A Software Platform for Fractionated Spacecraft

Abhishek Dubey, William Emfinger, Aniruddha Gokhale, Gabor Karsai, William R. Otte, Jeffrey Parsons, Csanád Szabó
Institute for Software Integrated Systems
Vanderbilt University
1025 16th Ave S, Suite 102
Nashville, TN 37212
{dabhishe, emfinger, gokhale, gabor, wotte, parsons, csanad}@isis.vanderbilt.edu

Alessandro Coglio and Eric Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
{coglio, eric.smith}@kestrel.edu

Prasanta Bose
Advanced Technology Center (ATC)
Lockheed Martin Space Systems Company
3251 Hanover Street
Palo Alto, CA 94304
prasanta.bose@lmco.com

Abstract—**A fractionated spacecraft is a cluster of independent modules that interact wirelessly to maintain cluster flight and realize the functions usually performed by a monolithic satellite. This spacecraft architecture poses novel software challenges because the hardware platform is inherently distributed, with highly fluctuating connectivity among the modules. It is critical for mission success to support autonomous fault management and to satisfy real-time performance requirements. It is also both critical and challenging to support multiple organizations and users whose diverse software applications have changing demands for computational and communication resources, while operating on different levels and in separate domains of security.**

The solution proposed in this paper is based on a layered architecture consisting of a novel operating system, a middleware layer, and component-structured applications. The operating system provides primitives for concurrency, synchronization, and secure information flows; it also enforces application separation and resource management policies. The middleware provides higher-level services supporting request/response and publish/subscribe interactions for distributed software. The component model facilitates the creation of software applications from modular and reusable components that are deployed in the distributed system and interact only through well-defined mechanisms.

Two cross-cutting aspects - multi-level security and multi-layered fault management - are addressed at all levels of the architecture. The complexity of creating applications and performing system integration is mitigated through the use of a domain-specific model-driven development process that relies on a dedicated modeling language and its accompanying graphical modeling tools, software generators for synthesizing infrastructure code, and the extensive use of model-based analysis for verification and validation.

TABLE OF CONTENTS

ABSTRACT 1
1 INTRODUCTION 1
2 THE OPERATING SYSTEM: F6OS 2
3 THE MIDDLEWARE: F6ORB 5
4 THE COMPONENT MODEL: F6COM 7
5 PLATFORM ACTORS 10
6 FAULT MANAGEMENT 12
7 SECURITY 16
8 MODEL-DRIVEN DEVELOPMENT 16
9 RELATED WORK 18
10 CONCLUSIONS 18
ACKNOWLEDGMENTS 18
REFERENCES 18

978-1-4577-0557-1/12/$26.00 ©2012 IEEE.

1. INTRODUCTION

A fractionated spacecraft distributes its functions across multiple *modules* that communicate via wireless links and that fly as a *cluster* [1]. Modules may be heterogeneous and may contain multiple sensors and computing *nodes*. The architecture is highly dynamic; the cluster deployment can include varying mission applications that take advantage of the computing and sensing devices available in arbitrary configurations. The cluster architecture itself may change; modules can join and leave the cluster at any time. Furthermore, the architecture must exhibit fault tolerance: if a software application, sensor, node, or module fails, the cluster must (preferably autonomously) reconfigure itself and continue operations. Finally, the spacecraft could serve as an open platform for executing multiple missions for various stakeholders whose information flows must be strictly regulated by security policies.

The above challenges have a significant impact on the design of the software architecture. Ultimately, the software infrastructure provides the platform for system integration; over the lifetime of the system, the physical structure changes infrequently (with the exception of adding and removing modules), but the deployed software applications and the information flows that connect them to the sensing and communication resources are highly variable. This necessitates a systems engineering approach where the software system itself is designed for change and evolution.

Advanced software and systems engineering approaches are based on *models*. With the advent of UML for software systems and SysML for systems in general, model-driven development and engineering practices are gaining acceptance. While modeling is fundamental to all engineering, the explicit use of models in implementation (synthesis) and verification (analysis) has only recently started to appear in the industry. The model-driven engineering of software systems is codified in the OMG's vision for Model-Driven Architecture (MDA) [2].

This paper introduces an information architecture called

F6MDA (F6[1] MDA) for fractionated spacecraft. F6MDA consists of (1) a layered component-based software platform that provides resource management, fault management, and multi-level security (MLS) [3] services, and (2) a suite of model-driven development tools that support software application development and system integration. As shown in Figure 1, the layers of the software platform are: (i) *F6OS*, an operating system that provides core abstractions for concurrency, synchronization, resource management, and secure communications; (ii) *F6ORB*, a middleware layer that implements the essential communication services for the distributed system; and (iii) *F6COM*, a component model that defines how components are built and how applications are constructed from components. Components are grouped into *actors*, which are temporally and spatially isolated from each other, and may be distributed and replicated across nodes. *Application actors* form applications; one application may be split across multiple application actors, potentially on different nodes and modules. *Platform actors* provide system-level services.

There are a number of cross-cutting aspects in the architecture: resource management, fault management, and MLS. These are addressed partially by each layer, and partially by cross-layer dedicated services. The architecture is complex, hence developers need the support of model-driven development tools that facilitate the high-level specification of components, interfaces, and the architecture of the applications. Such models are used to synthesize the engineering artifacts needed to build and deploy the applications on the software platform.

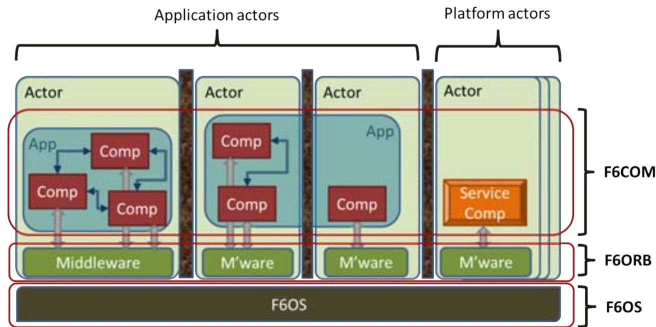

Figure 1. Overview of F6MDA

The paper is structured as follows. First, we describe the three layers of the software platform: F6OS, F6ORB, and F6COM. Then we describe platform actors. Next, two cross-cutting issues are discussed: fault management and the security services. After this, we present the model-driven development process and the tools used. Finally, we summarize essential related work and conclude the paper.

2. The Operating System: F6OS

F6OS provides the necessary abstractions to manage the resources available to all applications: CPU, memory, network, file system space, and threads. It also provides the low-level computing and communication services needed to run applications according to MLS policy. An instance of F6OS runs on each node of each module of a cluster, as well as on each ground node that communicates with the cluster.

[1]Short for 'Future, Fast, Flexible, Fractionated, Free-Flying Spacecraft united by Information eXchange' [1].

F6OS enforces strict limits on resource use, ensures fault isolation among applications, and weakens the opportunities for information leakage. This behavior is achieved by the partitioning of computing resources: CPU time and memory. Partitioning has been extensively used in avionics systems (*cf.* Integrated Modular Avionics [4]) and provides strong firewalls between applications for fault and information containment. Faults introduced in one partition do not propagate to other partitions. Information flows across partitions can take place only via mechanisms provided by the operating system, and all shared resources (*e.g.*, peripheral devices) must be encapsulated into their own partitions to avoid unintended interference among partitions. Figure 2 shows the schematic of F6OS.

Threads and Actors

A *thread* is the smallest unit of execution in F6OS. F6OS threads are similar to threads in typical operating systems. F6OS views an *actor* as a collection of threads that share a unique, isolated memory space. The relationship between threads and components is discussed in Section 4. F6OS actors are similar to processes in typical operating systems. F6OS ensures that actors are spatially separated, that is, no two actors share any memory space. As usual in modern operating systems, actors run in user space, which is separated from kernel space where F6OS runs. User space is virtualized: each actor sees an independent copy of the user space. An application is a collection of actors, possibly running on different nodes. Note that F6OS has no notion of an 'application', just that of an 'actor'. Which actors constitute which applications is a notion that is external to F6OS.

There are two kinds of actors:

- *Application actors*, which make up applications and perform mission-specific tasks. They can be dynamically installed and removed. Application actors include services that encapsulate mission-specific devices (*e.g.*, a camera).
- *Platform actors*, which complement and extend F6OS by providing services that are essential for other actors to run on the node. They run in user space but can make system calls (*e.g.*, to install application actors) that application actors cannot make.

Platform actors perform functions that have complex implementation logic that is developed and verified independently of F6OS. These functions are kept separate from F6OS for efficiency reasons. Modern operating systems often rely on such capabilities (*e.g.*, daemons in Linux, services in Windows). This approach also provides more security, because a faulty or compromised platform actor will cause less damage. There are six platform actors, shown in Figure 2. Platform actors are present on all nodes, and they are in the Trusted Computing Base (TCB) along with F6OS. F6OS and the platform actors form the foundation that supports application actors, as described in Section 5.

Partitions and Scheduling

A (*temporal*) *partition* is a periodically repeating interval that provides a slice of CPU time to one or more actors. (Allowing several actors to share a partition can lead to more efficient use of the CPU, but this choice has security implications as described in Section 7.) Actors in different partitions are temporally firewalled from each other, and even actors in the same temporal partition are spatially separated (separate memory spaces and file systems). At most one partition is running on the CPU at any given time, and each application

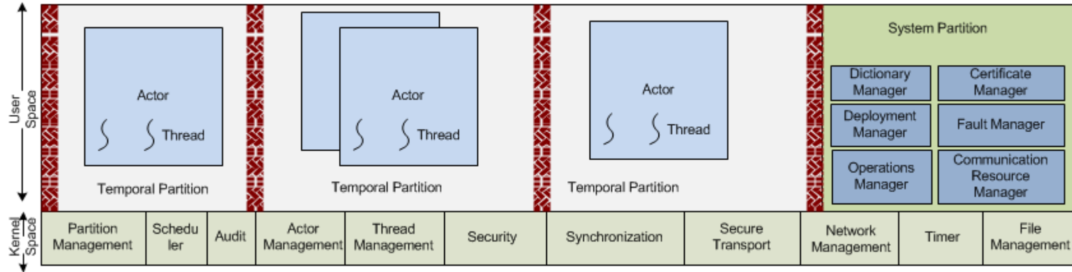

Figure 2. Overview of F6OS

actor belongs to a single partition. Each partition has a *period* and a *duration*, which determine cyclically repeating time intervals in which the partition's actors can run. Platform actors run in the *system partition* and are not subject to temporal partitioning. That is, they run on demand and concurrently with any application actor active at the moment.

The scheduler is responsible for maintaining temporal partitioning and scheduling the active threads. It is also responsible for managing the relative CPU utilization limit imposed on an actor. The F6OS scheduler divides CPU time into a periodically repeating sequence of time intervals called *hyperperiods*. The hyperperiod value is calculated as the least common multiple of the periods of all temporal partitions. Thus, every partition executes one or more times within a hyperperiod. The temporal interval associated with a hyperperiod is known as a *major frame*. The major frame is subdivided into several smaller time windows called *minor frames*, each of which belongs exclusively to one partition. The length of each minor frame is the same as the duration of the corresponding partition. A partition is awakened at the beginning of its minor frame and put into a dormant state at each frame's end.

The F6OS scheduler has the following states:

- *Initial*, when the system boots up. Upon boot, the platform actors are initialized. Once all the platform actors are running, the Deployment Manager (described in Section 5) deploys and runs the initial set of application actors as specified by the initial deployment plan;
- *Active*, when the application actors are running. The scheduler keeps executing the partition schedule;
- *Paused*, in which only the system partition is scheduled so that platform actors can run (to change the scheduling table).

System Calls

Actors interact with F6OS exclusively via *system calls*, which cross the boundary between user and kernel space and always return within a bounded time—each system call has a timeout. Some system calls are *privileged*, that is, they can be made only by specific platform actors, and not by any application actors.

System calls are divided into the categories listed in Table 1, where regular and privileged system calls are distinguished. The table begins with the most common categories of calls that are available to any actor. These are followed by categories that have both regular and privileged calls. The table ends with the categories that contain only privileged system calls. The platform actors allowed to access the privileged calls of each category are listed in the table as well. All application and platform actors can access all the regular system calls.

