Off-the-shelf Embedded Middleware Solution for UAVs HW-SW Platform Development

J. Barba, F.J. Villanueva, M. J. Abaldea, D. Villa, O. Acena and J. Carlos López
ARCO Research Group. School of Computer Science.
University of Castilla la Mancha. Ciudad Real (Spain)
Email: jesus.barba@uclm.es

***Abstract*—In this paper a Commercial-off-the-Self (COTS) platform for Unmanned Aerial Vehicles (UAV) is presented. The goal of this work is to provide the industry with a flexible and efficient solution to smooth the integration and heterogeneous development challenges in this scenario. On one hand, the proposed platform enables transparent and efficient interaction with the control station implemented in ZeroC Ice. On the other hand, the proposed approach abstracts both embedded and desktop software developers from the platform details. A customized hardware-software layer assures a high-level, efficient reliable communication while a complete tool-chain automatizes the generation of application specific code, reducing the development time.**

***Index Terms*—Middleware; UAV; ZeroC-Ice; Hw-Sw Integration;**

I. Introduction

Unmanned Aerial Vehicles (UAV) are extending its application field from military scenarios to civil applications [1]. The domain of civil UAV applications is dominated by low-cost vehicles which are devoted to specific tasks like smart agriculture [2], smart city [3], rescue tasks [4], etc. These type of UAVs usually rely on an embedded board with key features such as low-weight and low power consumption so as to extend UAV operation lifetime.

One of the most promising approaches for next-generation UAV on-board platforms is the use of hybrid systems where micro-controller and reconfigurable logic come together, for example. These type of boards are flexible enough to support a wide range of application fields, ranging from vision-based applications - where FPGAs (Field-Programmable Gate Array) can apply their computing power to avoid sending raw multimedia data through the wireless link - to sensor-based applications, cryptography, etc.

In this paper an object-oriented software platform for UAV application development is introduced.

The main contributions of our work are:

- Seamless integration of application specific hardware and software components which eases the development of hybrid systems. This is achieved because of the use of a common, high-level model which abstracts our platform functionality (either hardware or software) as a series of objects.
- Reduction of the development time. The developer is provided with a toolchain (interface definition language and interface compiler) which automatizes the generation of the integration infrastructure.
- Optimal and efficient implementation of the middleware engine. Traditional distributed object-oriented frameworks require excessive resources for most of the UAV HW platforms. As it will be shown later, the footprint of the core components is minimized in order to fit in the smallest embedded boards.

The remainder of this article is organized as follows. Next section describes the state of art in UAV application development and also establishes a set of requirements for a software platform in this application field. Section III and IV thoroughly describes the proposed architecture along with interface specification and development process. In section V the communication engine called IceC is described. Finally, some of the most relevant quantitative and qualitative aspects of the prototype are described, along with the main conclusions.

II. Previous works

There are a great market opportunity and a lot of companies are setting up their own platforms to support UAV applications. As consequence of this great expansion of these type of vehicles, there is a lack of embedded standard computing platforms for UAV app development.

Already in 2001, Boeing realized the need of a middleware for this type of vehicles, in [5] Boeing proposes an Open Control Platform (OCP) in order to deal with UAV control tasks. OCP uses ACE (Adaptive Communication Environment) an TAO (an ACE ORB) for flight control tasks modeling following the CORBA model. A similar approach is followed in [6]. We agree in this object-oriented vision but we extend to HW components in order to deal with HW implemented algorithms (e.g a vision-processing VHDL algorithm) like any other software object in a transparent way.

The component model also has been applied to UAV market. In [7] a C++ component-based middleware is developed providing with components related mainly with flight mission. Again there is a lack of a toolchain in order to automate the work in application specific components or any reference to HW component integration.

The service-oriented middleware (SOM) has been proposed for UAV development, in [8] an exhaustive listing of needed services for UAV application development is elaborated. In spite of the well-know set of benefits of SOM (quite similar to object-oriented middleware) in terms of software reuse and time-to-market reduction, there is no specific proposal in this work about toolchain or specific interfaces so it is about a top-level design exercise.

