VERSITA
Applied Computer Systems
doi: 10.2478/acss-2013-0007
2013/ 14

Architecture and .NET Implementation of Multi-Robot Management System

Aleksis Liekna[1], Agris Nikitenko[2], [1-2]*Riga Technical University, Faculty of Computer Science and Information Technology, Institute of Applied Computer Systems*

***Abstract* – In this paper we propose architecture for a multi-robot management system, ways of identifying the implementation challenges and analyzing how to address these challenges by using out-of-the-box features provided by .NET framework. We make a departure from using agent or robot specific development tools in order to take a fresh look at one of the industry leading general purpose development platforms that is .NET framework to analyze its applicability to multi-robot system development. For verification of the proposed ideas in practice, a multi-robot system is implemented using physical iRobot Roomba robots as the robot platform.**

Keywords – multi-robot systems, .NET framework, implementation challenges

I. Introduction

Numerous architectures and frameworks exist for developing multi-agent and multi-robot systems (e.g. AgentFactory [1], Player-Stage [2] and JADE [3], to name a few) that are created to address robot or agent specific concepts such as behaviors and communication that are not covered by general purpose development techniques. In order to apply those in practice, the developer has to study available frameworks and architectures to choose the one or a combination of some suited for solving the problem at hand. One has to acquire deep knowledge of the architectural and implementation details of the chosen solution to be able to implement it in practice. These aspects may cause difficulties for developers that are used to work with general purpose development tools and are not familiar with particular research field. It is also worth noting that abstractions like agents and multi-agent systems look very good on paper, but "..the corresponding implementations do not often express the same properties" [4].

This is why we choose to take a different path. We propose the use of general purpose development platform to create a multi-robot management system. We define an ad hoc architecture for multi-robot management system applicable to a set of problems, identify implementation challenges and analyze how to overcome those challenges using general purpose development platform. In this paper we focus on .NET framework as the development platform.

.NET framework has progressed rapidly over the past couple of years and a number of researchers (i.e. [5] and [6], [7]) have found it a suitable choice for implementing multi-robot systems. There is also a project named AgentService [4] – a .NET based multi-agent development framework, compliant with FIPA standards (see [8] for details on FIPA). This encourages us to analyze how the out-of-the box features of .NET framework may aid on overcoming several implementation challenges.

The rest of this paper is organized as follows. In Chapter II a problem domain is described as well as description of the proposed architecture is given. Chapter III defines implementation challenges of the proposed architecture. In Chapter IV the analysis is provided of how these implementation challenges can be solved by using out-of-the-box features of .NET framework. Chapter V describes implemented prototype. Chapter VI provides summary and evaluation of the results.

II. Multi-robot management system

A. Problem domain

The problem under consideration is joining multiple autonomous robots in a multi-robot system to accomplish the tasks that are beyond the capabilities of a single robot. Particular task can be defined as an area coverage or sweeping [9] task where multiple robots are required to cover a particular area, however the focus is shifted from the details of the sweeping algorithm to the overall system architecture and implementation. The following constraints are assumed to be held: a) capabilities of a single robot are not enough to cover the whole area; b) each robot is capable of autonomously covering some part of the area; c) a task can be decomposed into subtasks in such a way that each subtask can be accomplished by a single robot. An example of such a task is cleaning a large area warehouse using vacuum-cleaning robots that are designed to clean a hotel room – no robot is capable of cleaning the whole warehouse in a reasonable time, but the warehouse can be divided into smaller areas each of which can be cleaned by a single robot.

The resulting multi-robot system must accept tasks from external source (be it a user or another system), decompose them into subtasks and allocate those subtasks to the corresponding robots. This poses the need for a management system capable of providing multi-robot system with the abovementioned functionality.

B. Overall system architecture

According to the problem domain, a partially centralized architectural solution is chosen. The basic idea is partially similar with the ideas represented in [10], where a server (the central point) is present that is responsible for accepting user input and high-level task allocation (or scheduling).

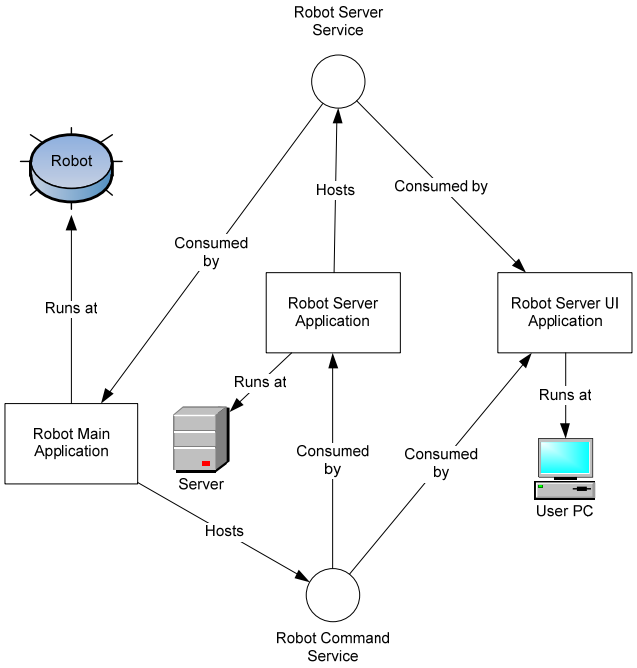

Fig. 1. Proposed architecture

When the task is allocated to a particular robot, it executes the task autonomously and communicates with the server for status-report only. In contrast, fully decentralized approaches (like [11]) respect each robot to independently select its goal. Partially centralized solution is considered to be more straightforward in our case, because of external nature of the task source. In case of a fully decentralized approach this means that each robot must be capable of accepting tasks from external source and at least initiate a distributed task allocation mechanism, which is not required in our case.