Regular	Privileged	Platform Actors
Timer		
Synchronization		
Actor Management	Privileged A. M.	Deployment Manager
Thread Management	Privileged T. M.	
File Management	Privileged F. M.	
Secure Transport	Privileged S. T.	
Audit	Privileged Audit	
	Partition Management	
	Scheduling	
	Security	
	Actor Management Notification	Fault Manager
	Network Management	Communication Resource Manager

Table 1. System call categories in F6OS

Restricted access to system calls is enforced by F6OS based on the identity of the calling actor—F6OS knows the identities of the platform actors, which are always present in the node. These restrictions ensure that an application actor cannot use privileged system calls to change the system behavior in a way that can affect other actors or create covert channels (*e.g.*, by manipulating partitions).

MLS and Labels

In order to clarify subsequent discussion, we introduce the basic concepts of MLS and labels. MLS is typically based on n linearly ordered, hierarchical classification *levels* (*e.g.*, Unclassified $<$ Confidential $<$ Secret $<$ Top Secret) and m non-hierarchical need-to-know *categories* (*e.g.*, mission identifiers) [3]. Levels and categories give rise to $n \times 2^m$ security *labels*: a label is a pair $\langle L, C \rangle$ where L is a level and C is a set of categories. A label $\mathcal{L}_1 = \langle L_1, C_1 \rangle$ *dominates* a label $\mathcal{L}_2 = \langle L_2, C_2 \rangle$ iff $L_1 \geq L_2$ and $C_1 \supseteq C_2$, *i.e.*, $\mathcal{L}_1$ has at least the same level and categories as $\mathcal{L}_2$. Labels and the domination relation form a lattice [5].

The MLS *policy* is that information can flow only from lower to higher or between equal labels (according to the domination relation), *e.g.*, a Secret process for mission A can read Confidential or Secret data for mission A, but not Top Secret data for mission A or Secret data for mission B. In practical systems, *trusted* applications are allowed to transfer information against the policy, *e.g.*, to declassify sanitized data.

F6MDA supports applications from different organizations on the same cluster. Different organizations may have different labeling *domains*, *e.g.*, NATO and the US have slightly different levels. If applications from different domains are isolated, a global MLS policy reduces to the individual do-

mains' MLS policies. However, if applications with different domains need to exchange data (*e.g.*, for cooperative space missions), a global MLS policy is less straightforward. A typical approach is to define mappings between the labels of the different domains (*e.g.*, US Secret could be mapped to NATO Secret) and to re-label data that crosses domains according to those mappings.

F6MDA's approach is to use *multiple-domain labels* [6]: a multiple-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 $\langle L_i, C_i \rangle$ is a label in D_i as defined earlier, consisting of a level L_i and a set of categories C_i. In the rest of this paper, the stand-alone term 'label' exclusively denotes a multiple-domain label. Data exchanged among multiple domains carries labels with levels and categories for all those domains ($p > 1$), *e.g.*, $[\mathtt{US}]\mathtt{TS}\{\mathtt{A}\}[\mathtt{NATO}]\mathtt{CTS}\{\mathtt{A}\}$ labels data that is both US Top Secret and NATO Cosmic Top Secret for joint mission A. Data used exclusively within one domain carries labels with levels and categories for just that domain ($p = 1$), *e.g.*, $[\mathtt{US}]\mathtt{C}\{\mathtt{B}\}$ labels data that is US Confidential for mission B.

We have used the Isabelle/HOL theorem prover [7] to formalize the syntax and semantics of labels and to prove that they form a lattice. Domination for multi-domain labels is defined as domination in every individual domain, and the absence of a domain in a label is equivalent to its presence with its minimum level and the empty set of categories. Details of the formalization and proofs will be published in a separate paper.

Actor Communications: Secure Transport and Networking

F6OS provides the mechanisms of *endpoints* and *flows* to support communication between actors deployed on the same node or on different nodes. An endpoint is a logical "port" that an actor can read/write *messages* from/to, via system calls. A flow logically connects a write endpoint to one or more read endpoints. The mechanism that securely delivers messages between endpoints through flows is called *Secure Transport* and is built into F6OS. All actors, including the platform actors, lack any other means to communicate with each other. Endpoints and flows are created by the Deployment Manager, as it implements a deployment plan.

F6OS assigns to each actor and to each endpoint a set of one or more labels and enforces the constraint that an actor's labels must include all the labels of the actor's endpoints. Each message has a security label, chosen by the sending actor from among the write endpoint's labels, which are among the sending actor's labels. A message is delivered to a read endpoint only if the message's label is among the read endpoint's labels, which are among the receiving actor's labels. Thus, message exchanges comply with MLS.

MLS does allow information to flow from lower to strictly higher labels. However, since a message has a label that must be among both the sending and the receiving actor's labels, F6OS does not allow, for example, an actor with a single label $\mathcal{L}$ to send a message directly to an actor with a single label $\mathcal{L}'$ that strictly dominates $\mathcal{L}$. A third actor with both labels $\mathcal{L}$ and $\mathcal{L}'$ must mediate the transmission, by re-labeling the message from $\mathcal{L}$ to $\mathcal{L}'$, which complies with MLS. This is a design choice: the complication of having mediating actors is balanced by the increased simplicity of the Secure Transport and avoids the question of whether the message should be labeled $\mathcal{L}$, $\mathcal{L}'$, or both.

The networking layer of F6OS is abstracted away from the actors by the Secure Transport system calls. From the actors' perspective, messages are sent on the write endpoint and received on the read endpoint. The mechanism of communication remains the same, except for differences in latency and bandwidth, whether communicating actors are on the same node or on different nodes.

When a message is exchanged between two actors on different nodes, the message and its label are transmitted across the network, cryptographically protected end-to-end, between the two nodes. Each node has a public key, used to establish the needed cryptographic protection (*e.g.*, using IPsec). The sending node checks that the message's label is among the read endpoints' labels on the receiving nodes before sending the message; for robustness and protection from compromise, the receiving node repeats that check on every incoming message.

Each flow is associated with a Quality of Service (QoS) class. This association is made during the creation of the flow. Along with the QoS class, F6OS provides the ability to associate QoS parameters (*e.g.*, priority, maximum packet rate) with a flow. F6OS uses these parameters to put the message in the correct rate-based device queue or limit the network usage of an actor. If an actor exceeds its bandwidth limits it receives an error. F6OS drops messages for the time interval in which the actor tries to send data exceeding its bandwidth limits. The actor is responsible for limiting (if necessary, decreasing) its network usage.

The Communication Resource Manager (CRM) is the platform actor responsible for managing the network resource. Each module has a wireless network interface that connects it to other modules. The quality of the network service is highly variable due to the orbital mechanics of the modules. It is also a very precious resource that requires access control. The CRM achieves such access control by managing classification rules in order to prioritize packets and to control the utilization of a priority class. The CRM manages the bandwidth access and utilization of local actors per mounted network device. It also implements the resource sharing policies for a given application concerning: (1) authorized communication links that the actor is allowed to use, (2) maximum transmit bandwidth to be used by the actor (per mounted network device), and (3) classification rules for a given endpoint. An actor can communicate through different endpoints with different QoS requirements.

The CRM continuously monitors link quality (as reported by the networking device), and modifies the routing tables and classification rules (thus the logic for admission control and routing), optimizing the utilization of the network. CRMs deployed on the cluster also share information with each other to optimize the performance of the network. The main driving requirement for the optimization is the changing bandwidth on the various wireless links. Through CRM coordination, congestion control can be performed as well.

F6OS is designed to support communication over the hierarchical networks that are present in the context of fractionated spacecraft. At the module level, nodes will be interconnected by a wired local area network. A dedicated node equipped with a wireless inter-module communication device will act as a gateway to other modules in the cluster. At the cluster level, modules may have several options to reach the terrestrial network. A module may use a direct ground link that provides a high bandwidth connection within a limited

time window. Modules with actors that have permanent availability requirements may use a Broadband Global Area Network (BGAN) [8] link. Admission control to these links is performed per actor by CRMs on the gateway modules.

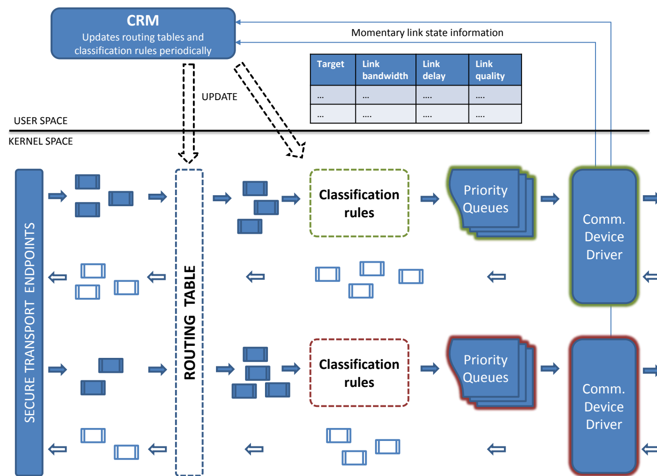

Figure 3. Overview of F6OS networking

Figure 3 gives an overview of the F6OS networking layer, which supports three use cases: transmit, receive, and forward. The routing table, which is periodically refreshed by the CRM, determines the network device for the transmit and forward cases and the read endpoint for the receive case. The classification rules are maintained per mounted network device. Based on these rules, F6OS inserts each message into the priority queue that conforms to the QoS class of the message. This classification is done only in the transmit and forward cases, after the routing decision is made.

File And Auditing Services

In addition to assigning a separate address space to each actor, F6OS also assigns a separate, isolated, and quota-controlled file system to each actor.

Moreover, F6OS logs audit data about the system calls it receives and their outcomes. Audit settings are done via privileged system calls. Furthermore, F6OS provides system calls for actors to log actor-specific audit data.

Audit data is stored in internal files by F6OS. Audit data is partitioned by actor: the audit data generated by a system call is added to the audit log of the actor that makes the system call. An actor can only append new data to its audit log; it cannot read, modify, or delete existing data in any audit log (with the exception of the Operations Manager, as described in Section 5).

To prevent audit data flooding, privileged system calls can set a cap on the size and rate of audit data produced by an actor. If the maximum size is reached, the oldest audit data is overwritten by the newest audit data. If the maximum rate is reached, the actor's execution is paused until the next minor frame for that actor.

3. THE MIDDLEWARE: F6ORB

Commonly available operating system and network layer services are low-level, and do not, in themselves, provide support for good reusable abstractions and design patterns for effective software development. A middleware layer can provide such reusable abstractions and services, which reduces complexity and increases productivity of software developers. In F6MDA, developers build actors using F6ORB, a middleware that (1) hides lower-level F6OS services (in particular, the Secure Transport), (2) specifies a well-defined set of communication patterns, both local and remote, that reduce complexity, and (3) provides a set of reusable, generic services that all actors can use. As shown in Figure 1, F6ORB is part of user space actors. However, it is implemented as a reusable library.

F6ORB provides two communication mechanisms between actors, described in the next two subsections:

- Synchronous and asynchronous point-to-point communication, built using a subset of the Common Object Request Broker Architecture (CORBA) standard [9]. This interaction is mostly suitable for control or management aspects, where a request is often accompanied by a response.
- Anonymous publish/subscribe, built using a subset of the Data Distribution Service (DDS) standard [10]. This interaction is ubiquitous and provides one-to-many, many-to-one, and many-to-many distribution patterns for transmitting data in a global data space. Often, due to the nature of the distribution pattern, this interaction is inefficient to implement with a point-to-point communication approach.

Synchronous and Asynchronous Point-to-Point Communication

The CORBA standard is a mature distributed object computing middleware that is standardized by the Object Management Group. CORBA provides robust facilities for automating many important tasks in F6MDA, namely:

- *Location transparency*: Actor code need not be concerned with the location of objects that it interacts with. From the perspective of the actor, it may interact with remote objects as if they were local. The ORB, in conjunction with the Deployment Manager, handles the mechanics of transparently routing the request to the remote host.
- *Request (de)multiplexing*: In the likely event that an actor provides multiple interfaces that are part of the same logical connection between two hosts, the ORB and Object Adapter ensure that requests and responses are routed to the correct destination.
- *Error handling*: The ORB can automate many common error handling tasks in distributed actors, including: request timeouts, re-tries in the event of transient failures, etc. Moreover, CORBA provides an exception facility that can be used to communicate both application- and system-level errors to application logic in a descriptive manner.
- *Parameter (de)marshaling*: CORBA includes a well defined mechanism for serializing and deserializing complex data types for transmission on the network, called *Common Data Representation*.