Data Distribution Service (DDS) from Object Management Group (OMG) [9] is used in the UAV field. When communications or interaction between components are stateless is a good approach however when we have to model more complex interactions with state, this approach doesn't fit with this type of applications.

Specific middlewares have been developed for specific tasks (coordination functions [10], avionics subsystems [11], etc.), we are more devoted to design a generic middleware providing guidelines and tools for implement any type of functionality.

None of the middlewares presented in this section have considered hybrid platforms or how to integrate HW components. We consider that there is a lack of solutions for this type of platforms which will play a key role in future UAV development.

Our motivation is to build a middleware with:

- Focus on hybrid platforms e.g. support hardware components and with transparent integration with the rest of the infrastructure.
- A well-defined set of interfaces as guideline for developers but providing the flexible reusability to adapt to each specific application.
- A toolchain for automatic stub and skeleton generation to reduce the error-prone tasks of communication between components of the system (e.g UAV to ground station) and for interaction between SW and HW components.
- Support for a efficient, commercial, legacy object-oriented middleware. In order to avoid *reinventing the wheel* effect, ZeroC Ice [12] is used as the starting point. The idea is to integrate the UAV control and communication operations as another remote object in the system. Thus, the application running in the ground station, could access the services provided by the UAV in a transparent way, using the standard middleware mechanisms.

How to fit all these requirements is a cross-layer challenge that it has not been addressed by previous works.

III. UAV EMBEDDED PLATFORM

Although many of the concepts and modules developed in our solution are platform agnostic, we set the Silica Xynergy board as the initial target for our solution. The reasons behind this election stem from lessons learned in a former versions of the UAV-platform and the evolution of the application requirements. Among this requirements, it is worth noting: cost-reduction, high-performance computing capabilities, a lower power budget to increase flying time and a reduction of the overall weight of the solution.

The Xynergy board is a heterogeneous platform which consists of a ARM Cortex-M3 based STMicroelectronics controller and a low-cost FPGA from Xilinx. Due to a rich collection of dedicated and general I/O ports and built-in protocol interfaces, the Xynergy board is suitable for a wide range of applications.

The FPGA fabric, a Spartan-6 chip, is tightly coupled to the ARM controller and is intended to hold Hw accelerating blocks for computing intensive applications: video, digital filtering, cryptography, etc. FPGA technology provides a close to custom hardware performance while being able to change their behavior through time.

The development of applications and custom IP (Intellectual Property) cores for the Xynergy board can be accomplished by means of off-the-shelf tools freely available for multiple platforms.

A. Hardware support

In this section, a description of the System-on-a-Chip deployed in the FPGA fabric is presented. The role of the reconfigurable device is multiple and comprises:

- Interface between the DDR3 RAM memory and the ARM controller which has no other mechanism to access the data.
- Accelerator of computing intensive functions by means of dedicated hardware.
- Communication manager through a tailored hardware controller.

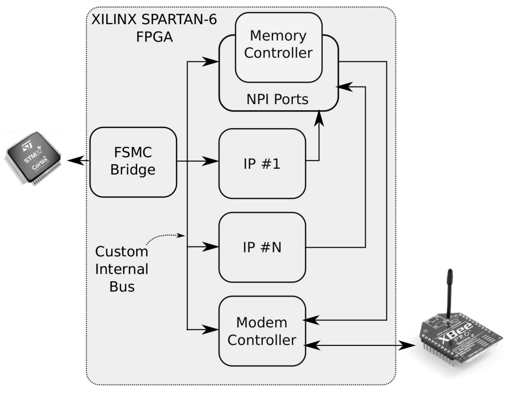

Fig. 1. Architecture of the SoC deployed in the Spartan-6 FPGA to support UAV embedded operations.

Figure 1 graphically represents the main modules and the architecture of the hardware infrastructure which supports the above-mentioned functionality.

