Software Architecture for Autonomous Vehicles

Ricardo Shimoda Nakasako
Department of Mechatronic and Mechanical Systems Engineering
Escola Politécnica da Universidade de São Paulo
Rua Prof. Mello Moraes, 2231 – Cidade Universitária, São Paulo – SP – Brasil – CEP 05508-900
ricardo.nakasako@poli.usp.br

Fabio Kawaoka Takase
Department of Mechatronic and Mechanical Systems Engineering
Escola Politécnica da Universidade de São Paulo
Rua Prof. Mello Moraes, 2231 – Cidade Universitária, São Paulo – SP – Brasil – CEP 05508-900
fabio.takase@poli.usp.br

Abstract*. Modern developments have increased the processing capacity of controllers along with the gradual decrease in their price. This has enabled the construction and use of intelligent control elements in embedded applications. These elements, such as smart sensors and actuators, not only can pre-process sensorial data using sophisticated algorithms but also enables the use of multiple communication protocols linking all the existing devices. The use of these control elements leads clearly to the construction of a distributed environment control in which the responsibility for processing environment data is divided between the several parts of the system. Moreover, there can be some redundancy in the data collected as well as the data collected by some sensors can have a greater priority level or must be collected using a shorter update deadline than others. In order to orchestrate this complex communication (which is natural on any distributed computing environment) between the elements of such a control system, the figure of a network connecting all the devices and a Middleware coordinator are evident.*
This article firstly analyzes the current efforts in developing the software architecture used to support these requirements in terms of message coordination and processing. Based on these, a software architecture is then proposed in order to enable the easy integration of new control elements dynamically as well as maintaining the real time response of the control system.

Keywords*: Sensor Networks, Real-Time, AUV, Software Architecture, Embedded*

1. Introduction

Autonomous Underwater Vehicles are mission-critical systems which must be controlled by real-time embedded software. This fact imposes several additional concerns and costs when developing code which controls such a vehicle. Moreover, the design always lead to the implementation of mission specific systems, therefore resulting in poor code reuse and, consequently, in higher costs [Pasetti and Pree, 1999].
In order to address this difficulty, several approaches have been used, from the development of components which enables the creation of modularized architecture which enhances code reuse to the establishment of development frameworks, which makes the reuse of code mandatory [Pasetti and Pree, 1999].
However, as can be seen on [Newman, Hill et al.] the current proposed models do not take advantage on the usage of modern technology and standards such as CAN (Controller Area Network) and the establishment of a sensor network. They also are not prepared to deal with modern sensors which acquire and process complex data (such as images) which, therefore, have a completely different response time and requires the implementation of a completely different control behavior. To end with, the control of such vehicles deals only with static positional data which, in the long run, limits the complexity of missions.
Finally, the existing logical structure is still limited when dealing with a variable number of sensors and actuators and this can compromise the fulfillment of a mission, for example, should in the middle of it one of these elements break (i.e.: stop working) or fail (i.e.: work improperly) [Newman].
The structure of this paper is as it follows: Firstly, the existing software modeling approaches are presented in a way to extract from these models their upsides and downsides. A model which tries to integrate the best characteristics of all the studied approaches with the new technology is then presented.

2. Analysis of an existing model: MOOS

The MOOS (Mission Oriented Operating Suite) framework was created in 2001 as a way to establish a framework upon which mobile robotic software could be developed. It consists of a core database module, called the MOOSDB, which is used as a central repository for reading and writing data, and several base

'framework' classes. These classes provide basic functioning such as logging, navigation and mission control functionalities and interface classes through which hardware specific developers could develop code to communicate with the central database. The MOOS topology is detailed in the following sub-section.

2.1. MOOS topology

The MOOS set of applications consists on two different types of modules, according to their functionality:

- Clients: these modules are the one which really gather data, process it and control the movement of the vehicle. They are so called 'clients' because their only data source (either for writing or reading) is the central module, the database (MOOSDB).
- Database (MOOSDB): this is the central module in the MOOS system. It acts as a unique common data source which must be used by all MOOS modules to store and read data.

Among the existing clients, there are two different types of applications:

- Processes: these are applications whose main objective is pure data processing, therefore not dealing directly with the vehicle's sensors and actuators. Examples of this kind of applications are: pHelm, which deals with determining the most suitable actuation commands (but not with an actuator per se); pNav: which takes data from sensors and provides an estimate of the vehicle's position and velocity and pLogger, which logs all the events and navigation decisions taken.
- Interface: these are applications which are built to deal with sensors and actuators in a organized way, either transforming the logical actuation commands into real actuator commands or gathering only relevant sensor data (blocking most of the repeated data and getting only the changes).

