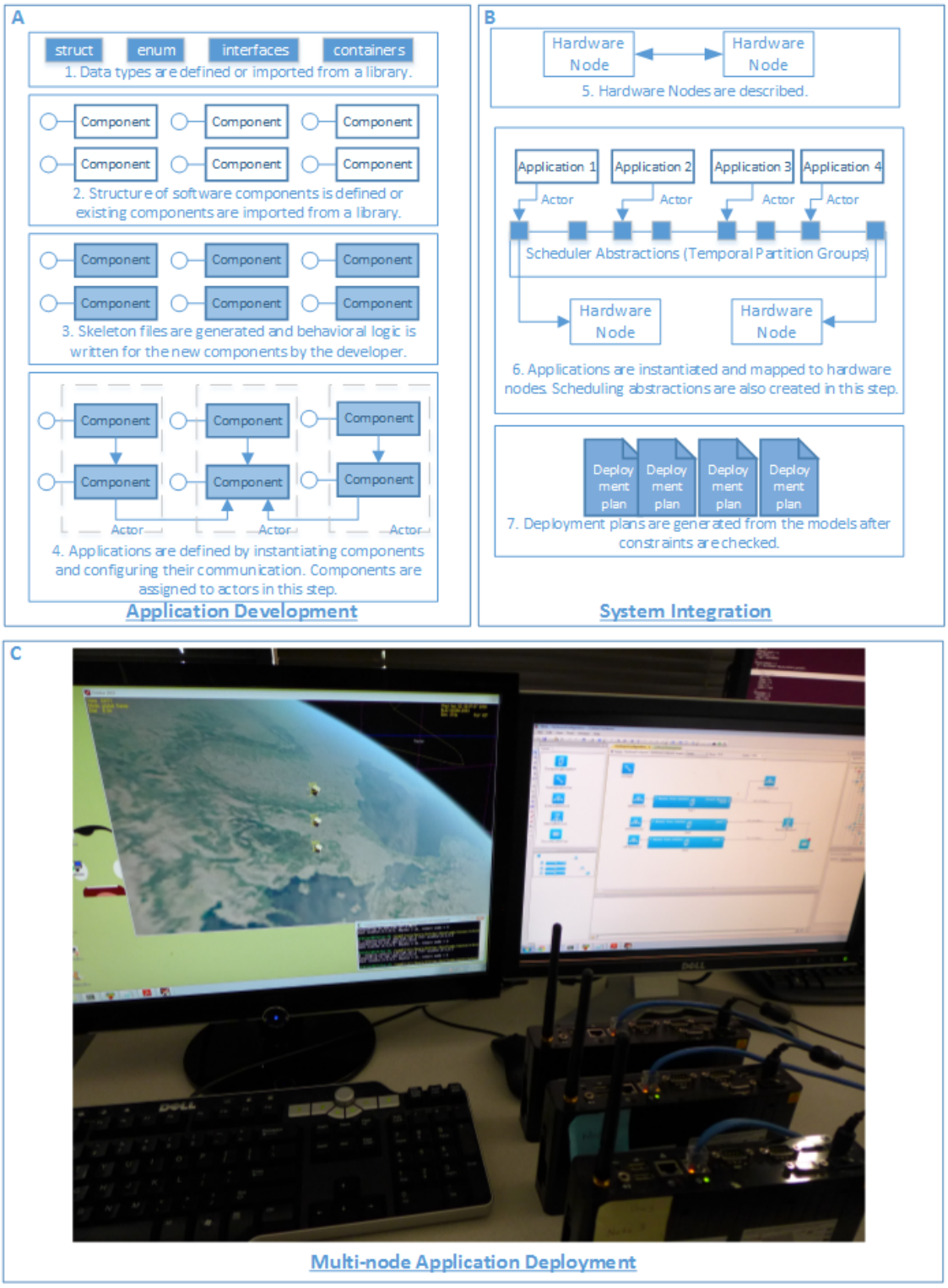

Fig. 3. Application development (A), system integration (B), and deployment on a three-node cluster of embedded processors. The simulator image shows three satellites, while the other display shows the deployment model of the experiment (C).

running dummynet [14], allowing full control of the bandwidth, latency, and packet loss on any network link (see bottom of Figure 3).

On this testbed, a cluster of three satellites was emulated, each running a copy of a cluster flight control application (CFA). CFA consists of three actors replicated on each satellite: *OrbitalMaintenance*,*ModuleProxy*, and *CommandProxy*. *ModuleProxy* connects to the Orbiter space flight simulator [15], which simulates the satellite hardware and orbital behavior. *CommandProxy* receives commands from the ground network. *OrbitMaintenance* keeps track of every satellite's position and updates the cluster with its current position.

Each node publishes a state vector describing its position and subscribes to the state vectors of all other satellites. Individual state vectors are periodically updated on each satellite through an AMI interface from *ModuleProxy* to *OrbitMaintenance*. This interaction represents the flight hardware periodically updating the control software with a new satellite state. The connection between Orbiter and *ModuleProxy* facilitates periodically getting position data from the satellite sensors.

When *OrbitMaintenance* receives a command from *CommandProxy*, it publishes the command as a *Satellite_Command* topic. The *OrbitMaintenance* actor on each satellite subscribes to the *Satellite_Command* topic, and upon reception of the topic, instructs the satellite thrusters to fire (via an AMI call to *ModuleProxy*), which activates the satellite thruster in the simulation.

Despite the complexity of the application, only 405 total lines of code (0.41% of the application code) were written by hand between the four components. The other 99.59% is generated code that governs all communications, timing, and interactions.

