A Middleware Architecture for Unmanned Aircraft Avionics

Juan López, Pablo Royo, Enric Pastor, Cristina Barrado, Eduard Santamaria
Technical University of Catalonia
Avda. del Canal Olimpic s/n
08860 Castelldefels, Spain
{lopez,proyo,enric,cristina,esantama}@ac.upc.edu

ABSTRACT

An Unmanned Aerial Vehicle is a non-piloted airplane designed to operate in dangerous and repetitive situations. With the advent of UAV's civil applications, UAVs are emerging as a valid option in commercial scenarios. If it must be economically viable, the same platform should implement a variety of missions with little reconfiguration time and overhead.

This paper presents a middleware-based architecture specially suited to operate as a flexible payload and mission controller in a UAV. The system is composed of low-cost computing devices connected by network. The functionality is divided into reusable services distributed over a number of nodes with a middleware managing their lifecycle and communication. Some research has been done in this area; yet it is mainly focused on the control domain and in its real-time operation. Our proposal differs in that we address the implementation of adaptable and reconfigurable unmanned missions in low-cost and low-resources hardware.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Software Architectures; J.7 [**Computers in Other Systems**]: Command and control

General Terms

Middleware Design and Implementation

Keywords

UAV, Avionics, Embedded, Middleware, Service-based, Publish-Subscribe

1. INTRODUCTION

An Unmanned Aerial Vehicle (UAV) is a non-piloted airplane designed to operate in situations in which the utilization of a traditional airplane could be dangerous. Nowadays and after many years of development, UAV's technology is reaching the critical point in which it can be applied in a civil/commercial scenario.

Basically, an UAV is automatically piloted by an embedded computer called the Flight Computer System (FCS)[15]. This is a system that reads information from a wide variety of sensors (accelerometers, gyros, GPS receivers, pressure sensors) and drives the UAV mission along a predetermined flight-plan. The airframe also carries some payload to transform the UAV into a valuable observation platform: TV or IR cameras, sensors, etc. The information generated by the payload can be processed on-board or sent to a ground control station via some communication infrastructure, such as radio modems or microwave links. Finally, some intelligent component, the mission controller, orquestrates all these components and automates the UAV task to its operator.

Many types of UAVs exist today; however the class of mini/micro UAVs is emerging as the valid option if this civil commercialization scenario is kept in mind. This type of UAV has the same limitations as most embedded systems: limited space, limited power resources, increasing computation requirements, complexity of the applications, time to market requirements, etc. All these stringent requirements are amplified in civil/commercial applications. In that context, the same platform should be able to implement a large variety of missions and operate with many types of payload; all of it with little reconfiguration effort and overhead if the system has to be economically viable. For this reason we believe that the effective application of UAVs in civil operations requires implementing new hardware/software systems that provide specific support to automatically control the actual missions to be carried out by the UAV.

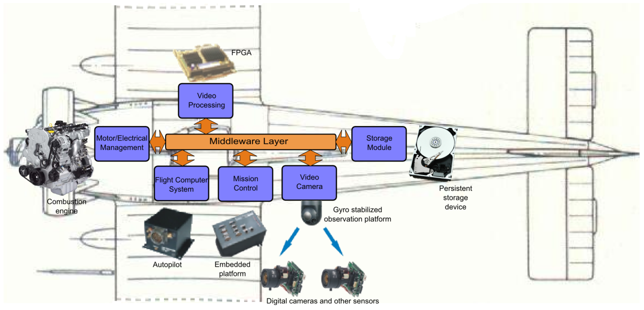

Figure 1: Unmanned Aircraft Services Architecture.

This paper presents a middleware specially suited to operate as a flexible mission and payload controller in an UAV.

The functionality of the system is divided into a set of reusable services that can be distributed over the different computing nodes of the UAV. A middleware manages the lifecycle and the communication between services, operating the global system as a Distributed Embedded System, as figure 1 shown.