The real-time nature of the F6MDA environment requires that F6ORB provide deterministic and predictable behavior, insofar as possible given changing network conditions. As a result, F6ORB explicitly excludes many of the dynamic features of CORBA that are not required for the fractionated spacecraft environment and would unnecessarily complicate the middleware implementation. A recent OMG standard, CORBA for Embedded [11] (CORBA/e), provides an excellent starting point for reducing the profile of the CORBA specification. F6ORB adheres to a modified version of the CORBA/e Micro profile, described below, which excludes support for most dynamic CORBA features and optional

CORBA services for event channels, load balancing, security, and naming. CORBA/e Micro explicitly excludes support for the CORBA `Any` datatype, which allows an arbitrary IDL[2] datatype to be stored. However, F6ORB includes support for the CORBA `Any` datatype. These simplifications significantly reduce the implementation complexity and run time footprint of F6ORB.

Unlike CORBA implementations for standard environments, F6ORB does not interact directly with the network to route messages to remote ORBs, and is not aware of the actual external address of itself or any of the remote ORBs that it may be interacting with. This is because all network connections are established via Secure Transport endpoints and flows as outlined in Section 2. Though seemingly rigid, this approach supports changing the connections between two ORBs without relying on the user space untrusted code (outside F6OS and platform actors), while ensuring that the connection satisfies MLS. However, as a result, the normal methods of creating object references are insufficient to provide location transparency to remote objects, and ORBs are incapable of creating their own references that are usable by a remote host. An object reference created by an ORB is usable only to route a request to a particular object once the request has arrived at its destination ORB. It is the responsibility of the Deployment Manager and Dictionary Manager, described in Section 5, to manage the creation of the Secure Transport endpoints and to rewrite the object reference when establishing connections.

F6ORB Thread Management—F6ORB does not include any of the complex dynamic thread management capabilities included in CORBA or Real-Time CORBA (RTCORBA). F6ORB, in nominal operation, is expected to have two or more threads that operate in the following roles:

- *Scheduler:* This thread operates as the main ORB event loop. It is responsible for listening for incoming requests, decoding any received messages, and dispatching a worker thread to handle the request. While a worker thread is actively servicing a request, the scheduler is responsible for monitoring the worker for deadline violations.
- *Worker:* These threads are responsible for processing requests in the application. When the scheduler provides an incoming request, the worker is released and provided to the actor code to service the request.

F6ORB Communication— F6ORB uses the General Inter-ORB Protocol (GIOP) version 1.4. This latest version of the protocol supports fragmented messages, which are necessary in order to support unreliable datagram oriented protocols (*e.g.*, UDP), which may be used by F6OS.

Given the maximum limit on the messages sent over an endpoint and received over an endpoint (as discussed in Section 2), F6ORB provides support for fragmenting requests and responses based on the endpoint size, and appending sequence numbers to them such that F6ORB on the recipient side can re-assemble the messages. The re-assembly timeout can be changed. After the timeout, for unreliable transmissions, the messages can be dropped on the recipient side if a fragment is missing. For reliable transmissions, the recipient ORB can request a retransmission of a missing fragment.

The ORB provides appropriate support for the OSI Layer 4 protocol supported by F6OS. If the underlying protocol is unreliable, F6ORB provides support for fragmenting requests that exceed the maximum transmission unit (MTU) supported by the protocol, and for implementing an acknowledgement protocol to ensure reliable transmission of fragmented requests over an unreliable network.

Anonymous Publish/Subscribe

Anonymous publish/subscribe is an important abstraction for building distributed systems, and it allows construction of application architectures that are more loosely coupled than is possible with remote method invocation technologies such as CORBA. F6ORB leverages a subset of features from the DDS standard from the OMG. DDS has seen wide adoption in many mission- and safety-critical applications in both military and commercial sectors.

DDS provides three important features to F6MDA:

- *Loose coupling*. Publishers and subscribers are matched through the use of *topics*. A *topic* includes a datatype, a unique name, Quality of Service (QoS) policies, and other attributes. Publishers and subscribers indicate interest in a topic, and the middleware handles the mechanics of matching these participants.
- *Quality of Service*. A rich set of QoS policies describe the requirements and behavior of the system, and allows the middleware to automatically enforce properties such as reliability, sample lifespan, data rates, etc.
- *Low overhead*. DDS minimizes the performance and bandwidth overhead necessary to implement the above advanced features.

The full DDS standard is, however, unsuitable for use in F6MDA. F6ORB provides only the Data Centric Publish/Subscribe (DCPS) layer of the DDS specification, which provides the basic support for efficient dissemination of data from publisher to subscriber. The main reasons for the unsuitability of full DDS are:

- The dynamic discovery mechanism through which publishers and subscribers are matched may incur unreasonable bandwidth overhead, especially over inter-module links. Moreover, the discovery mechanism is not MLS-aware.
- Many of the QoS policies included in the full specification are impractical to implement in the context of F6MDA due to limited resources, constrained bandwidth, unpredictable latency, and unreliable connections.
- The DDS built-in topics that are used to communicate system status to participants may establish unpredictable covert channels that communicate information to lower-labeled actors about the presence and activity of higher-labeled actors.

The DDS implementation in F6ORB is realized using interfaces found in the DDS for Lightweight CCM [12] [13] (DDS4CCM) specification. DDS4CCM provides a set of pseudo objects that provide a substantially simplified programming API to the application developer, hiding much of the complexity of using the standard DCPS interface directly. Moreover, the DDS4CCM objects may be integrated with the component model to provide formal anonymous publish/subscribe ports in component interfaces. The DDS4CCM pseudo objects are implemented and hosted inside the F6ORB.

DDS Discovery—Discovery is the process by which publishers and subscribers interested in a particular topic are matched so that data flows between interested parties. The DDS specification includes a decentralized discovery protocol that allows for dynamic matching at run time. The protocol, however, is inappropriate for use in F6MDA for the following reasons:

[2]Interface Definition Language, a language to describe datatypes independently from specific programming languages. Mappings between IDL and mundane programming languages exist.

• A spontaneous and decentralized discovery mechanism requires that each Domain Participant be able to talk to others on the network, which may violate MLS.
• The frequent message traffic required for dynamic discovery consumes precious network bandwidth, especially over the inter-module links.
• F6MDA is not fully dynamic, as application deployments are carefully planned by system integrators, so the full set of participants in a topic are easily calculated.
• In the event of system faults, fault recovery should be at the discretion of the Fault Manager, as it has a more complete view of the system than the DDS middleware.

To overcome these obstacles, F6MDA uses a quasi-static discovery mechanism implemented by the Deployment Manager and the Dictionary Manager. As discussed earlier, each participant in a DDS topic has only a single endpoint that allows it to communicate with its peers, and participants will be connected to others by manipulating the flows in F6OS on both the publishing and subscribing sides.

4. THE COMPONENT MODEL: F6COM

F6COM defines the basic software units that can be used to assemble applications. An application can be distributed across several actors, and each actor has one or more *components*. Figure 4 shows the anatomy of an application containing two actors created using F6COM components. The first actor hosts two components: Filter and Archive, the second actor contains a single component: Sensor. The components interact with each other via the generated 'glue' code that connects them to the middleware layer. Interactions within one actor involve only the middleware, while interactions across actors involve the F6OS Secure Transport. The middleware realizes its higher-level APIs in terms of F6OS system calls.

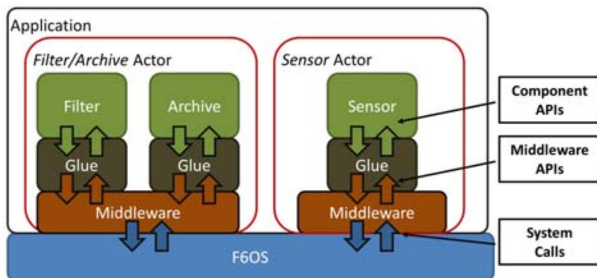

Figure 4. Anatomy of an application

Figure 5 provides an overview of a component. A component can have four different kinds of ports: *publisher*, *subscriber*, *facet* (or *provided interface*), and *receptacle* (or *required interface*). Note that the publisher/subscriber ports in F6COM differ from the event ports in the CORBA Component Model (CCM) [14]. A publisher port is a point of data emission; a subscriber port is a point of data reception. All data published or subscribed is strongly typed and is described using a topic (see Section 3). A facet is attached to the implementation of the methods defined in the provided interface and it services the requests issued through a receptacle on another component for these interface methods. Through these ports, two basic kinds of interactions can be realized: (a) asynchronous (or non-blocking) publish/subscribe, and (b) synchronous call/return. Facets and receptacles can also interact asynchronously, using callbacks. Details of these interactions are presented later in this section.

A component may also have a number of associated *triggers*. Triggers define the situations in which the various component operations associated with each port may be activated. The triggers may be invoked automatically and periodically, with a specified rate, or they can be generated by the arrival of a request detected in the middleware that hosts the component.

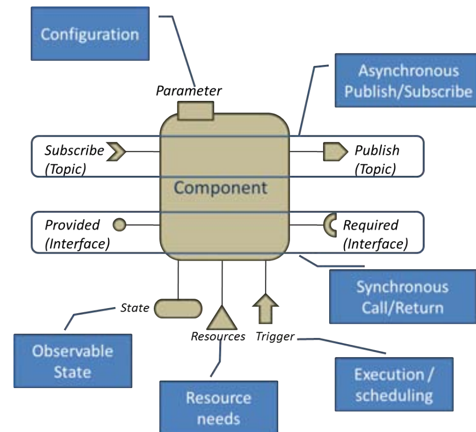

Figure 5. An F6COM component (this figure does not show the asynchronous message invocation interaction)

A component is single-threaded; at most one thread can be active in a component at any time. All component operations are serially scheduled based on the associated triggers. Unlike CCM components, all operations in an F6COM component must finish their unit of work within a specified deadline. This deadline can be qualified as *hard* (strict) or *soft* (relatively lenient). A hard deadline violation is an error that requires intervention from the underlying middleware. A soft deadline violation results in a warning. The intervention and mitigation for deadline violations is described in Section 6.

The following subsections provide the details of component internals. The 'operations' mentioned in these subsections are the stubs generated by the development tools (see Section 8) and are based on the component's specification. The signature of the generated operation depends upon the signature of the specified port.

Component Configuration Meta-Data

The following configuration meta-data is always required for components: (1) memory size, *i.e.*, the estimated total memory size needed by the component throughout its life cycle (as specified by the author of the component), and (2) a set of security labels associated with each of the component ports (specified by the system integrator).

Component Attributes

Component attributes are configuration values set by the Deployment Manager during initialization. They remain constant during the life cycle of that component. For each parameter, the generated code includes a getter and a setter. The setter of the parameter can be called only when the actor owning the component is not in the active state.

Threads

Each component has only one thread that is managed by F6ORB. Since the component has only one thread, component developers are not required to and should not use the locking APIs provided by F6OS. This prevents the developer from creating code that may result in a deadlock or race

condition.

The component thread is configured with an initial priority by the Deployment Manager. This value may be changed dynamically during run-time by the component up to a limit determined by the value set by the Deployment Manager (and determined by the system integrator) when the actor was created.

State Variables

State variables represent the internal state of the component. By default, the component exposes no interface to allow external entities to modify the component state directly. Any such modification must happen through standard port interactions. If the state variable is marked as observable, the IDL compiler will generate operations necessary to query the component's state.

State variables may also be configured with a history parameter. The value of the history parameter specifies the number of past samples of this state variable that are expected to be archived by the middleware. This attribute has a default value of 0, *i.e.*, only the current value is preserved. For variables with non-zero history value, the getter method has an attribute which specifies which past sample is requested or specifies that all past samples are requested.

Each state variable can be assigned an invariant condition based on the value of the *generate_invariant_hook* attribute, which is a Boolean function that, when specified, will be called by the middleware after the execution of the setter. The implementation of the invariant function can be provided by the component developer or auto-generated by the model-based development tools.