In order to implement a generic communication between any client and the MOOSDB, a common interface for data processing, requesting and storing was developed, all of it using a basic communication class (CMOOSCommClient) which would handle the operation in a TCP/IP network which connects all the clients to the MOOSDB.
This topology is illustrated in the following picture:

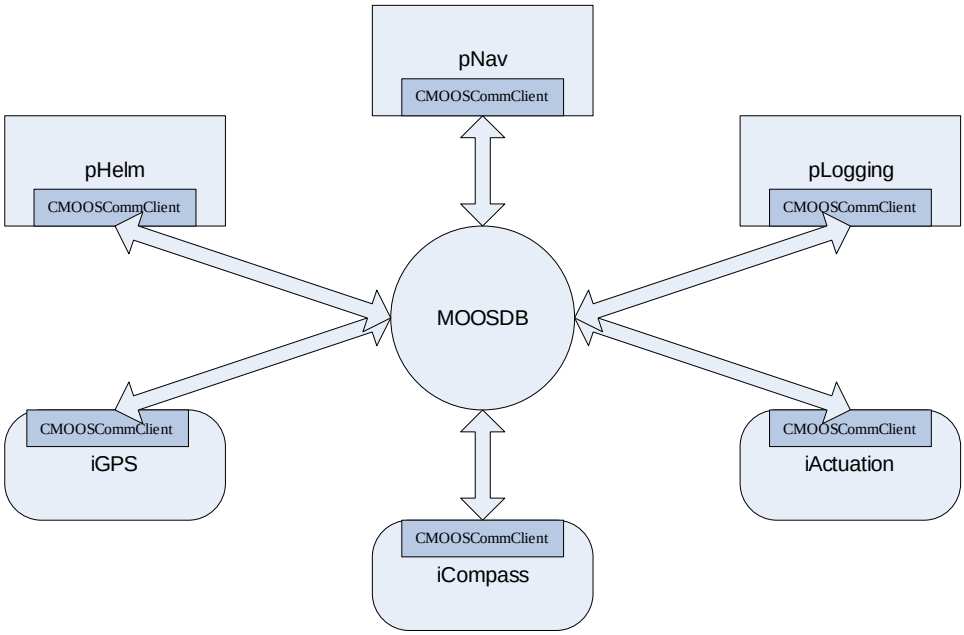

Figure 1. MOOS Topology model

The model presented is a simplified one. In the real model, each different sensor would have a communication link to the MOOSDB and would be internally classified into one of the following categories: Positional, Depth, Rotation and Velocity. In this model, also, all actuators would be linked only to the iActuation interface. This interface, in the proposed model, has internally only three different types

of actuators: Elevator, which regulates the depth of the vehicle, Thrust, which is the general actuator for the vehicle's velocity and Rudder, which is the general actuator for the vehicle's rotation.

2.2 MOOS model analysis

It is not very hard to see that the MOOS project has a single point of failure, which is its database module (MOOSDB). Therefore, development efforts on optimization and error control had to be fully focused on this module's development.
It is also easy to see that this centralized development had to occur in order to centralize the actuation decisions in only one part of the application, so that every sensor data had to go through an analysis in order to determine the vehicle's current location (pNav process) and how it impacted the mission's objectives before it could result into a actuator's order (pHelm). This process might not be so quick when dealing with sensors whose data needs a quick response, such as when eminent collision is detected.
From the communication standpoint, the TCP/IP protocol proves not to be the best when dealing with control data. This set of protocols is not prepared to deal with priority messaging (TCP implements reliability but not different levels of priority between messages) and support for this kind of communication can be processor consuming.
Finally, from the mission standpoint, it is easy to see that the generalization of how sensors and actuators affect the system does not correspond to the necessary complexity. Missions are absolute location related and, therefore, their definition does not include the need to, eventually, lock the underwater vehicle in a position relative to other underwater structure in movement.