The paper is organized as follows. First, section 2 describes the previous work in this area. Next, section 3 shows an overview of the proposed middleware. Section 4 describes the communication primitives it offers to the services. An application example is shown in section 5. Finally, section 6 provides some details about the implementation and section 7 concludes the article and shows the future lines of research.

2. PREVIOUS WORK

This section presents previous research in the area of avionics for civil use, in particular in the area of middleware for this type of embedded systems. For a more general review of middleware refer to [5] which presents the challenges and available technical solutions of several middleware approaches for distributed embedded systems. It is a nice and concise survey on middleware uses on mission-critical dynamic domains.

Classical papers on avionics buses [12, 4, 9, 2] are focused on real-time capabilities, in particular on flight control [9] and verification [12, 2]. If previous research targeted on the military domain, nowadays the challenge is civil avionics because their applications must be easily adaptable and reconfigurable. Also it is important to base solutions on low-cost and low-resources hardware. In these applications services and sensors are a key issue [6, 15, 11, 14, 2]: Their initial configuration, their description, their verification or their dynamic reconfiguration are several of the research targets.

Referring to middleware families most of research papers are based on CORBA Component Model [7, 12, 6, 2, 16]. For example, [12] presents Bogor, a model checking framework that models the semantics of real-time CORBA using a general complexity checking model. In the case of [6], Edwards et al. present how to automate the configuration of services in presence of quality of service capabilities. The testbed is an avionics system with 50 components of the CORBA Component Model. Finally in [7] Jung et al. implement a heuristic to understand the semantics of mission asynchronous events.

Another trend is ad-hoc middleware development. For example, [9] report presents the experience of an open implementation of a middleware for Java applications named Ovm. [4] presents the work developed under the A3M project. A novel middleware focused on space applications is developed. The main interest comes from the requirements met by the middleware: real-time and dependability. Two protocols are developed and verified under a real-time OS simulator. Middleware specially targeted to sensors are [15] and [14]. In [15], Zhang et al. have developed a middleware to manage large number of wireless sensors with delay tolerance but power management constraints. Diversity of sensors, large number of them and the final developed application are main target of this middleware. [14] presents the extension of the Gridkit sensor middleware to cover dynamic configuration and customization of a large sensor network.

Service oriented architectures (SOA) are getting common in several domains, for example Web Services [17] in the Internet world and UPnP [1] in the home automation area. SOA is an architectural style whose goal is to achieve loose coupling among interacting components or services. Some middleware proposals [11, 8] have also been influenced by this architecture. The work in [11] presents an extension of the FOCUS theory; this is a formalism to describe SOA services. They compare SOA services against CORBA objects and show the benefits of the service oriented approach. Their target application, though, is not in the aeronautics domain but on the automotive. In [8] it is presented a novel middleware also based on a service oriented model with publish/subscribe messaging but with the particularity of adding video processing capabilities. The middleware is tested on a network of thousands of sensors cameras.

Avionics testbeds are mainly done for general aircraft systems [12, 6]. Only [9] uses a testbed of flight demonstrations with a small UAV. Specific avionics buses testbeds are in [2], which presents the reengineering of a McDonnell Douglas software to be able to reuse the system elements across platforms and introduce a new software physical architecture for product line development. Also [7] uses the Boeing Bold Stroke avionics to enhance the CORBA Component Model middleware and to study the correlation between events to detect special situations known as semantic events.

In our approach the middleware is developed for high level mission design, concentrating efforts in functionality more than in real-time issues. Like in [16] which shows the advantages of reducing CORBA functionalities (NEST and OEP) in benefit of efficiency, we believe that a small number of really useful functionalities are more important and efficient that large implementations of today middleware. Also [13] argues that due to the complexity and completeness of many middleware platforms efficiency may be compromised. The paper studies several ways to bypass the middleware layer (or part of it) to improve efficiency.