The proposed architecture consists of several applications and services as depicted in Fig. 1.

Robot Main Application runs onboard robotic system. It is responsible for the low-level control of the robot and serves as individual localization and mapping source.

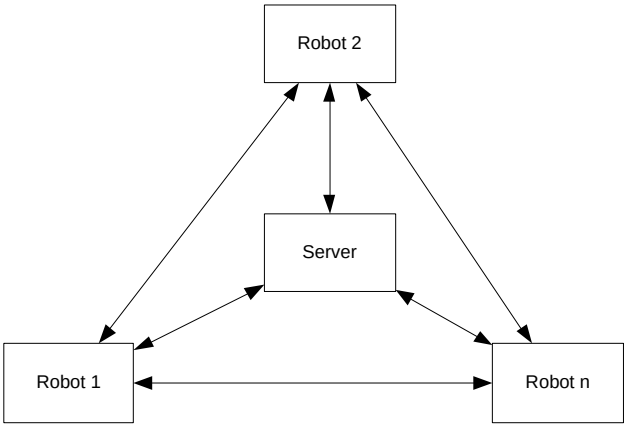

Fig. 2. Robot interaction

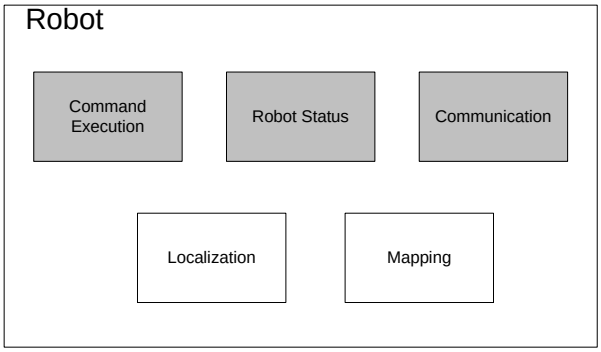

Fig. 3. Robot architecture

It also hosts Robot Command Service that provides interface to the robot for the rest of the system. Robot Command Service is used by Robot UI application, Server UI Application and Robot Server Application.

Robot Server Application runs at Server and serves as a central point of the management system. It monitors active robots in the system and is responsible for task allocation and scheduling among robots. It also hosts Robot Server Service that provides interface for task assignment and a list of active robots. Robot Server Service is used by Robot Server UI Application and Robot Main Application.

User can run Robot Server UI Application on his/her PC to acquire a complete system overview and issue tasks to the system as whole, as well as send individual commands to active robots. Robot Server UI Application maintains a connection to Robot Server Service for a list of active robots. It can then individually connect to some (or all) of them for sending individual commands (used mainly for debugging purposes).

In a typical usage scenario Robot Server UI Application is used for specifying system-wide tasks that are sent to Robot Server, which processes (e.g. decomposes) them and sends to individual robots.