3. Proposed solution

The MOOS solution is very complete and works fine when dealing with expected situations and simple missions. However, it does not comprise the level of complexity real-life missions might require (such as maintaining a position relative to another underwater object) and is limited in terms of the data sensors can give and actuators can process (a mechanical arm would be useless). Moreover, it does not implement an emergency way in which sensor data can directly interfere in an actuator's behavior.
In order to address these necessities, another software architecture is needed as well as another communication protocol between controllers, sensors and actuators. These are exposed in the following paper subsections.

3.1. The CAN Protocol

The CAN protocol is a de facto communication protocol standard between sensors and controllers which is widely used to implement distributed control systems in automobiles and in the industry. It specifies a bus topology and implements a CSMA-CA MAC layer protocol. In order to do so, before the message (or, payload) is transmitted in the bus, there must be a frame composed of an 11 bit identifier which is used for priority messaging and collision avoidance. This protocol works in the following way:
Every node, when trying to transmit data, listens to (at most) the first 11 bits in the bus to see whether it really got to send the data or not. Suppose nodes 1, 2 and 3 are trying to transmit data at the same time, node 3 with a higher priority than 1 and 2.

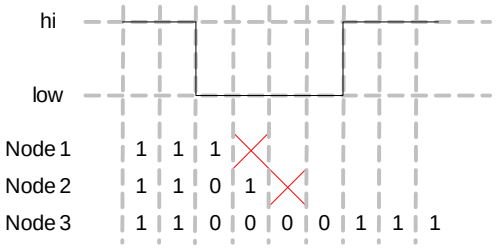

Figure 2 CAN priority implementation

When node 2 and 3 transmit 0 and node 1, 1, the bus goes to low and, as node 1 is listening to the bus, it knows his message is not top priority and stops transmitting data. The same occurs to node 2 when it tries to transmit 1 and the bus keeps in low.
This characteristic is essential when dealing with sensors whose generated data can or not be of extreme importance (such as collision eminence) for the vehicle. This message driven architecture of CAN forms the basis for the proposed network topology and, consequently, software architecture.

3.2. The proposed topology

In order to address the difficulties seen in the MOOS model, the following communication model is proposed. In this model, all the objects are CAN enabled devices. Each of the devices is uniquely identified by a 16 bit header which follows the normal CAN priority header.

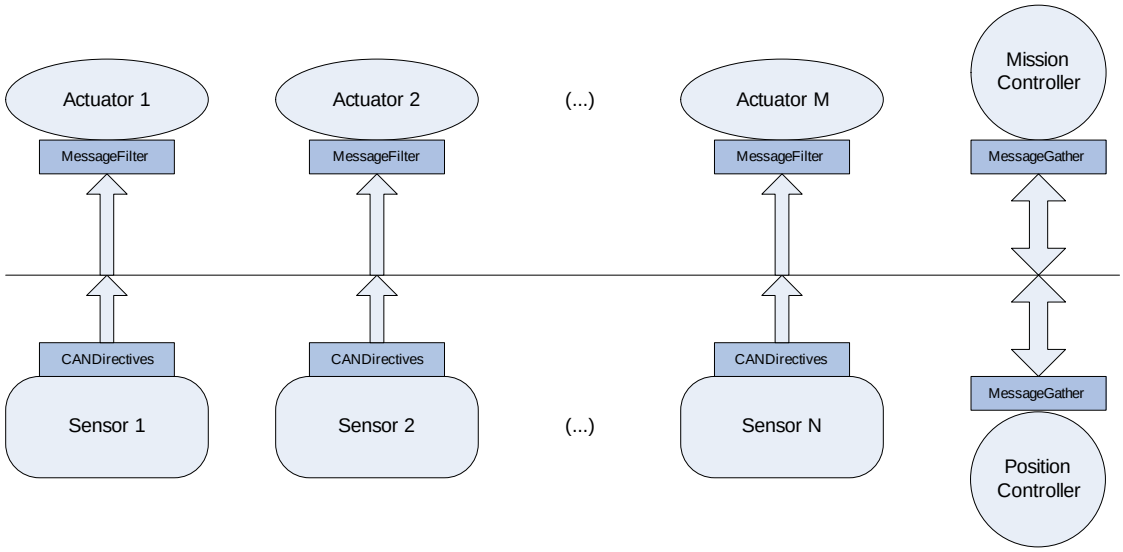

Figure 3. Proposed topology

The Mission Controller acts as a planning agent responsible, therefore, for planning (and, when necessary, re-planning) the necessary steps to take the vehicle to the expected goal position. It is also responsible for examining the existing sensors and calculating the mean level of Certainty of the data presented by the sensors. This data is used in order to establish a policy spread by a high level message throughout the sensors which dictates how CAN priorities must be calculated.
The Position Controller acts as an 'instinct' agent, therefore being responsible for maintain the vehicle's current position.
For each of the Sensors there is an object of the type CANDirectives. This object is responsible for the following actions:

- Verify the level of certainty of the sensor in question, the priority of the data generated by the sensor and translate it into a priority level, according to the policy established by the mission controller.
- Mount the message (priority and node id) and transmit it over the network, using the correct communication class as a basis.