The rationale for the above is as follows. It is common to store and process historical values in software components implementing mathematical algorithms that relate to physical phenomena. The invariant provides a way to detect anomalous conditions that violate the safety assumptions.

Local Interfaces

Components can have local interfaces, which are collections of methods that are accessible only from components collocated in the same actor.

Ports

All ports in F6COM are configured with one or more security labels, depending on the port type. Facets and receptacles may be configured with one or more labels, while publishers and subscribers may be configured with only a single label. These labels are configured by the Deployment Manager during component deployment, based on meta-data supplied by the system integrator. The set of labels associated with the component is the union of the labels associated with all the ports on that component. The set of labels associated with an actor is the union of the labels associated with all the components of the actor. Port labels are for use by the component business logic (which can query them), especially in the case of multi-label actors. The Deployment Manager also configures the component's associated Secure Transport endpoints with the appropriate security labels. These markings on the endpoints cannot be changed by the component business logic and are used by F6OS.

Publishers and Subscribers—A publisher port is a source of information associated with an instance of a topic to which other components will subscribe. Publishers are decoupled from subscribers, *i.e.*, there is no blocking interaction between the two. The messages exchanged between publishers and subscribers are strongly typed and are instances of a topic (see Section 3).

The following attributes can be set on both the publisher and subscriber ports:

- *History Depth*: Controls the number of previously received samples cached by the middleware.
- *Reliability*: If this option is set, the middleware that manages the component will attempt to resend a multicast message to subscribers that did not receive the message.
- *Latency Budget*: Indicates the maximum life of the message, resulting in an error if it is violated.
- *Destructive Read*: Specifies whether the read operation associated with the subscriber will still give the previously read data on subsequent reads.
- *Max Samples*: Specifies the maximum number of messages that can be read at once.

Facets and Receptacles—A facet is a collection of methods that are implemented by a component and provided to other components. A receptacle is the mirror image of a facet, *i.e.*, an interface by which a component that requires some operations can connect to a facet that provides them. Facets and receptacles support the synchronous and asynchronous point-to-point communication provided by F6ORB described in Section 3.

Contracts

If specified by the component developer, pre-condition and post-condition hooks can be inlined into the call path of an operation, including publish and subscribe operations as well as method invocations:

- *Pre-condition*: This is a function with a Boolean return value that is supplied by the component developer. It specifies a condition that must be satisfied before the operation is performed. This is typically used to build software anomaly detectors that can evaluate guard conditions over the current and historical values of parameters in the operation, as well as current and historical values of the state variables of the components.
- *Post-condition*: This takes the same form as a pre-condition but is checked after the operation finishes.

Note that the total time taken for the operation will be the time needed to complete the operation's business logic plus the time taken to evaluate the pre-condition and post-condition.

Triggers

Triggers define the situations in which the underlying middleware managing the component needs to invoke some component operation. Each trigger is associated with one and only one component operation. All activities have a finite deadline by which they must finish their task. Failure to do so will result in some action being taken by the fault management in the middleware based on the nature of the deadline (*i.e.*, hard or soft).

A trigger can be either time-based (which can be used to schedule an activity at the specified rate), or event-based. There are three kinds of trigger events:

- *ON_DATA_AVAILABLE*: This trigger is used to start an operation associated with a subscriber.
- *ON_CALL_AVAILABLE*: This trigger is used to start an operation associated with a facet. The operation (*i.e.*, the

method) launched will depend on the incoming request.
• *ON_CALLBACK_AVAILABLE*: This trigger is used to start a callback method associated with an asynchronous method invocation (AMI).

Trigger and Operation Interaction—Figure 6 shows the interaction between a time-based trigger, the middleware, and the component. The middleware launches the operations associated with the trigger when the trigger becomes active. A time-based trigger will generate timeout events at a specified rate. If the operation is not completed within the time deadline specified, an anomaly is detected and reported to the Fault Manager. Violations of hard deadlines also result in the termination of the operation by the underlying middleware.

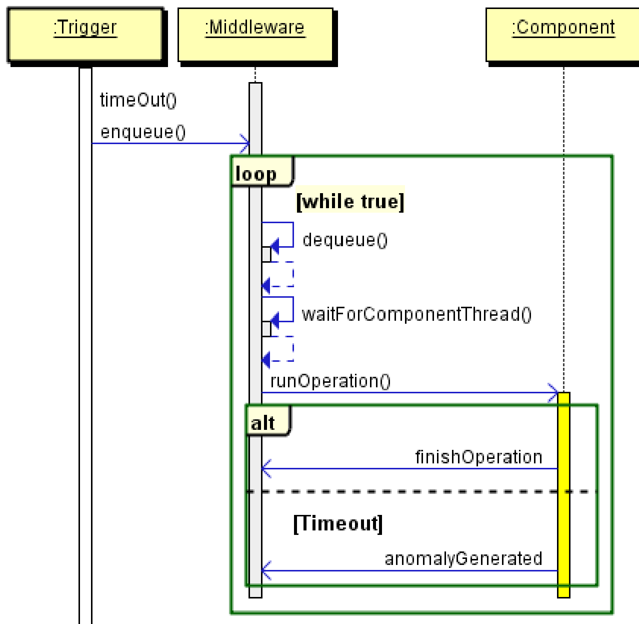

Figure 6. Interaction of a time trigger and component

Triggers may be used to realize the following interaction patterns:

• *Periodic Publisher (Time-trigger with Publisher)*: This pattern relies on an operation in a component that publishes data at a specified rate.
• *Periodic Subscriber (Time-trigger with Subscriber)*: This pattern can be used to implement a 'pull' subscriber operation that is triggered periodically. Based on port properties, the subscriber will consume up to the maximum number of samples specified. A periodic subscriber should not block when reading the message; the middleware ensures that this does not happen. Note that the automatically scheduled periodic subscriber can oversample (at a higher rate) or undersample (at a lower rate) the data produced by a publisher.
• *Aperiodic Subscriber (Event-trigger with Subscriber)*: This pattern will allow the implementation of a 'push' subscriber operation, which gets triggered only when the data is available.
• *Periodic Receptacle (Time-trigger with Receptacle)*: Similarly, a time-triggered receptacle can be specified. This enables a component to periodically refresh itself with new data from another component.

Operation to Operation Interaction— Inside a component, once a trigger has started an operation, it is possible to daisy-chain the calls by directly invoking other operations as functions. However, care should be taken because the deadline of the first operation must account for the longest chain of other operations that can be called as part of the first operation.

Component Life Cycle

Component attributes and states are set by the actor's deployment component, the *Actor Home*, which executes the necessary operations in the actor on behalf of the Deployment Manager. A component can be in one of four states during its life-cycle. State changes are initiated by the Actor Home. The component states are:

• *Initial*: This is how the component starts after being instantiated. In this state the Actor Home can configure the component parameters. Component parameters cannot be altered in any other state.
• *Activated*: This is the typical state of a component when it is fully operational *i.e.*, it can perform all its operations.
• *Passivated*: The component can execute only state variable setter operations, and cannot serve any other request. This state is used to implement passive replicas [15].
• *Semi-activated*: The component can execute state variable setter operations and subscriber operations, and can execute facet operations with only *in* arguments and receptacle operations with only *out* arguments. This state is used to implement active replicas. However, it ensures that the replica cannot provide service to any other component. Only primary or the 'activated' component can provide service.

Component Interactions

While each component and its associated ports, state variables, and triggers can be individually configured, an assembly of components (*i.e.*, an actor or application) is not complete until the interactions between all ports are configured. The association between the ports depends on their type (synchronous or asynchronous) and the datatype/interface type associated with the port. Two kinds of interactions, asynchronous and synchronous, are possible between components.

Asynchronous Interaction

Publisher/Subscriber Interaction— This interaction occurs when a publisher port of a component is logically associated to one or more subscriber ports of another component. This logical association between publishers and subscribers is specified at development and integration time, and is based on the topic of the publisher and subscriber. A topic includes not only a data type, but also a security label. At runtime, a subscriber is associated with a publisher by (i) the Deployment Manager, which checks that the subscriber has the topic of the publisher and creates F6OS flows to associate the subscriber's endpoint with the multicast address, and (ii) the Communication Resource Manager, which adds the subscriber's node to a multicast communication group. For a publisher and subscriber to be associated with each other, the topic of the publisher must be the same as the topic of the subscriber, and the reliability settings on the publisher and subscribers must match. Because each topic has a unique label, a topic match implies that the publisher and subscriber can be associated from a security point of view.

Since the publisher has a single label, the message sent by the publisher operation is automatically marked with that label by F6ORB. This label is used to send the message to the underlying F6OS endpoint. Inside F6OS, the Secure Transport checks that the current message security label

matches the multicast group. The message is forwarded to the multicast group only after the security checks. Any security violation is logged. This is the key enforcement mechanism for secure information flows in the architecture. Note that when a node receives the message, the Secure Transport on the recipient side double-checks that the message label matches the security label on the subscriber endpoint.

Based on the associated triggers, one can classify publishers and subscribers into two categories: (a) periodic (time-based trigger), or (b) aperiodic (event-based trigger). If a subscriber is associated with a periodic trigger, it exhibits sampling behavior. Even if the rate of the publisher is indeterminate (for example if the publisher is aperiodic), setting the period of the subscriber ensures that the events from the publisher are sampled at a specific rate. When the interacting publisher and subscriber are both periodic, the value of the subscriber's period relative to the publisher's determines if the subscriber is over-sampling or under-sampling.

Interaction between a periodic publisher and an aperiodic subscriber indicates a pattern where the subscriber is reactive in nature. In such a case, the subscriber stores incoming published events in a queue, which is processed in a FIFO manner. If the queue size (history of subscriber and size of the buffer in the read endpoint) is configured appropriately, the subscriber can operate on all the events received.

The case of interaction between an aperiodic publisher and an aperiodic subscriber is similar to the one between a periodic publisher and an aperiodic subscriber.

Asynchronous Method Invocation (AMI)— In order to use AMI for a receptacle, the component developer must mark the interface as 'two-way AMI'. Based on this attribute, the IDL compiler generates a callback operation for every method in the receptacle. A callback method can be associated only with an event-based trigger. Thus the callback method is triggered when the reply from the facet arrives back at the middleware of the requesting component.

AMI has the constraint that the receptacle can be used for only one call at a time. The receptacle cannot be used to send another request unless the reply has arrived back (via a callback or a timeout specified on the receptacle). If a method in the receptacle is invoked before this happens, the call will return with an error that must be handled by the calling operation.

Synchronous Interaction

This interaction follows a request/response semantics. A receptacle port can be associated with a facet port of an identical interface type. A facet can be associated with one or more receptacles. A receptacle can be associated with one or more facets. However, the component business logic on the receptacle side must indicate the facet to which the call must be sent. The connection between facets and receptacles is implemented by the Deployment Manager, which creates the required endpoints and flows to connect facets and receptacles.

At run-time, a multi-label port sets the label to use in subsequent calls using the component context. This label should be chosen from among the port's labels, but it is always then checked by the Secure Transport in the F6OS to decide whether the message should be sent to the facet. If the message cannot be sent, a *NOT_AVAILABLE* error code is returned to the middleware, which forwards it to the component. Because of the synchronous nature of these interactions, the deadline of the required interface method (*i.e.*, the caller) must be greater than the deadline value for the provided interface method (*i.e.*, the callee).

Synchronous facets in this model are always event-triggered. The interaction patterns for synchronous ports are borrowed from CCM. The key difference is deadline monitoring and security label monitoring via F6OS. The default type of interaction is request/response or two-way communication, *i.e.*, a receptacle port blocks until the facet finishes the operation and returns the results.

One-Way Call—A receptacle method can be marked as 'one-way'. A one-way call returns from the receptacle as soon as the message is handed over to the Secure Transport (and, if the facet is on another node, handed over to the network queue). A one-way call cannot have *out* arguments.