3. MIDDLEWARE ARCHITECTURE

Middleware-based software systems consist of a network of cooperating components, in our case the services, which implement the business logic of the application and an integrating middleware layer that abstracts the execution environment and implements common functionalities and communication channels. In this view, the services are semantic units that behave as producers of data and as a consumers of data coming from other services. The localization of the other services is not important because the middleware manages their discovery. The middleware also handles all the transfer chores: message addressing, data marshaling and demarshalling (so subscriber services can be on different platforms than the publisher service), delivery, flow control, retries, etc. Any service can be a publisher, subscriber, or both simultaneously. This publish-subscribe model virtually eliminates complex network programming for distributed applications.

In our architecture, the middleware takes the form of a component called **service container**. The services are executed and managed by a service container that is unique in each node of the distributed system network. The service container can manage several services and provides common functionalities (network access, local message delivery, name resolution and caching, etc.) to the services it contains. The key benefit is that services are entirely decoupled. Very little design time has to be spent on how to handle their mutual interactions. In particular, the services never need informa-

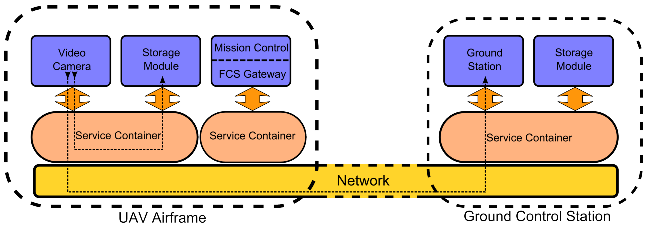

Figure 2: Middleware Architecture.

tion about the other participating services, including their existence or locations. The container automatically handles all aspects of message delivery, without requiring any intervention from the service, including: determining who should receive the messages, where recipients are located and what happens if messages cannot be delivered.

As seen in figure 2, the container can communicate services installed in the same container or in other service container present in the same local network. The protocols used are designed to maximize the performance by using the multicast capabilities of the underlying network. The service container supports mechanisms that go beyond the basic publish-subscribe model. More concretely, the main functionalities that provide the service container are:

- Service management: The container is the responsible of starting and stopping the services it contains. It is also on charge of watching for their correct operation and notifying the rest of containers about changes in the services status.
- Name management: The services are addressed by name, and the Service Container discovers the real location in the network of the named service. This feature abstracts the programmer from knowing where the service resides or how to communicate with it. In case of service malfunctioning, it is also the container responsibility to notify the other containers in the domain and to choose another provider service if it is available. In this way, the containers are able to clear and update their caches. From the name management point of view, the Service Container acts as a proxy cache for the services it contains.
- Network management and abstraction: The services do not access the network directly. All their communication is carried by the service container. This abstracts the network access, allowing the middleware to be deployed in different networks. Moreover, the container hides the bookkeeping related with the management of UDP/TCP ports and multicast groups.
- Resource management: Given that each network distributed node has a unique container, and that all the services in that node are layered on top of it, the container is the right place to centralize the management of the shared resources of the node: memory, CPU time, input/output devices that are accessed in exclusive mode, etc. In some cases, for example when the node is a low-end microcontroller, the service container can act as a minimal operating system.

Several middleware for embedded systems have been proposed in both the academy and industry. Our proposal is focused on the network centric low-resources embedded applications. Most of the existing middleware promotes a distributed computing paradigm; however our target application, UAV avionics, suggests the use of a global data space approach. In this environment, most communicating components are sensors that spread its samples to several controlling components. These components evaluate the data from several sources and again send control data to many actuator components.

From the point of view of a distributed application there are basically three models for information communication: Point-to-Point, Client-Server and Data Distribution System (DDS). DDS model arose as a solution of most novel distributed applications today. It promotes a publish/subscribe model for sending and receiving data, events, and commands among the nodes. Nodes that are producing information publish that information and other nodes can subscribe to them. The middleware layer takes care of delivering the information to all subscribers that declare an interest in that topic. The DDS model has been shown as a very good solution for many-to-many communication frameworks. They are also very efficient for distributing time-critical information.