The FSMC (Flexible Static Memory Controller) bridge implements a translator between the NOR/SRAM memory access protocol and the *in-house* internal bus protocol.

The DDR3 access has been implemented using the Xilinxs Multi Port Memory Controller core, configuring a minimum of two 64-bit NPI ports depending on the number of cores that must have access to the RAM memory.

The modem controller abstracts the access to the radio/wifi modem providing the same low-level operations for modem configuration and data transmission/reception (i.e. source and target MAC addresses, reset, baud-rate, memory map, etc.).

The controller logic generates the corresponding AT commands to the network modem which is connected to the FPGA via a serial link.

Finally, several IP cores can be instantiate whenever there were available resources in the FPGA. The functionality of each core may vary depending on the application, but the interface to the rest of the system remains unaltered.

Low-level Hw-Sw interfacing: Communication between the software routines and the hardware components is implemented through a register-based interface and an interrupt mechanism. First, 16 memory words are assigned to the control and configuration registers for each processing IP core. Then, one or more memory blocks are assigned to the hardware cores. The number and the size of the blocks is defined by the application developer at design time. These memory blocks are used as input/output buffers where processing cores will write the data produced as the result of their operation or read the input data to be processed. Thus, it is established a producer-consumer integration pattern between the involved pairs in the communication process.

The core bus wrapper decodes the addresses only within the control and configuration memory range. The content of the buffer memory can be accessed through the memory controller by the software routines but not through the individual core wrappers. Processing IPs are configured with the base addresses for their respective input/output buffers.

The modem controller is an example of processing core. Configuration and control operations are done by means of regular write and read FSMC primitives. The FSMC bridge translates them into bus transactions and the register content is written or read. To send a data packet via the radio interface, the processor writes the content of the packet in RAM memory and then signals the core via a write operation that the packet is ready to be sent. The value to write indicates the modem controller the total size in bytes of the packet. Once the packet has been sent, the modem controller triggers an interrupt that wakes up a service routine responsible for checking error conditions.

The receive operation for a data packet works conversely; first the modem controller signals the processor via an interrupt that a new packet is ready to be processed. The modem controller will have written the content of the packet in main memory, into the output buffer region assigned. The service routine will, finally, process the incoming packet.

IV. Software Platform

The FPGA SoC, previously described, provides the UAV platform with hardware support for essential services such as external data storage and efficient radio packet management. Also, a placeholder for UAV hardware-accelerated functions and a protocol to access them and manage the movement of the data is available to developers.

Based on a traditional register/memory mapped approach, a layered software infrastructure has been built firstly. The ultimate goal is to abstract all the communication details to the application level so that the embedded software developer will only have to deal with a set of high-level, automatically generated software routines.

The μC/OS-II operating system is at the basis of the software stack due to its low footprint, real-time and task model features.

In figure 2, the reader can find a representation of the whole software architecture for the middleware infrastructure.

Focusing on the physical and transport layers, a customized FSMC driver has been coded in order to be compliant with the functional specification of the FSMC bridge; providing read and write primitives to access the DDR3 and radio IPs. The radio driver is a standalone task in charge of transferring packet fragments from/to the DDR3 memory and provides the next layer in the hierarchy (*reliable network layer*) with *send* and *receive* primitives.

A send operation copies the packet fragment in the output buffer assigned to the radio controller and then signals the hardware to read it from memory and transfer to the radio interface. The radio driver manages the DDR3 output buffer as a circular buffer using a fixed size vector with all the necessary control information. A receive operation exposes a bit more complex behavior. Once the radio controller has finished writing a packet fragment in its DDR3 input buffer, it raises/triggers an interrupt that is trapped by the radio driver. Depending on the nature of the packet, the radio driver must generate either the acknowledge messages (a new fragment has been received) or update the sending status of a packet (when an acknowledge has been received) which will be interpreted by the reliable network protocol layer later on.