VII. Discussion

There certainly exist state-of-the-art development environments and run-time platforms that address some of the needs discussed earlier. There are model-based development environments for embedded systems (*e.g.*, Mathworks's toolsuites, IBM's UML tools, etc.), there are various real-time operating system products with sophisticated development toolchains (*e.g.*, Integrity by Green Hills), and there are systems that support Multi-Level Security (*e.g.*, SELinux). However, to the best of our knowledge we are not aware of any single development environment and run-time platform that holistically provides all these capabilities in one package.

In our experiments, we found that emerging cloud paradigms for mobile devices can be supported through a managed runtime platform with integrated support for multi-level security and advanced component models. A model-based development environment that abstracts the runtime platform and automatically generates the required interface code eases the burden of developing applications for a new platform.

Sidebar 1: Further Reading

- DREMS page at ISIS: http://www.isis.vanderbilt.edu/DREMS
- F6 Project Page at Kestrel Institute: http://www.kestrel.edu/home/projects/f6/
- Generic Modeling Environment project page: http://www.isis.vanderbilt.edu/Projects/gme

Acknowledgments: This work was supported by the DARPA System F6 Program under contract NNA11AC08C. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of DARPA. The authors thank Olin Sibert of Oxford Systems and all the team members of our project for their invaluable input and contributions to this effort.

REFERENCES

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010.

[2] Owen Brown, P Eremenko, and C Roberts. Cost-benefit analysis of a notional fractionated satcom architecture. In *Proc. of the 24th AIAA International Communications Satellite Systems Conference, AIAA-2006-5328, San Diego, CA*, 2006.

[3] Abhishek Dubey, William Emfinger, Aniruddha Gokhale, Gabor Karsai, William Otte, Jeff Parsons, Csanad Szabo, Alessandro Coglio, Eric Smith, and Prasanta Bose. A Software Platform for Fractionated Spacecraft. In *Proceedings of the IEEE Aerospace Conference, 2012*, pages 1–20, Big Sky, MT, USA, March 2012. IEEE.

[4] Rajesh K. Karmani and Gul Agha. Actors. In *Encyclopedia of Parallel Computing*, pages 1–11. 2011.

[5] ARINC Incorporated, Annapolis, Maryland, USA. *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, January 1997.

[6] William R. Otte, Abhishek Dubey, Subhav Pradhan, Prithviraj Patil, Aniruddha Gokhale, Gabor Karsai, and Johnny 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, June 2013.

[7] Clemens Szyperski. Component Technology: What, Where, and How? In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[9] William R. Otte, Aniruddha Gokhale, Douglas C. Schmidt, and Johnny Willemsen. Infrastructure for Component-based DDS Application Development. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 53–62, New York, NY, USA, 2011. ACM.

[10] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, Volume I, MITRE, 1973.

[11] Olin Sibert. Multiple-domain labels. Presented at the F6 Security Kickoff, 2011.

[12] RFC 2460: Internet Protocol, version 6 (IPv6) specification, December 1998.

[13] S. Kent and K. Keo. IETF RFC 4301: Security architecture for the internet protocol, dec 2005.

[14] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Computer Communication Review*, 40(2):12–20, April 2010.

[15] Bruce Irving. Playing in space: Interactive education with the orbiter space flight simulator. In *International Space Development Conference (ISDC) 2007*, 2007.

VIII. Author Information

Tihamer Levendovszky is a Research Assistant Professor at Vanderbilt University. He received his PhD from the Budapest University of Technology and Economics. His interests include model-based engineering and performance analysis of software systems.

Abhishek Dubey is a Research Scientist at ISIS at Vanderbilt University. He received his PhD in Electrical Engineering from Vanderbilt University. His interests include distributed fault-tolerant real-time systems and autonomic computing.

William R. Otte is a Research Scientist at ISIS at Vanderbilt University. He received his PhD in Computer Science from Vanderbilt University. His interests include middleware for real-time embedded systems and deployment and their configuration.

Daniel Balasubramanian is a Research Scientist at ISIS at Vanderbilt University. He received his PhD in Computer Science from Vanderbilt University. His interests include the lightweight application of formal methods and analysis to model-based development.

Alessandro Coglio is a Principal Scientist at Kestrel Institute. He received a degree in Informatics Engineering from University of Genoa, Italy. His interests are formal methods and tools to develop correct-by-construction software via formal specification, refinement, and theorem proving.

Sandor Nyako is a Senior Research Engineer at Vanderbilt University. He received his BSc degree at Eotvos Lorand University, Hungary. Sandor has over 13 years of experience in the telecom, finance and computer entertainment fields.

William Emfinger is a Graduate Research Assistant at ISIS at Vanderbilt University. His research focuses on networking for critical systems. He received his B.E. in Electrical Engineering and Biomedical Engineering from Vanderbilt University in 2011.

Pranav Srinivas Kumar is a Graduate Research Assistant at ISIS at Vanderbilt University. His research focuses on modeling, analysis and verification techniques for distributed component-based software applications. He received his B.E. in Electronics and Communications Engineering from Anna University, India in 2011.

Aniruddha S. Gokhale is an Associate Professor in the Department of Electrical Engineering and Computer Science, and Senior Research Scientist at ISIS, Vanderbilt University. He received his PhD from Washington University, St. Louis. Dr. Gokhale is a Senior member of both IEEE and ACM.

Gabor Karsai is Professor of Electrical and Computer Engineering and Computer Science at Vanderbilt University and Senior Research Scientist at ISIS. He conducts research in model-integrated computing (MIC), design automation for model-driven development processes, automatic program synthesis, and the application of MIC in various government and industrial projects. He is a senior member of the IEEE Computer Society.