Relationship with CORBA Component Model (CCM)

F6COM shares a number of features with the CCM [14]. However, there are key differences in functionality. These differences are (a) single-threaded operation, (b) specification of time bounds on each operation associated with the component, (c) replacement of the standard CCM event subsystem with anonymous publish/subscribe interactions, (d) specification of triggers, time- or event-based, that determine when the component operations are scheduled, (e) fixed resource allocation, and (f) security labels on ports. Since F6COM components are expected to operate on resource-constrained systems, their resource needs are declared at development time, verified at system integration time, and enforced at run time. Ports in F6COM have security labels, assigned by the system integrator, which play a role in supporting MLS.

5. Platform Actors

Table 2 lists the platform actors that complement F6OS (see Figure 2). Recall that platform actors run in the system partition and are not subject to temporal partitioning.

Operations Manager	Manages Spacecraft System Operations
Deployment Manager	Deploys Software Applications
Dictionary Manager	Maintains Dictionary of Objects
Certificate Manager	Verifies Public Key Certificates
Communication Resource Manager	Manages Communication Resources
Fault Manager	Provides Fault Management Services

Table 2. F6MDA platform actors

This section describes four platform actors: the Operations Manager, the Deployment Manager, the Dictionary Manager, and the Certificate Manager. The Communications Resource Manager, responsible for managing networking resources on a node, is described in Section 2. The Fault Manager, responsible for system wide fault mitigation, is described in Section 6.

Operations Manager

The Operations Manager (OM) is responsible for the overall management of operations on a node, a module, or the cluster. It works with the application that runs the flight control

software, the Fault Manager, and the Deployment Manager. There is one OM per node, called the *node OM*. In each module, the node OMs elect one node OM to be the leader of that module, called the *module OM*. In the cluster, the module OMs elect one module OM to be the leader of the cluster, called the *cluster OM*. All OMs periodically emit a heartbeat message, which is used to assess their liveliness. If an OM dies, F6OS will restart it automatically.

The periodic heartbeats, leader election, and automatic restarts are intended to support fault tolerance of the OMs and to establish a clear command and control hierarchy. If an OM terminates, its own F6OS will attempt to restart it. If it fails to restart (because the node is dead, for instance) the failure will eventually be detected by the other OMs. If the failed OM played a leader role (as a module OM or a cluster OM), it is replaced by a newly elected OM. The status of an OM as either a node, module, or cluster OM dictates its behaviors and responsibilities. A node OM is responsible for managing the operations on only that node. A module OM has a wider array of responsibilities, including:

- Maintenance and periodic refresh of a module-local cluster membership table.
- Receiving communications from OMs of modules that wish to join the cluster.
- Management of the relative security authentication, resource allocation, and potential (re-)deployment of applications.

In addition to the functions of a module OM, a cluster OM has the following responsibilities:

- Coordination with module OMs about cluster-wide autonomous flight operations like scatter/gather maneuvers, formation flight, and orbital changes.
- Receiving communications from the ground about imminent scatter maneuvers.
- Receiving communications from the ground to deploy new software configurations.

Deployment Manager

The Deployment Manager (DeM) is responsible for deploying and configuring the components, actors, partitions, and Secure Transport endpoints and flows on a node. The DeM can also modify the partition scheduling table of the node.

Analogously to the Operations Manager, there are three roles for DeMs:

- *Node DeM*, responsible for coordinating deployments only on the local node.
- *Module DeM*, responsible for coordinating application deployments across nodes in a module. The DeM collocated with the module Operations Manager is given this role.
- *Cluster DeM*, responsible for coordinating application deployments across the cluster. The DeM collocated with the cluster Operations Manager is given this role.

On each node, the DeM uses the functionality provided by the Actor Home (described in Section 4), which runs on the main thread of an actor and is created when the actor is first spawned. The Actor Home acts as a component server for the components that make up an actor.

The interface of the DeM and Actor Home, as well as the structure of the meta-data used to encode information about application deployment and the hardware environment, is derived from the OMG Deployment and Configuration for Component-based Application specification [16] [17].

Dictionary Manager

The Dictionary Manager (DiM) maintains a directory of references and publish/subscribe information that is used at run-time by the platform actors to locate platform actors on other nodes and by the Deployment Manager to resolve connection endpoints and publish/subscribe groups.

DiMs across nodes in the cluster remain synchronized in order to provide the most up-to-date information to their local Deployment Managers, and to ensure that changes made by the Fault Manager to mitigate faults propagate to other nodes and modules in a timely manner. The architecture does not mandate a particular synchronization strategy or algorithm, as this may change depending on a number of factors, including available inter-module network bandwidth/latency or the status of a DiM as the cluster DiM (by virtue of its collocation with the cluster Operations Manager).

The DiM performs three functions: (1) it stores addressing information for the platform actors for each node; (2) it stores information necessary to establish connections between components; and (3) it facilitates the quasi-static discovery mechanism used by the DDS portion of F6ORB. These three functions are described below.

Node-Level Platform Actors—The DiM stores information necessary to locate all the platform actors on a remote node. This information is stored in the DiM as a set of records, one for each node, which contains a reference for each of the node Deployment Manager, Operations Manager, Fault Manager, and Communication Resource Manager. Additionally, this record provides the network address of the node. These records are indexed by the unique node name assigned by the cluster owner (*i.e.*, system integrator).

Component References—The DiM stores information necessary for the Deployment Manager to resolve connection references during deployment of application actors. These references are maintained at the granularity of components inside those actors. The dictionary information is indexed using a unique component name that is comprised of the unique identifier of the deployment plan that was used to deploy the component, the actor name that hosts the component, and the name of the component as specified at deployment. The component data record consists of a list of facet and receptacle ports that are provided by the component. Additionally, the component record contains a list of replicas that have been deployed through either explicit action by the system integrator, or through autonomous fault mitigation strategies planned by the Fault Manager.

Publish/Subscribe Communication Groups—The DiM is responsible for facilitating the quasi-static discovery mechanism used by the anonymous publish/subscribe middleware described in Section 3. This is accomplished by maintaining a dictionary of all topics present in the cluster.

Data associated with a topic includes a domain ID (represented by an integer), a topic name (represented by a string), a datatype (represented by the CORBA *RepositoryID* of the datatype), the multicast group name, QoS policies, and the associated security label.

Certificate Manager

Recall that each node has a public key used to establish cryptographic protection for messages (and their labels) transmitted over the network by the Secure Transport. Each node

has a signed certificate that binds the node's public key to the node: the certificate is distributed to and verified by all the other nodes that communicate with that node. Certificate verification is carried out by the Certificate Manager (CM), using standard techniques (*e.g.*, as used by web browsers to verify certificates of web sites). Since the techniques are standard and described in detail elsewhere, here we only provide highlights and F6MDA-specific information.

The CM contains one or more root certificates, which the CM regards as containing public keys of Certification Authorities that are trusted to sign other node's certificates (root certificates are often self-signed, because they are trusted but not verified). Root certificates may vary from one node to the other. The root certificates of a node's CM are set by the Deployment Manager of the same node, as part of the node's configuration.

The CM verifies node certificates upon request of the Deployment Manager, as part of a deployment plan. If the deployment plan includes a flow between an endpoint on node N_1 and an endpoint on a different node N_2, the Deployment Manager of N_1 asks the CM of N_1 to verify N_2's certificate, and the Deployment Manager of N_2 asks the CM of N_2 to verify N_1's certificate. Each CM checks the signature of the node's certificate with the public key of the Certification Authority (contained in the associated root certificate) indicated by the node's certificate.

The CM also contains a Certificate Revocation List (CRL), *i.e.*, a list of certificates of public keys whose corresponding private keys have been compromised and therefore are no longer valid. As for root certificates, the CRL of the CM of a node is updated by the Deployment Manager of the same node, as part of the node's configuration.

6. FAULT MANAGEMENT

Providing uninterrupted services for performance-sensitive distributed applications operating in resource-constrained environments is hard. It is even harder when the operating environment is dynamic, where processor or process failures, system workload changes, and fluctuating network connectivity are common. An information architecture based on fault-tolerant middleware for these applications must assure high service availability and satisfactory response times for clients. F6MDA is representative of such systems, calling for a range of solutions that provide fault tolerance to the entire fractionated spacecraft. The F6MDA fault management architecture supports both high availability and application performance while being aware of the resource constraints.

The primary vectors of the F6MDA fault tolerance design include the following capabilities:

- Customizable and layered fault detection and isolation strategies, which detect anomalies in different layers and diagnose root causes to be treated by the fault mitigation logic.
- Customizable and layered fault mitigation strategies, which serve to mitigate faults as closely as possible to the source of the faults, by restarting and reconfiguring failed entities.
- Semi-customizable fault recovery strategies, which use redundancy-based mechanisms based on primary-backup (passive replication) mechanisms to provide autonomous run-time recovery from faults, while attempting to maintain acceptable response times to applications and being aware of the scarcity of resources.
- A model-driven fault management framework, which F6MDA application developers can use to define fault management policies for their applications, which in turn are followed by the run-time fault management architecture.

System Assumptions and Fault Model

Before describing our fault tolerance solution, it is important to understand the underlying system assumptions and the fault model.

System Assumptions—F6MDA assumes that a predominant number of applications in the fractionated spacecraft missions require soft real-time assurances, where some tolerance to variation in response times is acceptable. Both periodic and aperiodic applications are assumed to be present. Applications are realized as a collection of one or more connected actors. Actors in turn can include a collection of one or more collocated components and are assumed to execute in partitions on nodes within modules.

The system is highly resource-constrained, there being limits on the available computing, storage and networking resources, in addition to constraints on other physical resources such as cameras, radios, and GPS. These resource constraints prohibit any sophisticated, resource-intensive, transactional solutions for fault tolerance. Instead, simpler yet robust solutions to fault tolerance are desirable. The fractionated spacecraft is an inherently distributed system of satellites and ground stations. Because of changing orbital conditions, network connectivity is highly variable, with available bandwidth varying over time, and at times no connectivity at all.

In terms of workloads, the system is not a statically defined closed system where all the possible applications and their workloads are predefined. On the contrary, it is assumed that new applications can be introduced into the cluster via ground stations, while existing applications can be decommissioned for a variety of reasons, including mission changes. Despite the open nature of the system, the "openness" is bounded in that before introducing a new application (or decommissioning an existing application), the system integrator has sufficient information to perform a system-wide resource management function. By system-wide, we refer to decisions that must be made at the level of the overall system, which could include a node, module or cluster.

All physical devices in the system, including sensors, communication devices, GPS receivers, spacecraft sensors and actuators, etc., are assumed to be encapsulated by appropriate service actors that are able to indicate anomalies with their devices. These actors provide access control and resource management functions for the devices, in addition to anomaly reporting and fault mitigation functions.

Finally, a Trusted Computing Base is assumed that hosts the critical fault management capability.

Fault Model—Two types of faults are handled in F6MDA: physical and software-level. Each of these two categories may experience both permanent and transient faults. Examples of permanent physical faults include: a node malfunction, a module malfunction, or a cluster-wide malfunction, all of which are fail-stop (*i.e.*, the failed entity does not respond to any stimuli, and it is possible for external entities to observe their failure); a malfunction of other physical entities, such as sensors, GPS, radio, etc., becoming inoperative; network links permanently down; and "blabbering idiot" (*i.e.*, an entity producing useless and arbitrary information,

interfering with normal operations). Examples of transient physical faults include: temporary resource exhaustion (*e.g.*, storage full) that may cause delays in responses, or temporary network disconnection (*e.g.*, due to signal fading or obstacles).

Examples of permanent software-level faults include: OS exceptions such as segmentation violation; security violations; response times that exceed limits and deadline violations for critical tasks; unintended actor (process) termination; and invariant, pre- and post-condition violations reported by a component that cause it to stop. Temporary software-level faults include temporary exhaustion of locks and timeouts on operations.

Fault Management System

The F6MDA fault management architecture includes layered anomaly detection, diagnostics, mitigation strategies, and redundancy-based fault recovery (needed when mitigation strategies are insufficient). The design allows application developers to override the default approach by specifying policies using the model-driven framework, manually providing the deployment meta-data for fault management, or by using out-of-band solutions.