For each of the Actuators there is a MessageFilter object. This object is responsible for the following actions:

- Verify if the message comes directly from a sensor with high priority or from a Mission Controller message (normal priority) or from a Position Controller message (normal priority).
- If the message comes from a sensor, then it implements the reaction agent behavior, inferring directly in the actuator's settings.
- If it is a Mission/Position Controller message (normal priority) it verifies if the next 16 bits corresponds to the actuators id. If so, then the control message is translated to the actuator.

Finally, the Mission Controller is apt to receive all the sensor messages (including emergency ones) which enable it to recalculate the necessary tasks to accomplish a certain mission or even decide for a mission abort action.
Through the use of such a distributed architecture, it is possible to:

- Control and update dynamically the tasks which accomplish a certain mission;
- Make a direct bridge between Sensor and Actuator in emergency cases;
- Enable a priority messaging which enables the use of several Sensors linked to the same bus at the same time.

This topology is made possible by the use, when necessary, of intelligent microcontrollers (such as TINI) which allow the quick implementation of complex data structures and data processing logic directly on hardware [Eisenrach and DeMuth] using a high-level programming language (such as Java).

3.3. Sensor class modeling

Sensors, from the most primitive ones (whose output signal is analog and, thus, must be capured) to the most complex ones (which implements events and numeric output) can have their output processed accordingly and, thus, be modeled into several Sensor classes.
These classes can be abstractly represented by a single sensor interface, which allows generic object serialization to happen in the Topology network. The idea is to create classes which are generic enough to be carried through the network, but with a level of information good enough to be usable by the Mission Controller module. This implementation uses the facade design pattern.
The Sensor interface defines that at least four properties must be implemented by a Sensor-type class:

- Certainty: this defines a level of certainty in the information being passed
- Priority: this defines the level of priority in the information being passed
- Timestamp: this value defines a timestamp which is used by the controller in order to determine if the information passed is still valid or not
- Value: this represents the value which was measured by the sensor itself

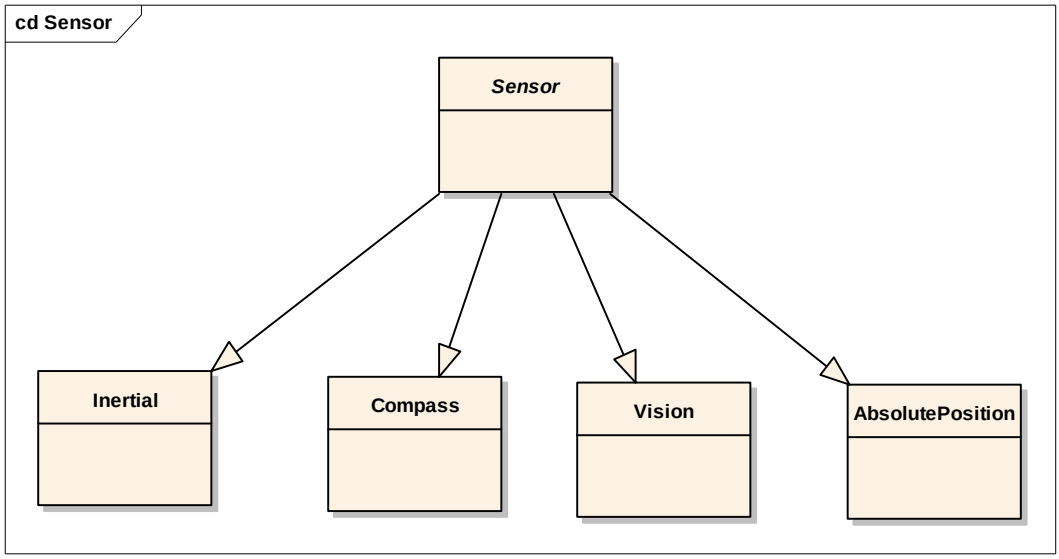

Figure 4. Sensor hierarchy classes

The first three properties can be used in order to calculate the level of priority which will be used when dealing with the Comm class, should it support priority handling in the link level (such as CAN). The third property (Timestamp), though, is essential for the controller in order to know whether the information is reliable or not.
The fourth property is an object (which derives from the derived Sensor classes) which will have to be processed accrdingly by the controller in order to take its actions.