The network layer implements a reliable transmission protocol since it is foreseen the use of restrictive wireless links for UAV to ground communication. The IEEE 802.15.4 RFC [13] has been considered for its use in the implementation of the first prototype of the UAV platform. This standard is suitable for low-cost and low-power wireless communications at the price of a low-bandwidth and small MTU (Maximum Transmission Unit). The overhead per fragment is small (just 4 or 5 bytes) together with an acknowledge message that is generated by the receiver when a fragment is processed.

Although μC/OS-II comes with a implementation of the TCP/IP protocol stack, the requirements for the envisioned UAV platform open the door to the usage of a simpler and less resource demanding network protocol. So far, only point-to-point communication is required so it is only needed a reliable mechanism to fragment and reassemble application packets, which are likely not to fit into the physical MTU, leaving out all aspects regarding the routing of packets.

A. IceC and application layers

From the application's point of view, the embedded software developer might make use of the *reliable send* and *reliable receive* primitives which are part of the API exposed by the reliable network layer. However, in order to ease the integration and access of functionality from ground to UAV and vice verse, the UAV programmer is provided with the same system abstractions, work-flow and tools as the ones already available

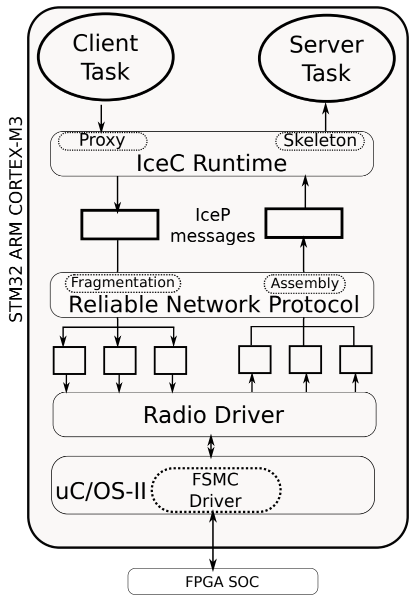

Fig. 2. Functional decomposition of the developed layered middleware infrastructure.

```
module Autopilot {
  enum TypeCode {
    boolT, intT, uintT, floatT, timevalT, byteseqT, stringT
  };
  [...]
  sequence<byte> ByteSeq;
  struct Param {
      ParamId id;
      TypeCode type;
      ByteSeq data;
  };
  sequence<Param> ParamSeq;

  [":hw"] interface HwProcessingEngine {
    void setParameterA(short value);
    void startProcessing(void);
    void getResult(ResultType *res);
  };

  interface Vehicle {
    bool isAlive(short sessionId);
  };

  interface Station {
    void vehicleState(Vehicle* uav, ParamSeq params);
  };
};
```

Fig. 3. Example of Slice specification for IceC.

in communication middlewares for networked systems. As a result, no special training nor skills are needed, leveraging previous background on distributed system application development.

The embedded IceC [1] layer is the key element to make the magic happen. IceC is a lightweight implementation of IceP, the standard protocol of ZeroC Ice middleware[2]. Therefore, our UAV embedded platform is fully compatible with other ZeroC compliant systems.

The only requirement is to reach an agreement on the definition of the services to be implemented. Since ZeroC Ice features an object-oriented model, the service declaration consists in a list of method and user-defined data type definitions. To this end, the developer uses a straightforward *Interface Definition Language* called Slice.

As an example, Figure 3 shows a fragment of a simplified Slice definition of the communication interface used between the Ground Station (*Station* interface) and the UAV (*Vehicle* and *HwProcessingEngine* interface). When it comes to the specification of functionality which is going to be implemented as a hardware module, the interface specification must be tagged with the *[":hw"]* meta-data directive.

At this point, the Slice definition is processed by a home-made interface compiler which generates a set of functions providing the high-level, object-oriented stylish API to the application programmer. This functions or communication stubs (namely, *proxies* and *skeletons*) interact both with the client, server tasks and the IceC runtime so as to progressively translate the invocations into Ice Protocol messages to be sent through the radio interface.