4. COMMUNICATION PRIMITIVES

For the specific characteristics of a UAV mission, which may have lots of systems which may interact many-to-many, the proposed solution is based in the DDS paradigm. Many DDS frameworks have been already developed, each one contributing with new primitives for such open communication scenario. In our proposal we implement only the communication primitives required by a minimalistic distributed embedded system in order to keep it simple and soft real-time compliant. The UAV mission and payload control has been used as a motivating example and guiding application. This section describes the proposed communication primitives, which have been classified in four types: Variables, Events, Remote Invocation and File Transmission.

4.1 Variables

As variables, we mean the transmission of structured, and generally short, information from a service to one or more additional services of the distributed system. This information is sent at regular intervals or each time a substantial change in its value occurs. This relative caducity of the information allows to send it in a best-effort way. The system should be able to tolerate the loss of one or more of these data transmissions. This communication primitive follows the publication-subscription paradigm.

A service can provide zero or more variables. Each of them is composed of a basic type (boolean, integer, floating point real, character string, etc.) or by a composition (vector, struct or union) of basic types. From the point of view of the allowed data types in a variable our middleware is similar to a C-like language. By means of its service container, a service announces the availability of its variables. This way, other services present in the distributed embedded system can subscribe themselves to one or more of these variables. From the moment a service subscribes to a variable, the provider service is responsible for sending it with the accorded quality of service characteristics. In any case, the services using this communication primitive should tolerate the loss of some variable values. If this situation goes on, the service container will warn of this timeout circumstance to the affected services.

The provider service can specify the variable validity as a quality of service parameter. When a variable value is lost, the subscribed services can receive previous values as long as they are still valid. In addition, the middleware has a mechanism that guarantees an initial exact value for the services that need it. The service container maps this sort of communication over UDP packets in broadcast or multicast mode, when the underlying network allows it. This sort of transmission allows optimizing the bandwidth use because one packet sent can arrive to multiple nodes in the distributed embedded system.

4.2 Events

Events are similar to variables in the sense that both work following the publication-subscription paradigm. The main difference is that events, opposite to variables, guarantee the reception of the sent information to all the subscribed services. The utility of events is to inform of punctual and important facts to all the services that care about. Some examples can be error alarms or warnings, indication of arrival at specific points of the mission, start of some pre-programmed actions like taking a photo, etc. Events can contain associated information (error codes, current position, etc.) or have meaning by themselves. When they carry additional information, data is coded the same way as in the variable communication primitive. In the case of events, another important fact that has to be taken into account is latency. Reservation of time slots in both the processor and the network will ensure this critical constraint. The publisher and subscriber services interact with each other always using the service container like in the variable case. Finally, this communication primitive is mapped by the service container over TCP or over UDP using a mechanism to acknowledge and resend lost packets. This specific retransmission mechanism in the application layer is more efficient for event messages than the generic case provided by the TCP stack.

4.3 Remote invocation

Most middleware implements some way of distributed computing based on the remote procedure call paradigm, for example ONC RPC, CORBA or Web Services. However, for some distributed embedded domains the data publication-subscription or global data space paradigm seem more appropriate. For this reason we provide a first-class set of publish-subscribe communication primitives.

Nevertheless, remote invocation is an intuitive way to model some sort of interactions between services. Some examples can be the activation and deactivation of actuators, or calling a service for some form of calculation. Thus, in addition to variables and events, the services can expose a set of functions that other services can invoke or call remotely. The functions exposed by a service can have an arbitrary number of parameters and optionally a return value. This communication primitive implements two-way point-to-point communication between two services; one acts as a client and the other as server. The client service is always the initiator of the communication and the server service location is abstracted by the middleware. However, a difference from previous communication primitives is that the client service is not continuously subscribed nor connected to the server service. Their relation is occasional and delimited by the time the invocation is executed.