3.4. Layer Communication modelling

After modeling the Sensors, it is necessary to establish a way in which sensors and the controller can communicate (i.e.: How can sensors deliver their measures). In order to do so, a coherent logical structure is made necessary. This logical structure uses the following classes/interfaces:

- Comm: this class represents a generic communication class, should it implement TCP/IP or CAN protocols. It is represented in the original topology as the CANDirectives and the MessageFilter layers.
- Sensor_Skeleton: this class implements the sensor itself (as it was modeled in the previous part of the paper) and is serialized to be transmitted over the network.
- Sensor: this interface is used by both sides of the network to provide the necessary generalization level when transmitting sensor data on the network.
- Sensor_Stub: this class has both a Comm and a Sensor interface implementation and it is used to receive serialized Sensor objects though the network.
- Sensor_Client: this class represents the Sensor itself in the Mission Controller or Actuator layers.

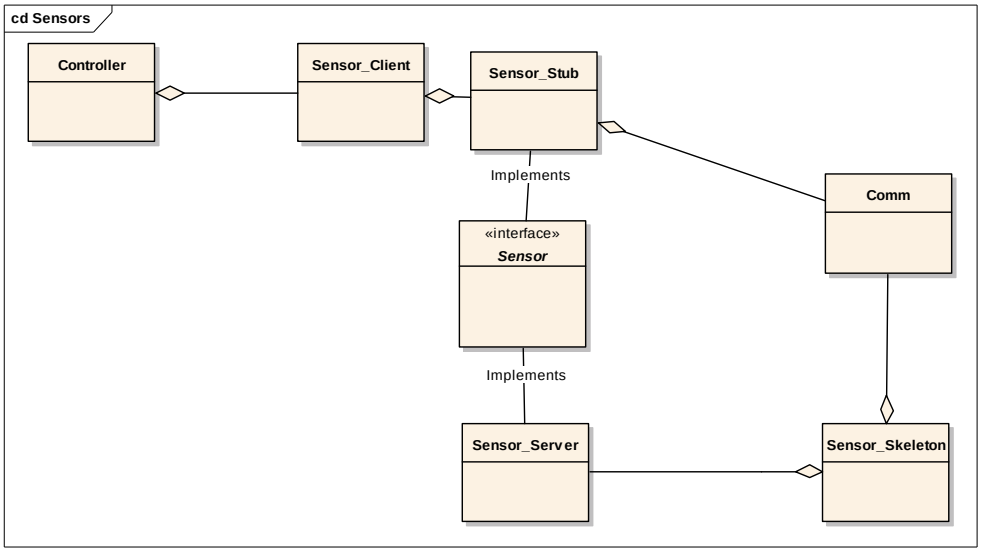

Figure 5. The communication classes

3.5. Implementation

In order to implement this design in a down-to-earth, realistic way, the chosen language was Java. Not only it has several 'real-time' facilities implemented in its virtual machine, but it also has several communications classes already implemented. Moreover, the usage of this language enables the design of hardware interfaces using TINI [Eisembach and DeMuth, 2003].

The Comm class in the previous diagram can be implemented as an extension to the existing CAN class designed to work with TINI. Several issues, though, must be taken care for from the creation of better methods for trying (and retrying) to send messages in general to the correct formation of frames and enhanced object serialization.