[1]IceC source code is available at: `https://bitbucket.org/arco_group/icec`

[2]http://www.zeroc.com

V. Deeply embedded middleware

This section aims to provide an insight into the architecture and implementation details of the IceC engine. The IceC runtime layer is a minimal realization of the ZeroC Ice middleware, optimized for our UAV platform needs. In order to keep the compatibility with ZeroC Ice PC-based applications (the target platform for the Ground Station logic) and maintain the same philosophy and work-flow in the UAV side, several extensions and engineering works has had to be carried out.

First and foremost, the development of the main architectural components of the commercial version following a low-footprint policy. The layer *IceC* comprises the core elements such as Communicator, Object Adapter, Servant Management, Connections and other helper functions. As a result, a complete static ad-hoc implementation of the principal ZeroC features has been obtained using ANSI C (C90 standard) which allows:

- An impressive reduction of the memory requirements.
- Being supported by Keil compiler for ARM, the IDE used to develop the embedded software.
- The resulting application to be potentially certificated when real-time restrictions apply.

Second, the implementation of a transport specific configurable plugin (PC and UAV) for the xbee connection: the *xbee-at endpoint*. This was necessary due to the following reasons: (a) Xbee is the choice for UAV-Ground Station radio link; and (b) is a necessary step in order to make it usable conventional ZeroC Ice vendor-libraries, tools and C++ bindings over GNU/Linux.

And last but not least, the definition of a mapping of the ZeroC Ice Protocol for ANSI C language which did not exist before. This step comprises a set of rules and languages equivalences to transform Slice grammar and semantics into C structures. As a starting point, CORBA C language Mapping Specification [14] and conventions used in the Ice C++

mapping were taken into account. The result of all this work is a fully new language binding together with a interface compiler which generates the middleware *stubs* responsible for the *marshalling* and *unmarshalling* processes.

As mentioned in the previous section, these communication stubs depend upon the Slice definition and they are produced specifically for an application in an automatic way.

```
static void Autopilot_Vehicle_vehicleState(
                    Autopilot_StationPrxPtr this,
                    Autopilot_VehiclePrxPtr uav,
                    Autopilot_ParamSeq params){
  Ptr_check(this);

  Ice_ObjectPrx_connect(this);
  Ice_OutputStreamPtr os = &(this->stream);

  /* request header */
  Ice_OutputStream_writeBlob(os, Ice_header,
                            sizeof(Ice_header));
  Ice_OutputStream_writeIdentity(os, this->identity);
  Ice_OutputStream_writeString(os, "vehicleState");
  [...]

  /* encapsulated params */
  Ice_OutputStream_writeShort(os, sessionId);
  Ice_OutputStream_writePrx(os, uav);
  Autopilot_ParamSeq_writeToOutputStream(&(params), os);
  Ice_OutputStream_setMessageSize(os);

  Ice_ObjectPrx_send(this);
}
```

Fig. 4. IceC generated code for a proxy.

Figure 4 shows and example of the generated ANSI C code for the *vehicleState* method defined in Figure 3, Section IV-A. This function is a proxy to the actual functionality implemented as a server in the Ground Station. The client task, which runs in the UAV, only needs to call the procedure as it is defined by its signature.

By means of a set of utility functions implemented in the core of IceC middleware, the different parts of an IceP messages are generated and all the parameters serialized and stored in memory through the *Ice_OutputStream* object. This object is a high-level abstraction of the memory write *buffers*, where the IceP messages are temporally held before they are sent. For each user data type defined in Slice (i.e. *ParamSeq*) a pair of functions are also generated which knows how to marshall and unmarshall such data type, according to the Slice (*Autopilot_ParamSeq_writeToOutputStream*) definition. Finally, the built message is sent out to the network using the reliable send primitive provided by the network layer.

Client-server communication between UAV and ground station is performed by referring to remote objects. Coding in the UAV must be programmed in ANSI C (for μC-OSII) but for the ground station we have the flexibility to use any programming language supported by ZeroC Ice.