During middleware initialization, the services check that all the functions they need to properly accomplish their task are provided by one or more services available in the network. Redundancy and fault-tolerance are managed by the middleware, that can also redirect remote calls to server services statically or dynamically. Static allocations of the client-server relationships are useful in critical services where resources like CPU time or network bandwidth are pre-allocated. On the other hand, runtime information can be used to redirect calls to non-critical services to the server where best performance is expected. For this, load balancing techniques are used. Upon service failure, if another service is implementing the same functionality, the middleware will detect the situation and redirect requests to the redundant service. This allows the system to continue its mission, although perhaps in a degraded mode. If no service provides the requested function the middleware will warn the system to take the programmed emergency procedure.

In some cases, both event and remote invocation primitives can be applied to realize a same functionality. In this case, in our current implementation, events seem faster than their function equivalent. This communication primitive is generally mapped by the service container over TCP, but UDP plus retransmission at the middleware level can also be used. This primitive is never mapped over broadcast or multicast given that is always a point-to-point communication.

4.4 File-based transmission

In our preliminary prototypes, it has been discovered that some requirements of the mission are not fulfilled by the proposed communication primitives. In some cases, there is the need to transfer with safety continuous media. This continuous media includes generated photography images, configuration files or services program code to be uploaded to the service containers. Some modifications could be done to the previous primitives to accept this sort of information transfer. But finally, a specific primitive has been developed to treat this case, given the huge performance benefits that can be attained. This primitive is basically used for transferring long file-structured information from a node to many others.

This primitive implements a protocol loosely based on Starburst MFTP [10]. It has three phases: announce, transfer and completion. On the first phase the service publishes the availability of a resource and the interested services subscribe to it. The file is divided in equally sized chunks and each participating service know the total size, the number of chunks and the revision of the file. Revision numbers identify different versions of the same resource and allow the services to know when a change happens. During the transfer phase the publishing service will continuously send chunks in multicast mode to all the subscribers. Obviously, the UDP packets can be lost or arrive unordered and then all the chunks are numbered to receiver being able to reconstruct the original file. When the publisher service has sent all the chunks it asks the subscribers for its completion status. If a subscriber has all the chunks, it sends an ACK to the publisher and it removes finished receivers from its subscribers list. In the other case the subscriber sends a NACK with a compressed list of the chunks it lacks. The publisher begins a new transfer phase only with the asked chunks and this process iterates until the subscribers list is empty.

This protocol is designed in a way the phases can overlap for different subscribers. In fact, when the transfer is on-going, a new service can subscribe to it and resume at the current point. At the completion phase it will ask for all the chunks sent before it was connected to the transfer. Subscribers can also be notified of revision changes to the file and can decide if they go on with the transfer in progress, they start a new transfer with the new revision or both. Obviously, to minimize the overhead, in the case of communicating services in the same service provider, the transfer is bypassed by the container as direct access to the resource.

5. APPLICATION EXAMPLE

To check the applicability of this service model, we have several UAV avionics use cases showing the generic service reutilization and the flexibility provided by the implemented communication primitives. Here, we are going to describe a simple use case but complex enough to use all the primitives. The proposed scenario in figure 3 is composed by several services that interact to capture images at specified locations of the flight and to process them in an on-board FPGA based system.

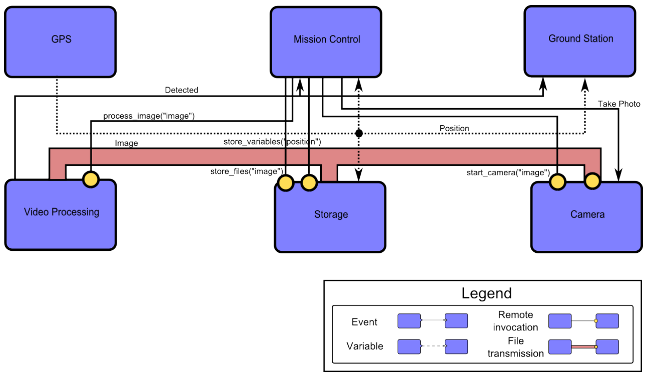

Figure 3: Image processing scenario.