Initially the implementation of a primitive set of such classes has been done via the usage of TCP classes in a test environment (Figure 6). Further work will support the use of CAN and CAN communication libraries.

```
/*
Sensor_Skeleton class
*/

import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;

public class Sensor_Skeleton extends Thread {
   SensorServidor meuServidor;
   public Sensor_Skeleton(SensorServidor servidor) {
     this.meuServidor = servidor;
   }
   public void run(){
    try {
    ServerSocket serverSocket = new ServerSocket (9000);
    Socket socket = serverSocket.accept();

while (socket!=null){
      ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
      String metodo = (String)inStream.readObject();
      if (metodo.equals("Position")){
           int Position = meuServidor.getPosition();
      ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
      outStream.writeInt(Position);      outStream.flush();
      } else if (metodo.equals("Certainty")){
           String Certainty = meuServidor.getCertainty();
      ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
      outStream.writeObject(Certainty);   outStream.flush();
      }
   }
  } catch (Throwable t) {t.printStackTrace(); System.exit(0);}
}
public static void main (String[] args) {
    SensorServidor sensor = new SensorServidor("Sensor Visao",2);
    Sensor_Skeleton skel = new Sensor_Skeleton(aluno);
         skel.start();
    }
}

/*
Sensor_Stub class
*/
import java.io.*;
import java.net.Socket;

public class Sensor_Stub implements Sensor {
   Socket socket;
   public Sensor_Stub() throws Throwable {
      socket = new Socket("localhost",9000);
   }
   public int getPosition() throws Throwable{
    ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
        outStream.writeObject("Position");
        outStream.flush();
        ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
        return inStream.readInt();
   }
public String getCertainty() throws Throwable{
    ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
        outStream.writeObject("Certainty");
        outStream.flush();
        ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
        return (String) inStream.readObject();
   }
}
```

```
/*
Sensor interface
*/

public interface Sensor {
   public int getPosition() throws Throwable;
   public String getCertainty() throws Throwable;
}


/*
Sensor_Server class
*/

public class Sensor_Server implements Sensor {
   int Position;
   String Certainty;
   public SensorServidor(String no, int id) {
       this.Position = id;
       this.Certainty = no;
   }
   public int getPosition(){
    return Position;
   }
   public String getCertainty(){
    return Certainty;
   }
}

 /*
 Sensor_Client class
 */

 public class Sensor_Client {
    public static void main (String [] args) {
     try{
         Sensor sensor = new Sensor_Stub();
                 System.out.println(
                     "The sensor announces Position " +
                     sensor.getPosition() +
                     " with certainty " +
                     sensor.getCertainty());
     } catch (Throwable t) { t.printStackTrace();}
    }
 }
```

Figure 6. Partial implementation of communication classes

3.6. Future work

After having modeled the Sensors and how they communicate with the controller, it is necessary to model the Actuators in a similar, generic way. It is obviously necessary, though, to implement a 'keep alive' mechanism (which may use the exchange of statuses) so that the Actuator can, initially, announce its behavior/effect when moving the AUV and, from time to time, literally announce its own state (full, degraded performance or broken). Moreover, it must, when taking a reflective action (such as when the message transmitted by the sensor is really urgent), communicate the deed to the Mission Controller.
Secondly, the Mission Controller must be implemented in such a generic way which uses all the information given by both sensors and actuators in order to either calculate a new strategy which accomplishes the objective or decides for complete mission failure.
Finally, implementation of the communication classes must be done and a real environment must be used in order to produce results. A coherent test environment must be also set, using mock objects representing sensors and actuators in order to test the architecture implementation.

4. References

Etschberger, I.K., 2000, "CAN-Based Higher Layer Protocols and Profiles", IXXAT Automation GmbH

Hill, J., Szewczyk, R., Woo, A., Hollar, S. Culler, D. Pister, K., 2000, “System Architecture Directions for Networked Sensors”, Proceedings of the IEEE, vol. 91. no.7 pages 1002 - 1022

Newman, P.M., 2000, “MOOS – Mission Oriented Operating Suite”, PhD Thesis, Department of Ocean Engineering, Massachusetts Institute of Technology.

Pasetti A. and Pree, W.,1999, “A Reusable Architecture for Satellite Control Software”, Dept. of Computer Science, University of Constance, Germany.

Pasetti A. and Pree, W., 1999, “The Component Software Challenge for Real-Time Systems”, Proceedings of the 1st International Workshop on Real-Time Mission-Critical Systems; 30 Nov -1 Dec, 199, Scottsdale, AZ, USA.

Richardson, P. Sieh, L. Haniak, P., 2001, “A Real-Time Control Network Protocol for Embedded Systems Using Controller Area Network (CAN)”, IEEE Electronics and Information Technology Conference, Rochester, MI

Eisenreich, D. and DeMuth, B., 2003, “Designing Embedded Internet Devices”, Newes

5. Responsibility Notice

The author is the only responsible for the printed material included in this paper.