Figure 7 depicts the overall fault management system design and its interaction with other F6MDA entities. The overall functionality is realized through collection and coordination of individual, layered fault management functions. F6MDA requires system-wide fault management capabilities for a node to reside and execute in the system partition (where platform actors execute). This task is handled by a system-wide Fault Manager (FM). Component-level fault management is realized within F6ORB, while actor-level fault management is handled by a special function running as an actor in the user partition (*i.e.*, the temporal partition in which an application actor executes) along with application actors. Fault management of platform actors is realized within F6OS, which is where part of the fault management functionality also resides, as shown in Figure 7.

The FM is responsible for system-wide fault diagnostics and fault management function. This platform actor exists on every node, and interacts with F6OS and other elements of the fault management architecture, but also with other platform actors, as well as other FMs residing on other nodes and modules, as discussed below. Note also the distinction between the fault management system function (which is distributed across multiple entities and handles localized fault management tasks) and the FM (which is one platform actor on a node dedicated to system-wide fault management tasks). This layering of the fault management functions and the need to provide security imposes a certain hierarchy and flow of messages as shown in Figure 8.

The rationale for this scheme is as follows. Distributing the functionality of the fault management system is necessary because F6MDA relies on temporal and spatial partitioning, which means that the fault management functionality should also remain appropriately isolated. Moreover, maintaining all the functionality in one place (*i.e.*, within a single, monolithic FM inside the system partition), is not appropriate since it would lead to significant inter-partition messaging and cause delays for any corrective action that could be taken at the source of the fault. Finally, security concerns require that appropriate separation is maintained. Consequently, different parts of the fault management system must exist in different places and play different roles. In F6MDA, both mitigation

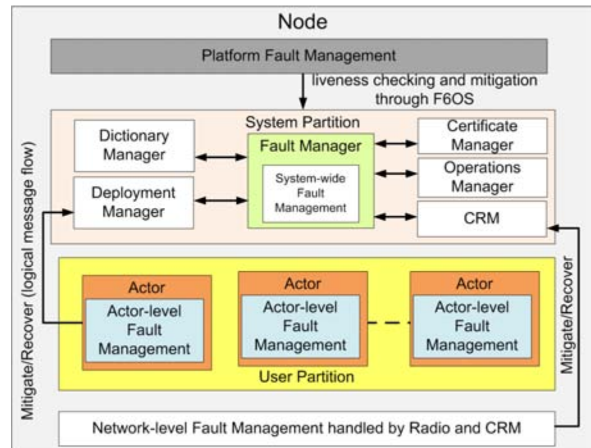

Figure 8. Interactions

and recovery from faults require coordination of the various fault management functionalities with the other F6MDA platform actors.

Fault Detection and Isolation— At each layer of the fault management architecture, anomalies are detected, possibly diagnosed, and mitigation actions are taken. The detection mechanisms are layer-specific: anomalies with the communication device can be detected by the wireless networking hardware, the device driver, or the Communication Resource Manager; anomalies with the computing hardware can be detected by F6OS; anomalies with actors can be detected by F6OS; anomalies in software components can be detected by F6ORB; and anomalies with nodes or modules can be detected indirectly, through platform actors (*e.g.*, the Operations Manager) that may receive reports from the spacecraft fault protection system. The actual detection can take many forms: hardware signals (for the network hardware) or lack of response from the hardware (for the device driver); the detection of an actor's unexpected termination (for F6OS); the violation of pre- and post-conditions, or deadline violations on components (for F6ORB).

Fault detection is followed by fault diagnostics (also called fault isolation) that analyzes anomalies and hypothesizes the causes of faults. In many situations, symptoms (anomalies) can be directly mapped to faults, so the diagnosis is trivial. However, when faults result in complex sequences of anomalies, the diagnosis becomes more difficult and a full diagnostics reasoning capability is needed. A diagnostics reasoner collects multiple anomaly reports, correlates them, and generates hypotheses that best explain the observed anomalies. This process is called 'system-wide fault diagnostics'.

The F6MDA layered diagnoser design is flexible, making it possible to customize the diagnostics and reasoning capabilities at every layer that maps the detected anomaly to a fault hypothesis. For system-wide diagnostics, the use of a graph-based reasoning engine is suggested. This approach is based on a temporal failure propagation graph model that represents the causal relationships between various fault sources and their effects as they propagate through the system. This propagation can lead to observable anomalies or to secondary faults in other parts of the system, which can trigger subsequent anomalies. The reasoning engine within the diagnoser performs a backward analysis to relate the observed sequence

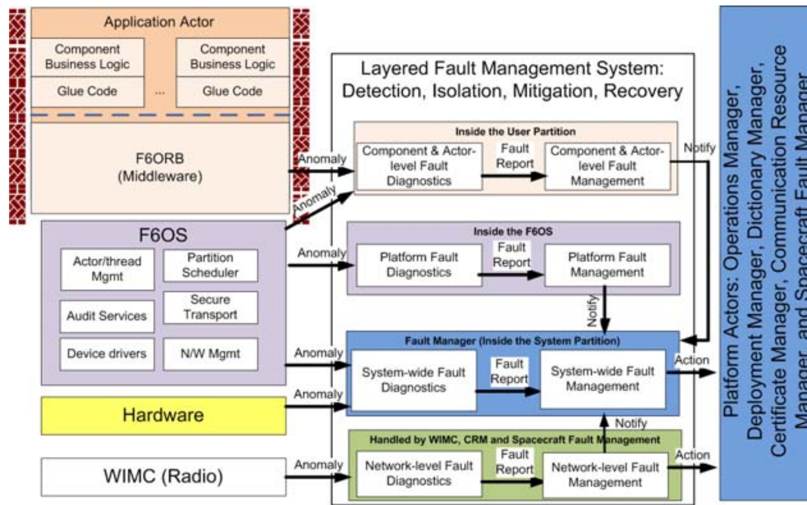

Figure 7. Layered fault management in F6MDA

of anomalies to the fault sources. Once fault sources are isolated, they are used to form ranked fault assumptions that represent plausible explanations for the anomalies.

The rationale for the above is as follows. Separating the isolation and detection from other steps of fault management (*e.g.*, mitigation and recovery) is important because it enables various diagnostics algorithms [18] to be used to produce a fault hypothesis while totally decoupling the system from the actual mitigation/recovery logic.

Mitigation-based Fault Management—The layered fault management architecture in F6MDA provides several opportunities to mitigate faults before more sophisticated recovery mechanisms are invoked. The design of the fault management architecture is such that applications may decide to override default behavior as desired. In the rest of this section, we elaborate the default mitigation strategies at each layer and the overrides that are possible.

Component-level Fault Management: This function provides mitigation and fault management at the level of fine-grained components and is handled by F6ORB. Making the component-level fault management capability reside within an actor (in the middleware) provides for a simpler implementation and provides a way to realize finer-grained, actor-specific actions to be taken after a fault.

Components include ports that can be monitored to detect anomalies internal to a component for interested entities outside of the component. The component-level fault management function receives anomaly events from the component. Anomaly reports could include violations of pre- and post-conditions, violations of component state variable invariants, deadline violations, and timeouts. When such an anomaly reaches the component-level fault management function in F6ORB, it can immediately suspend the further execution of the component until a mitigation action is taken, so that further anomalies are not introduced into the system by the malfunctioning component.

Depending on the state of the failing component, the corresponding action could be: ignore; ignore but report to the node FM (but not directly, see below); or abort the current operation in the component and report to the node FM. The aborting action can be performed internally by F6ORB. For security reasons, the architecture does not allow any user actor to directly communicate with the platform actors. Consequently, the component-level fault management function must escalate the reporting process to a special actor executing in the same partition as shown in Figure 9. This special actor performs application actor-level fault management and is a trusted entity developed and deployed in each partition by the system integrator.

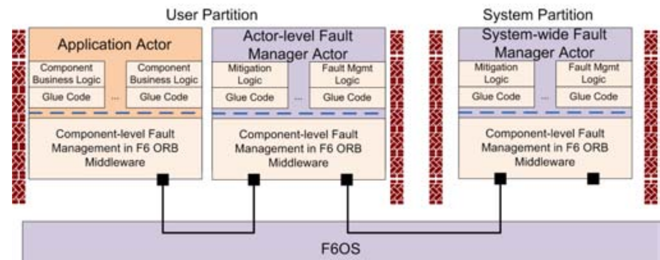

Figure 9. Fault escalation

Actor-level Fault Management: The special actor mentioned above implements the actor-level fault management function. An actor-level fault management function has two responsibilities: (1) act as an intermediary to relay reports from the component-level fault management function to the system-wide FM, and (2) be responsible for the health management of application actors in that partition. The actor-level fault management function is assumed to be a trusted actor developed and deployed by the mission integrators that executes in each user partition. Because they are trusted, they are allowed to communicate with the system-wide fault manager through the endpoints as shown in Figure 9.

Creating a per-user-partition actor-level fault management function helps in system security, where application actors are prevented from communicating with platform actors. It also helps to throttle the rate at which messages from a component-level fault management function may be generated. This capability is important when the component-level fault management function may be malicious and intends to cause a denial-of-service attack. Finally, delegating the monitoring of the status of individual application actors in a partition to this special actor helps scale the system. Other-

wise, a single system-wide FM would have to track the health status of all the actors on the node, which might consume additional resource cycles in the system partition, thereby reducing the resource cycles available to user partitions.

Network-level Fault Management: Network faults occur when the radio or its associated monitor signals a serious problem, which makes the radio unusable. If no other radio exists on the module, it would mean the module is unreachable. The presence of additional radios provides a possible alternate route to reach the module. The F6MDA network-level fault management function relies on fault management (and possibly a power cycling option) within the radio to mitigate faults within the radio. The Communication Resource Manager (CRM) constitutes the remainder of the network-level fault management. In particular, the CRM is responsible for determining alternate routes and updating the internal routing tables to reflect those changes. These mitigation strategies are feasible if alternate radios are present on the module. Otherwise, the CRM informs the system-wide FM of the failure with the radio.

Security Violations: Security violations are considered faults. F6MDA fault management requires that F6OS notify the FM of the violation at the appropriate layer. Mapping is then performed to a specific type of fault, and appropriate mitigation or recovery action is taken. For this purpose, the fault management functions at all the layers that are interested in security and other policy violations register themselves with the F6OS, using a specially provided system call. In turn, F6OS will asynchronously report violations to the fault management function. The mappings from security violation to the type of fault and the action to be taken are specified by the system integrator. The default action will quarantine the violating entity by stopping the misbehaving actor(s) through coordination with the Deployment Manager.

Managing Faults in Physical Devices: Managing faults in other physical devices of the system requires custom solutions as demanded by the missions and in concert with the module's fault management capability. Consequently, the flexible design of the FM in F6MDA allows integration of custom fault management capabilities, which will handle these additional kinds of faults.

Platform-level Fault Management: Platform actors might malfunction (*i.e.*, fail-stop). F6OS maintains a node-level fault management function that can restart one or more of the failed platform actors. Naturally, when the failed platform actor is rejuvenated, its state is made consistent using the persistent file store used by the platform actors.

System-wide Fault Management: When the layer-wise mitigation strategies do not succeed, or when the layer-wise fault management functions determine that it is necessary to inform the system-wide fault management of the mitigation action, the FM executing in the system partition gets notified and must perform specific operations.[3] Since the action of deploying and connecting components of an application, or starting/stopping actors, can be performed only by the Deployment Manager that runs in the system partition, it is important to include another trusted actor that can communicate with the Deployment Manager and other actors of the system partition. Consequently, maintaining a fault management function in the system partition is the right approach. Not only can such a fault management function deal with local mitigations, but since it is a trusted entity, it can also participate in node-level, module-level and cluster-level fault management.

Recovery-based Fault Management—Sometimes simple mitigation strategies are insufficient, or certain applications may not want to undergo an iterative and incremental process of mitigating a fault in the application. In both cases there is a need for a redundancy-based mechanism to recover from faults. Figure 10 illustrates the architecture of the F6MDA autonomous, run-time mechanism for recovery from failures, which attempts to deliver desired application response times while conserving resources.