The starting service is the GPS which generates the *position* variable containing the geographic coordinates that are consumed by the Mission Control and the Ground Station. The *position* is a high rate changing data and the consumer services can lost some values without problem, then the variable primitive for its high efficiency is preferred over the safer event primitive.

The Ground Station (GS) service represents the station where the operator checks and controls the UAV operation. In this simple use case, the ground station basically shows the subscribed variables and events in a terminal. The Mission Control (MC) is a service that monitors the status of the mission and following a provided flight plan orquestrates the rest of services to autonomously accomplish the mission. In this case the MC is instructed to take high resolution photos at specified locations and to process them onboard to detect specific characteristics on the image. Before arriving the first location, the MC instructs the camera to prepare itself to take photos and publish them with the specified name.

The storage service is a generic service that provides storage and retrieval of data by providing access to an inner file system. It is told to store the photos and the GPS positions by the MC. At the same time, the video processing module is told to process the same file resource. All these initialization have remote call semantics, mainly because the MC consumes the operations provided by the different services. Later on, the MC will notify the camera with an event each time the aircraft arrives a position where a photo should be taken. The multicast file transfer will be then used for efficiently sending the image to the storage and video processing modules. If the video process detects the pre-programmed characteristics in the image it can notify the GS and MC.

In this scenario all the services are generic enough to be reutilized in most of the UAV missions and shows the applicability and usage of all the provided communication primitives.

6. IMPLEMENTATION

In this section we are going to describe briefly some details of the middleware implementation. Our implementation follows the PEPt architecture [3] in which presentation, encoding, protocol and transport subsystems are decoupled and accept new pluggable subsystems. Presentation provides the datatypes and APIs available to the service programmer. Encoding describes the representation of these data on the wire. Protocol frames the the encoded data to denote the intent of the message. Protocol subsystem is also responsible for frame retransmission and other low level bookeeping tasks. And finally, transport moves the resulting frames from one node in the network to another.

This subsystem decoupling allows us to test and evaluate different algorithms and implementations for the same layer very easily. In figure 4, it is shown the main classes of the implementation layered on the different PEPt subsystems.

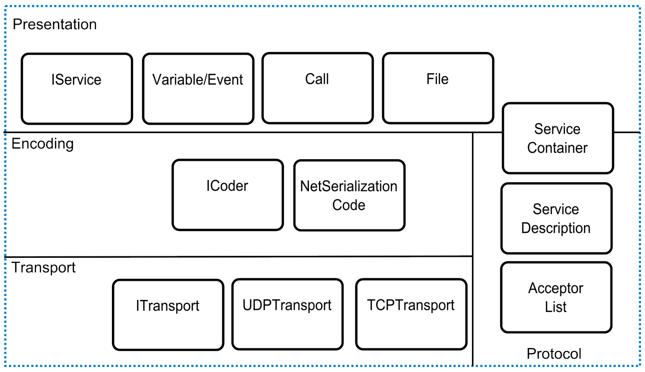

Figure 4: Class Diagram.

In addition to the PEPt subsystems, our implementation also have an pluggable scheduler that queues and arranges event/variable handlers and service calls execution. This simple approach to scheduling can only support soft real time, not enough to control applications but adequate for the applications we are currently focused. In fact, current scheduler implementation is basically a simple thread pool with fixed priorities for each named primitive and relaying in standard system threads.

The current minimalistic prototype is based on Microsoft C# and has 36 classes and less than 1500 lines of code. We are currently doing functional analysis with several avionics use cases, performance and soft real time compliance will be tested next. The service model and its communication primitives has demonstrated that are flexible and simple enough to easily distribute existing UAV applications. For example, the telemetry interface with FlightGear simulator has been done by a person without previous knowledge of the architecture in only 2 days.