Figure 5 shows the code for the server running at the UAV platform (upper half) and the code of the corresponding client executing on a standard PC written in Python (lower side). The objects that we can be identified are: (a) the *communicator* is part of the middleware core library and represents the object broker in ZeroC Ice parlance; (b)the PROXY and HWOBJECT are a local references of remote objects. In the first lines of the program, the necessary declarations are made and configuration and initialization of the XBee modem takes place as well. Following, the *Ice_Communicator* is created, setting "xbee-at" as the *endpoint* to use[3]. Finally, the objects and references to objects needed by the application are got and the data structures containing the parameters to send are filled out before invoking the SETPARAMETERA remote method.

```
void Autopilot_HwProcessingEngine_setParam(short value){
  //servant goes here, registers acces through FSMC iface
  FSMC_write(...)
  [...]
}
void Autopilot_HwProcessingEngine_startProcessing(void){
  //servant goes here, registers acces through FSMC iface
  FSMC_write(...)
  [...]
}
void appMainTask (void) {
  XBeeInitTypeDef xbeeConf = {XBEE_BD_115200,0x0000,0x0000};
  Ice_Communicator  ic;
  Ice_ObjectAdapter adapter;
  Autopilot_HwProcessingEngine servant;

  xbeeInit(&xbeeConf);
  Ice_initialize(&ic);
  XBeeATEndpoint_init(&ic, "xbee-at");

  Ice_Communicator_createObjectAdapterWithEndpoints(&ic,
       "Adapter", "xbee-at", &adapter);
  Ice_ObjectAdapter_activate(&adapter);
  Autopilot_HwProcessingEngine_init(&servant);
  Ice_ObjectAdapter_add(&adapter,
                       (Ice_ObjectPtr)&servant,
                       "hwengine1");

  Ice_Communicator_waitForShutdown(&ic);
  [...]
}
```

```
class Client(Ice.Application):
  def run(self, args):

    ic    = self.communicator()
    proxy = ic.stringToProxy('hwengine1 -o:xbee-at')
    hwobject  = Autopilot.HwProcessEnginePrx.uncheckCast(proxy)

    paramValue = 0

    hwobject.setParameterA(paramValue)
    [...]
```

Fig. 5. UAV embedded server (ANSI C for μC-OSII) and Python client (ground station)

In this scenario, the server requires of the ICE_COMMUNICATOR object (already explained), the ADAPTER and the SERVANT. The ADAPTER provides visibility to the objects registered by the server, exposing their functionality to the rest of the network. A SERVANT is the piece of code which actually implements the functionality of the interface.

In this example, the reader can identify the interface *Hw-ProcessingEngine* which corresponds to an simplified example of a possible hardware object running on the FPGA platform. For each method declared in the Slice file for such interface

[3]XBee-AT endpoint implementation is available at https://bitbucket.org/arco_group/xbee-at-endpoints/

a function is generated which represents the API to be used from the ground station to, for example, setting up the algorithm parameters (i.e. *setParameterA*) or start the accelerating function.

VI. Resource usage

The implementation of the UAV embedded platform described in the preceding sections has been carried out focusing in size optimizations.

TABLE I
Synthesis results for the FPGA SoC.

Resource	Total amount
Flip Flops	1537 (8%)
LUTs	1728 (18%)
Slices	772 (33%)

Table I summarizes the total amount of FPGA resources used by a basic configuration of the UAV SoC. It is only considered the FSMC bridge, the Memory Controller and the Radio IP. More than two thirds of the reconfigurable logic area is available for user application purposes. The results have been obtained with the ISE Design Suite 11.3 from Xilinx, targeting the XC6SLX16 Spartan-6 chip. The design runs at a clock frequency of 100Mhz but the FSMC interface is limited by the Xynergy board to 25Mhz which represents a serious bottleneck.

TABLE II
Size in bytes of the different software layers.