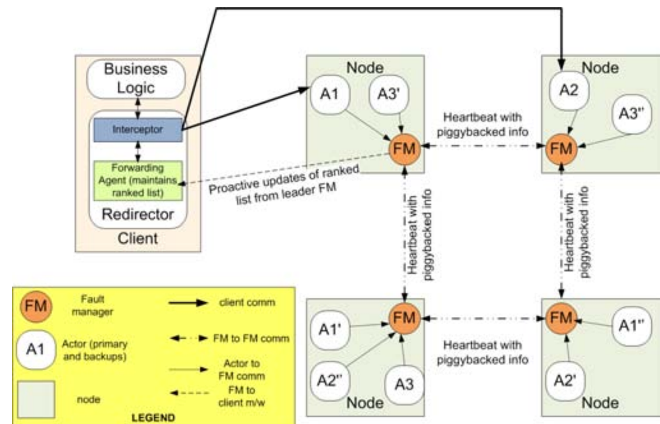

Figure 10. Redundancy-based autonomous recovery mechanisms

The primary vectors of the redundancy-based fault recovery mechanism are:

Redundancy-based Proactive Failure Recovery: F6MDA uses redundancy-based recovery using primary-backup (passive) replication [15]. The passive scheme is attractive for environments that are resource-constrained and where applications predominantly possess soft real-time requirements, *i.e.*, where average response times should be within tolerable bounds. To enable fast and transparent failover, F6MDA employs a proactive, resource-aware failover strategy [19] that attempts to maximally meet response times of applications by dynamically ordering the failover targets based on measured resource utilization.

F6MDA maintains the notion of a ranked list of failover targets on the client side middleware, which resides in the actor that plays the role of a client for an F6MDA application. The ranked list is periodically computed by the FM, which orders the replicas of the server according to the utilization of the resources consumed by the replicas. Note that a server is one or more actors of an application that together serve client requests. In the simplest case, an ordering could be from least-utilized to most-utilized resources. The motivation for this ordering is that on failure, the client will fail over to the replica that has the least load, and hence will provide the best performance in a failure case.

The FM proactively distributes the ranked list of failover targets for this application to the client side. Rather than making the client application aware of this list, the ranked list is cached on the client side middleware, inside a forwarding

[3]Due to space limitations, details of the specific actions taken are not described.

agent. The interceptor logic at the client side middleware is responsible for intercepting any exception conditions. One such exception condition is the connection failure to the primary replica. When this exception is intercepted by the middleware, it will consult the cached ranked list, and a connection to the newly chosen backup will be made by the middleware. At this stage, all new requests from the client will be made to the newly promoted backup in the ranked order list. To make this happen, the middleware on the client-side will open a new endpoint to communicate with the newly chosen backup.

Since F6MDA operates in a resource-constrained environment, the resource allocation for the server replicas uses a resource-aware allocation technique based on backup resource overbooking [20]. This strategy leverages the properties of the primary-backup scheme, wherein the fact that a backup replica does not impose the same load on a resource as the primary is exploited to pack more backup replicas of different applications within the available resources.

Scaling the Recovery Mechanism to Module- and Cluster-level: Since F6MDA must provide fault tolerance across nodes, modules and clusters, there must necessarily be coordination among all the FMs on each node. F6MDA provides a hierarchical scheme wherein among the FMs on the nodes of a module, one such FM is chosen as a *leader*. In a similar fashion, a cluster-level leader gets chosen from among all the module-level leaders' FMs. Ground stations are also seamlessly made part of this logic. However, ground FMs are never chosen as cluster leader because of the frequent loss of connection of modules with ground stations.

Recall that besides the FM, each node also includes a number of other platform actors. Any one of these entities may fail and hence each one should be made fault-tolerant. It is possible to use the same hierarchical fault tolerance scheme for each platform actor. However, leader election for each platform actor could lead to excessive messaging overhead on an already resource-constrained environment; membership protocols for every platform actor, and potentially inconsistent leader election for each actor, *i.e.*, a FM may become a leader for a module but not necessarily its Deployment manager.

To overcome these overheads and inconsistencies, F6MDA delegates the leader election responsibility to the Operations Manager. Whichever Operations Manager becomes the leader at either the module- or cluster-level will then delegate the leader responsibilities for the corresponding roles to the other platform actors in its node.

7. Security

The enforcement of MLS by F6OS on the messages exchanged among actors is a form of Mandatory Access Control (MAC), *i.e.*, it cannot be bypassed. The spatial separation among actors ensures that actors cannot communicate directly via shared memory or files, but only via messages. The temporal separation among partitions prevents the formation of covert timing channels among actors in different partitions; two actors in the same partition should have the same security labels or should be trusted/verified to not attempt to exploit covert timing channels. The enforcement of MLS and of spatial/temporal separation by F6OS depend on data internal to F6OS (*e.g.*, the labels assigned to actors and endpoints), which can be set only by privileged system calls, which themselves can be made only by platform actors, which in turn process requests only from other platform actors[4]. Ultimately, F6OS internal data is set from deployment plans constructed by the system integrator. F6OS and platform actors are part of the Trusted Computing Base (TCB), *i.e.*, they are critical to enforce security.

A single-label actor can send and receive messages only with its assigned label, and therefore cannot violate MLS. A multi-label actor can potentially violate MLS (*e.g.*, by re-labeling and forwarding a message), but only within its assigned labels (*e.g.*, an actor with Secret and Top Secret labels may re-label Top Secret data to Secret, but not to Unclassified). Thus, multi-label actors are part of the TCB: they must be trusted (i) to not violate MLS or (ii) to "legitimately" violate it. For example, a camera-wrapping actor may serve requests from actors that have different labels at different times, keeping data associated with different requests separate and thus satisfying MLS. An example of an actor that must legitimately violate MLS is a downgrader that turns a Top Secret high-resolution image into an Unclassified low-resolution version of the image. The legitimacy of these violations depends on the semantics of the data and must be verified by the system integrator.

Since shared mission-specific resources are encapsulated in actors, Discretionary Access Control (DAC) is not enforced by F6OS, but by individual actors as part of their logic. For example, a camera-wrapping actor may allow access only to some actors by responding to messages from those actors and rejecting messages from other actors. DAC, as the term implies, is at the resource owner's discretion (unlike MAC). The restriction that platform actors process requests only from other platform actors is also a form of DAC.

To ease verification and certification, F6MDA aims at minimizing the size and complexity of the TCB. F6OS is designed with a micro-kernel approach in mind; platform actors are factored out of F6OS, and mission-specific device drivers should be put into application actors to the extent possible. F6ORB includes only the needed CORBA and DDS features because the F6ORB libraries in multi-label actors are part of the TCB. F6COM focuses the analysis of a multi-label actor on its multi-label components since single-label components do not directly affect MLS. Our Isabelle/HOL formalization of multiple-domain labels (mentioned in Section 2) is the starting point of a formal approach to verify and certify F6MDA that we intend to pursue.

8. Model-driven Development

As described earlier, F6MDA provides a software platform with well-defined APIs to applications. The development process for applications is similar to that of other embedded system applications: analysis, design, implementation, verification and testing (preferably in spiral progressions). The artifacts produced by the development process are the applications and meta-data about the applications. The applications are integrated, subjected to verification and validation (V&V), and deployed on the actual platform by a *system integrator*, who is normally the cluster owner or its representative. This is illustrated in Figure 11.

[4] With two exceptions: (1) the actor-level Fault Management Actor can send messages to the Fault Manager, as explained in Section 6; and (2) the ground Operations Manager can receive commands (*e.g.*, to upload deployment plans) by some trusted application that includes suitable user authentication.

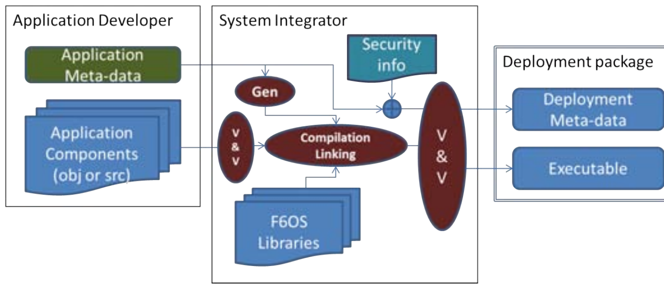

Figure 11. Application development and system integration for F6MDA

The application meta-data includes the following elements:

- Manifest of the files for components constituting the application.
- Interface definitions for all data structures, topics, and interfaces used in the application.
- Resource needs of the application components (CPU time, memory footprint, network bandwidth, etc.).
- Timing specifications for all component operations.
- Fault characterization, *i.e.*, expected anomalies and fault mitigation strategies.
- Manifest of external library dependencies (*e.g.*, POSIX math library).
- Deployment plan for the application.

The application meta-data is used in a number of ways: (1) to generate part of the implementation code (that gets eventually linked into the final application), (2) to combine with integrator-provided security information (specifically security labels) to form the final deployment meta-data, (3) to configure the fault management system, (4) to set policies for QoS driven resource management, and (5) to support both design-time and run-time verification and certification.

Note that the application developer may supply only the object code (not the source code) of the application. The object code is not necessarily trusted; it may attempt to make privileged F6OS system calls. Since a well-behaved, component-based application interacts only with its own components or with specific service actors via well-defined interfaces, the application object code is first checked for unauthorized calls[5]. Next, the developer-supplied meta-data is used to re-generate the interface code (*i.e.*, the client and server stubs) that glue the application code to F6OS and middleware libraries. Then, the application executable is produced using trusted compilers and linkers. Finally, the produced executable and the deployment meta-data is subjected to a V&V regime.

The extensive nature of the V&V and the use of trusted tools in the integration phase provides a path towards certification. Furthermore, F6OS makes certain security guarantees regardless of the behavior of the actors and the correctness of the application V&V process.

The above development process can be realized in two ways:

1. *The conventional process:* The application developer constructs all the software using an implementation language (*e.g.*, C++), and (potentially) uses middleware libraries to access the services provided by F6OS. Technically the developer can develop applications using the core F6OS APIs provided by the F6OS libraries, but this involves a lot of low-level coding and re-building functionality that is provided by the middleware libraries. The developer delivers the application (as source or object code) and the meta-data, as required by the integrator.
2. *The model-driven process:* The application developer performs the architecting and the high-level specification of the application using model-based tools (*e.g.*, an architecture modeling language with graphical tool support), uses the tools to generate the infrastructure ('glue') code needed to integrate the application logic with the F6OS SDK libraries, and adds the 'business logic' of the application using conventional interactive development environment (*e.g.*, a tool like Eclipse). In this scheme, the developer uses the well-established, conventional code development style for implementing the core application functionality, and all the low-level, glue code is auto-generated from the models. The models can be used to instantiate architectural design patterns that solve specific design problems (*e.g.*, primary/backup replication for fault management). The developer delivers the application (as source or object code) and the models of the application (from which the glue code and the meta-data will be re-generated by the system integrator).

The model-driven process is outlined in Figure 12.

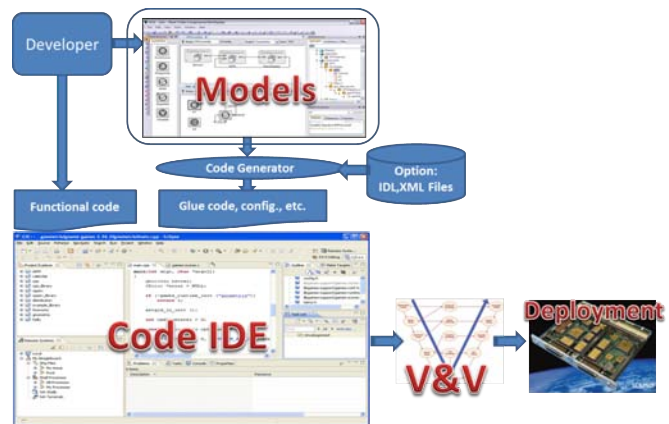

Figure 12. Model-driven development for F6MDA

The *F6 Model-driven Development Kit* (*F6MDK*) includes the tools for model-driven development, including a domain-specific modeling language and its supporting visual modeling environment, the various software generators that produce code and other implementation artifacts, and model-driven tools for model analysis and verification. Other model-driven tools can be used as well. For instance, the business logic of applications can be developed using Simulink/Stateflow and the resulting models (and the code generated from them) integrated into the final application.