7. CONCLUSIONS

This paper presents a middleware for an UAV avionics that permits a rapid, efficient and low-cost mission definition and execution. The paradigm of the presented architecture is the full distribution of services in the form of net centric applications. The services are semantic units that behave as producers of data and as consumers of other data coming from other services. The localization of the other services is not important. Data can come from services in the same physical node or from a physically Ethernet connected node. The middleware makes transparent the physical distribution of communications and services. It creates a virtual global data space based on publication-subscription primitives and enhances data transfer depending on their semantics.

The presented architecture and middleware are designed to hide hardware complexity to the applications, being able to implement a large variety of missions with little reconfiguration time. The evaluation of the proposal has been done with the implementation of a prototype solution. The benefits can be measured in terms of productivity and return of investment. The development time was shown to be very short and thus we assume a high productivity for forthcoming applications that can rely on a model solution.

As a future work we plan to introduce real-time approach for the critical events and services. For execution scheduling, we plan the introduction of a real time operating system. At the same time we plan to implement more civil UAV applications to verify the characteristics of the provided communication tools and concept. Improving efficiency while ensuring real-time is also a key issue for future versions of the middleware.

8. ACKNOWLEDGMENTS

This work was supported in part by the Spanish Ministry of Education (MEC) under Grants TIN2007-63927 and TIN-2004-07739.

9. REFERENCES

[1] Upnp forum. http://www.upnp.org.

[2] B.S.Doerr and D.C.Sharp. Freeing product line architectures from execution dependencies. Technical report, Boeing Report.

[3] H. Carr. PEPt. A minimal RPC architecture. In *OTM 2003*, Italy, Nov 2003.

[4] C.Honvault, M. Roy, P.Gula, J.C.Fabre, G. Lann, and E.Bornschlegl. Novel generic middleware building blocks for dependable modular avionics systems. *EDCC 2005, LNCS*, 3463:140–153, 2005.

[5] D.C.Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, pages 43–48, Jun 2002.

[6] G.Edwards, G.Deng, D.C.Schmidt, A.Gorkhale, and B.Natarajan. Model-driven configuration and deployment of component middleware publish/subscribe services. *GPCE 2004, LNCS*, 3286:337–360, 2004.

[7] G.Jung, J.Hatchiff, and V.P.Ranganath. A correlation framework for a CORBA component model. *FASE 2004, LNCS*, 2984:114–159, 2004.

[8] H.Detmold, A.Dick, and K.Falkner. Middleware for video surveillance networks. In *MidSens'06*, Melbourne, 2006.

[9] J.Baker et al. A real-time java virtual machine for avionics, an experience report. Technical report, Purdue University, 2006.

[10] K.Robertson, K.Miller, M.White, and A.Tweedly. Starburst Multicast File Transfer Protocol (MFTP) Specification. Technical report, Internet Draft, Internet Engineering Task Force, April 1998.

[11] M.Broy, I.H.Krüger, and M.Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, Feb 2007.

[12] M.Hoosier, M.B.Dwyer, and J.Hatcliff. A case study in domain-customized model checking for real-time component software. *IsoLA 2004, LNCS*, 4313:161–180, 2006.

[13] O.E.Demit, E.Wohlstadter, and S.Tai. An aspect-oriented approach to bypassing middleware layers. In *ACM AISD'07*, Vancouver, 2007.

[14] P.Grace, G.Coulson, G.Blair, B.Porter, and D.Hughes. Dynamic reconfiguration in sensor middleware. In *ACM MidSens'06*, Melbourne, 2006.

[15] P.Zhang, C.M.Sadler, and M.Martonosini. Middleware for long-term deployment of delay-tolerant sensor networks. In *ACM MidSens'06*, Melbourne, Nov 2006.

[16] Venkita Subramonian et al. Fine-grained middleware composition for the Boeing NEST OEP. In *OMG Workshop on Real-time and Embedded Distributed Object Systems*, July 2002.

[17] W3C. W3c note: Web services architecture. http://www.w3c.org/TR/ws-arch.