Code Size	Subsystem
1116	FSMC Driver
1795	Radio Driver
1307	Network Layer
2622	IceC

Concerning the footprint in memory of the developed software stack, the results shown in table II confirm the high compactness of the solution with a total size of approximately just 7 KB. The generation of the object code has been done using the integrated compiler from the μVision IDE from Keil with the highest optimization level and disabling the debugging symbols.

It has not been considered the inclusion of the memory overhead due to the IceC communication stubs. Since these pieces of generated software are application dependant, there is no impartial way to measure the impact in general terms. For the same reason, no timing analysis has been taken into account in this work.

VII. Conclusion

HW-SW hybrid boards are emerging platforms for UAV applications, however there is a lack of solutions/tools for developing on these type of platforms.

In this paper, it has been dissected the architecture and implementation details of a tailored middleware infrastructure for hybrid embedded system. With extremely little resources, it is assured the compatibility with standard ZeroC Ice subsystems. No special tools, languages or design skills are required in order to integrate HW or SW components with our solution into an object-oriented networked system. The application use case, ground to UAV communication, introduces additional challenges which are faced using a combination of hardware and software.

From an industrial point of view, apart from certification issues, our current efforts are devoted to extend the number of heterogeneous embedded platforms and legacy middlewares supported which can integrate in IceC.

Acknowledgment

This work has been partly funded by the Spanish Ministry of Economy and Competitiveness under project REBECCA (TEC2014-58036-C4-1-R) and by the Regional Government of Castilla-La Mancha under project SAND (PEII-2014-046-P).

References

[1] AIA, "Unmanned aircraft systems: Perceptions & potential," Aerospace Industries Association, Tech. Rep., 2013.

[2] J. Barcelo-Ordinas, J. Chanet *et al.*, "A survey of wireless sensor technologies applied to precision agriculture," in *Precision agriculture 13*, J. Stafford, Ed. Wageningen Academic Publishers, 2013, pp. 801–808.

[3] F. Mohammed, A. Idries *et al.*, "Opportunities and challenges of using uavs for dubai smart city," in *New Technologies, Mobility and Security (NTMS), 2014 6th International Conference on*, March 2014, pp. 1–4.

[4] P. Doherty and P. Rudol, "A uav search and rescue scenario with human body detection and geolocalization," in *AI 2007: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, M. Orgun and J. Thornton, Eds. Springer Berlin Heidelberg, 2007, vol. 4830, pp. 1–13.

[5] J. Paunicka, B. Mendel, and D. Corman, "The ocp - an open middleware solution for embedded systems," in *American Control Conference, 2001. Proceedings of the 2001*, vol. 5, June 2001, pp. 3445–3450 vol.5.

[6] J. Paunicka, D. Corman, and B. Mendel, "A corba-based middleware solution for uavs," in *Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on*, May 2001, pp. 261–267.

[7] J. Kwon and S. Hailes, "Mirea: Component-based middleware for reconfigurable, embedded control applications," in *Intelligent Control (ISIC), 2010 IEEE International Symposium on*, Sept 2010, pp. 2385–2390.

[8] P. Royo, J. Lpez *et al.*, "Service abstraction layer for uav flexible application development," in *Proceedings of the 46th AIAA Aerospace Sciences Meeting and Exhibit*. Reno, Nevada (USA): AIAA, Jan 2008.

[9] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *Proceedings of the 2003 IEEE Conference on Military Communications - Volume I*, ser. MILCOM'03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 242–247.

[10] M. Tortonesi, C. Stefanelli *et al.*, "Multiple-uav coordination and communications in tactical edge networks," *Communications Magazine, IEEE*, vol. 50, no. 10, pp. 48–55, October 2012.

[11] E. Santamaria, P. Royo *et al.*, "Increasing uav capabilities through autopilot and flight plan abstraction," in *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, Oct 2007, pp. 5.B.5–1–5.B.5–10.

[12] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, 2004.

[13] G. Montenegro, N. Kushalnagar *et al.*, "RFC 4944 – transmission of IPv6 packets over IEEE 802.15.4 networks," IETF RFC.

[14] *C Language Mapping Specification*, Object Management Group, June 1999.