The rationale for F6MDK is as follows. The model-driven approach can increase the productivity of the developer (as it will generate the infrastructure code that is hard and error-prone to write). It also offers an opportunity to analyze the system based on the models. Knowing the component interaction patterns and the resource needs (*e.g.*, CPU and memory), interesting properties like end-to-end latency can be determined based on the models. Models can also be converted into a discrete-event simulation model that can

[5]Note that even if unauthorized system calls are made at run-time, they will fail due to security checks performed by F6OS, as described in Section 2.

assist in estimating system performance. The model-driven approach also provides a basis for integrated V&V. Meta-data added to the model elements can be exploited for safety and security analysis at design-time and at run-time.

9. Related Work

An approach to objects based on time-triggered (periodic) and event-triggered (sporadic) methods has been presented in [21]. The approach described is implemented in the form of object structures, and many concepts are similar to our approach. However, there are two differences: we rely on a novel operating system as the underlying platform, and we build a framework on top of that to provide specific services for component interactions and scheduling.

Kuz et al. presented a component model called CAmkES in [22]. They built their system on the L4 micro kernel. CAmkES does not provide temporal partitioning. Instead, it is designed to be a low-overhead system that can run on small computing nodes by enforcing static components (*i.e.*, a singleton and not a session-based component) and static bindings. We had to also enforce similar restrictions in our framework to keep the component interactions simple and predictable.

Delange et al. recently published their work on POK (PolyORB Kernel) [23]. It uses AADL specifications to automatically configure and deploy processes and partitions to a QEMU based emulated computing node. We are currently working on obtaining details about this project. DIANA [24] is a new project for implementing an avionics platform called Architecture for Independent Distributed Avionics (AIDA) using Java as the core technology. One of the challenges in using Java is the threading model, which requires their Java virtual machine, called PERC Pico, to handle the thread scheduling itself (instead of the operating system). This adds another layer of scheduling above the operating system. Hence, they do not provide a one-to-one mapping between a Java thread and an APEX process. Another issue with using Java mentioned in their paper is the complexity in estimating and bounding memory usage per thread, which is a critical requirement for F6MDA. Finally, they also mention that the errors signaled by PERC Pico are hard to diagnose and correct [24]. This is partially due to the extra layer imposed by the Java virtual machine.

Lakshmanan and Rajkumar presented a distributed resource kernel framework used to deploy real-time applications with timing deadlines and resource isolation in [25]. Their system consists of a ‘partitioned’ virtual container built over their Linux/RK platform. They have reported that their framework provides temporal resource isolation because they ensure that the timing guarantees provided to each independent application do hold irrespective of the behavior of other applications, by using CPU as a reserved resource. However, to the best of our knowledge, they do not support process and partition management services as specified for F6OS. Moreover, their framework does not support a component model.

Note that a component model for hard real-time systems has been developed in our earlier work [26]. That work merged the concepts of CCM with those of ARINC-653, and as such it should considered a precursor of this work.

10. Conclusions

Fractionated spacecraft require a new class of information architecture. This paper presented a layered design for such an architecture: F6MDA. The architecture is based on an operating system that provides essential resource management functions and includes a novel combination of other capabilities: time and space partitioning, privileged operations for platform services, multi-level security for information flows, and secure transport capability. This layer is used by a middleware layer that provides higher-level abstractions for synchronous and asynchronous communication based on restricted implementations of the CORBA and DDS specifications. In the top layer, a component model defines a framework for applications built from distributed interacting components. The architecture addresses the cross-cutting aspects of fault management and multi-level security across applications. A prototype implementation of the F6MDA design is in progress.

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 necessarily reflect the views of DARPA. The authors thank Paul Eremenko, Program Manager for his support of the project, Richard A. Golding of KTSi and Olin Sibert of Oxford Systems for their invaluable input and guidance during the design process.

References

[1] O. Brown and P. Eremenko, “The Value Proposition for Fractionated Space Architectures,” AIAA Paper 2006-7506, 2006.

[2] *Model Driven Architecture (MDA) Guide V1.0.1*, OMG Document omg/03-06-01 ed., Object Management Group, Jun. 2003.

[3] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE, Technical Report 2547, Volume I, 1973.

[4] C. Watkins and R. Walter, “Transitioning from federated avionics architectures to integrated modular avionics,” in *Digital Avionics Systems Conference, 2007. DASC ’07. IEEE/AIAA 26th*, oct. 2007, pp. 2.A.1–1 –2.A.1–10.

[5] G. Birkhoff, *Lattice Theory*, 3rd ed., ser. Colloquium Publications. American Mathematical Society, 1967.

[6] Olin Sibert et al., “Multiple-domain labels,” in preparation.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer-Verlag, October 2011. [Online]. Available: http://www.cl.cam.ac.uk/research/hvg/Isabelle

[8] A. Franchi, A. Howell, and J. Sengupta, “Broadband mobile via satellite Inmarsat BGAN,” *IEE Seminar Digests*, vol. 2000, no. 67, pp. 23–23, 2000. [Online]. Available: http://dx.doi.org/10.1049/ic:20000547

[9] *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 ed., Object Management Group, Jan. 2008.

[10] *Data Distribution Service for Real-time Systems Specification*, OMG Document formal/2007-01-01 ed., Object Management Group, Jan. 2007.

[11] *CORBA/e Adopted Specification*, OMG Document formal/2008-11-06 ed., Object Management Group, Nov. 2008.

[12] Object Management Group, *DDS for Lightweight CCM Version 1.0 Beta 2*, OMG Document ptc/2009-10-25 ed., Object Management Group, Oct. 2009.

[13] W. R. Otte, D. C. Schmidt, A. Gokhale, and J. Willemsen, "Infrastructure for Component-Based DDS Application Development," in *To Appear in the Proceedings of the Tenth International Conference on Generative Programming and Component Engineering (GPCE'11)*, Oct. 2011.

[14] *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 ed., Object Management Group, Jan. 2008.

[15] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-backup Approach," in *Distributed systems (2nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216.

[16] *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 ed., OMG, Apr. 2006.

[17] W. R. Otte, D. C. Schmidt, and A. Gokhale, "Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems," in *Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM '10)*, Bengarulu, India, Nov. 2010.

[18] N. Mahadevan, S. Abdelwahed, A. Dubey, and G. Karsai, "Distributed Diagnosis of Complex Systems using Timed Failure Propagation Graph Models," in *AUTOTESTCON, 2010 IEEE*. IEEE, 2010, pp. 1–6.

[19] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," in *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS '09)*, San Francisco, CA, Apr. 2009, pp. 118–127.

[20] J. Balasubramanian, A. Gokhale, F. Wolf, A. Dubey, C. Lu, C. Gill, and D. C. Schmidt, "Resource-Aware Deployment and Configuration of Fault-tolerant Real-time Systems," in *Proceedings of the 16th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS '10)*, Stockholm, Sweden, Apr. 2010, pp. 69–78.

[21] K. Kim, "Object structures for real-time systems and simulators," *Computer*, vol. 30, no. 8, pp. 62–70, Aug 1997.

[22] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[23] J. Delange, L. Pautet, and P. Feiler, "Validating safety and security requirements for partitioned architectures," in *Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*. Berlin, Heidelberg: Springer-Verlag, June 2009, pp. 30–43.

[24] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy, "Use of PERC Pico in the AIDA avionics platform," in *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2009, pp. 169–178.

[25] K. Lakshmanan and R. Rajkumar, "Distributed resource kernels: OS support for end-to-end resource isolation," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 195–204, 2008.

[26] 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. [Online]. Available: http://dx.doi.org/10.1002/spe.1083

Abhishek Dubey is a Research Scientist at the Institute for Software Integrated Systems at Vanderbilt University. He has nine years of experience in software engineering. He conducts research in theory and application of model-predictive control for managing performance of distributed computing systems, in design of fault-tolerant software frameworks for scientific computing, in practice of model-integrated computing, and in fault-adaptive control technology for software in hard real-time systems. He received his Bachelors from the Institute of Technology, Banaras Hindu University, India in 2001, and received his M.S and PhD from Vanderbilt University in 2005 and 2009 respectively. He has published over 20 research papers and is a member of IEEE.

William Emfinger is a Graduate Research assistant at the Institute for Software Integrated Systems at Vanderbilt University. He has 2 years of work experience with real time systems for sensing applications. He intends to focus on real time systems for high-criticality and high-security applications. He received his B.E. in Electrical Engineering and Biomedical Engineering from Vanderbilt University in 2011, and is currently pursuing a Ph.D. in Electrical Engineering and Computer Science from Vanderbilt University.

Aniruddha Gokhale is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, TN, USA. His primary research interests are in investigating novel model-driven engineering (MDE) solutions for systems problems pertaining to distributed real-time and embedded systems. He has more than fifteen years experience in middleware technologies. Dr. Gokhale has published over 100 technical papers. He obtained his B.E (Computer Engineering) from University of Pune, 1989; MS (Computer Science) from Arizona State University, 1992; and D.Sc (Computer Science) from Washington University in St. Louis, 1998. Prior to joining Vanderbilt, he was a member of technical staff at Lucent Bell Laboratories, NJ, USA. Dr. Gokhale is a Senior Member of IEEE and Member of ACM.

Gabor Karsai is Professor of Electrical and Computer Engineering at Vanderbilt University and Senior Research Scientist at the Institute for Software Integrated Systems. He has over twenty years of experience in software engineering. He conducts research in the design and implementation of advanced software systems for real-time, intelligent control systems, and in programming tools for building visual programming environments, and in the theory and practice of model-integrated computing. He received his BSc and MSc from the Technical University of Budapest, in 1982 and 1984, respectively, and his PhD from Vanderbilt University in 1988, all in electrical and computer engineering. He has published over 160 papers, and he is the co-author of four patents.

William R. Otte is a Ph.D. student in the Department of Electrical Engineering and Computer Science (EECS) at Vanderbilt University. His research focuses on distribution and component middleware for distributed real-time and embedded systems, and in particular techniques to ensure correct, predictable, fault tolerant, and adaptable deployment and configuration of component-based applications. In addition, he is interested in techniques for run-time planning and adaptation for component based applications, as well as specification and enforcement of application quality of service and fault tolerance requirements. He received a B.S. in Computer Science and Mathematics from Vanderbilt University in 2005, his M.S. in Computer Science from Vanderbilt University in 2008, and is expected to be awarded a Ph.D. in Computer Science from Vanderbilt University in December, 2011.

Jeffrey Parsons is a Research Engineer at the Institute for Software Integrated Systems at Vanderbilt University, where he has worked for nine years. He has over twelve years experience with middleware and generative programming, and his interests also include model-based design of component systems. He received a B.S in Computer Science from Washington University in St. Louis in December 1998 and an M.S. in Computer Science from the same institution in August 2002.

Csanád Szabó is a Research Engineer at the Institute for Software Integrated Systems at Vanderbilt University. He worked for eight years in the industry in various roles and dealt with topics on networking, mobility management and embedded systems. Two years ago he returned to the research, he worked first on an emotional engine modeling project and joined ISIS recently where he is active in modeling and simulation of distributed and embedded systems. He received his M.S. in Electrical Engineering from the Technical University of Budapest in 2001.

Alessandro Coglio is a Principal Scientist at Kestrel Institute, where he has been working on formal methods, theorem proving, specification languages, provably correct refinement, program generation, the Java Virtual Machine, smart cards, cryptography, and software for fractionated satellites. Prior to joining Kestrel in 1998, Mr. Coglio was a Consulting Researcher at the University of Genoa (Italy), where he worked on theorem proving, discrete event systems, Petri nets, and artificial emotions. Mr. Coglio received a degree in Informatics Engineering in 1996 from the University of Genoa.

Eric W. Smith is a Computer Scientist at Kestrel Institute, with expertise in theorem proving, formal methods, and security. He has over a decade of experience in formal verification of software and hardware, in both academia and industry, and he is an expert user of the ACL2 theorem prover. In 2011, he received his Ph.D. in Computer Science from Stanford University. His dissertation work focused on automatic formal equivalence checking of cryptographic programs, including block ciphers and cryptographic hash functions.

Prasanta Bose is a Principal Scientist at the Advanced Technology Center, Lockheed Martin Space Systems. He leads the distributed autonomous systems group. His research is focused on software and systems architectures of complex systems that leverage principles from cross-disciplinary fields of communications, computation and control and their model-driven engineering for safety and security. He received his PhD in Computer Science from University of Southern California and his MS in Electronics and Communications from Indian Institute of Science, Bangalore.