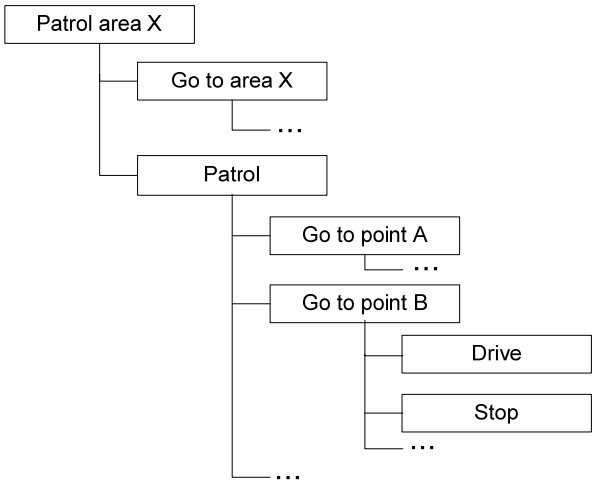

Fig. 4. Example of hierarchical robot command structure

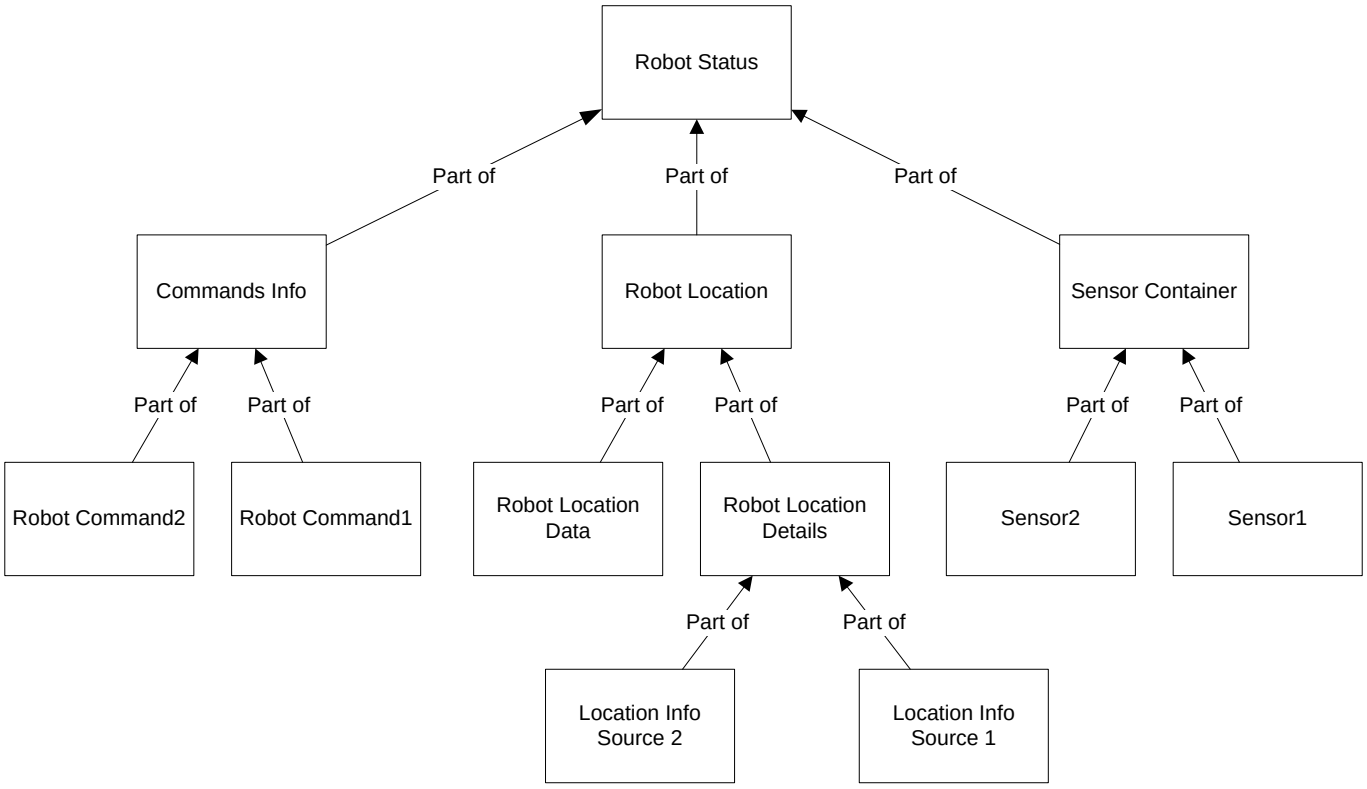

Fig. 5. Robot status data structure

Note that in Fig. 1 only one robot is shown for the sake of simplicity. The proposed architecture supports multiple robot instances capable of communication with each-other by consuming their respective Robot Command Services (see Fig. 2). Robot server is used for service discovery, since it maintains a list of active robots in the system.

C. Robot architecture

Robot architecture consists of 5 general modules: command execution, robot status, communication, localization and mapping (see Fig. 3). In this paper we address only the first three of these (darker blocks in Fig. 3), since the analysis of localization and mapping is beyond the scope of this paper.

Command execution module includes command manager responsible for accepting (or rejecting) new commands as they arrive. Commands are processed using FI-FO queue and only one command is allowed to be executed at any given time. When execution of the current command has been finished or aborted for some reason, the next command in queue is executed. Command manager is also able to abort certain (or all of the) commands if asked to do so.

Commands can be hierarchical – each command can contain its own command manager with own command queue. This allows creation of more complex commands that are composed of simpler commands. For example, command “patrol area X” can contain sub-commands “go to area X” and “patrol”, where command “patrol” can contain several “go to point” sub-commands, each containing several “drive” and “stop driving” commands, etc. (see Fig. 4)

To be able to react to certain situations (e.g. to avoid obstacles), each command can have one or more conditional actions. An example of such conditional action of command “drive” is “stop when bump occurs”.

Conditional actions together with hierarchical commands provide the basis for defining robot behavior in a modular manner. Robot behavior in this context can be defined as a set of commands to be executed. More complex commands can be composed of simper commands, thus providing the necessary means of modularity.

Robot status module is responsible for populating robot status data structure with the appropriate data. Examples of such data are robot sensor readings and the command currently being executed. The data structure is hierarchical in the sense that there is one upper-level object named “Robot Status”, which contains other lower-level objects representing the corresponding details (see Fig. 5). Robot status data structure is not just a placeholder for robot status data – all of the corresponding nested objects populate their appropriate fields in real-time providing the necessary means of abstraction. For instance, object “Sensor Container” contains information about robot sensor readings. When a new reading of sensor data is available, this information is updated internally within the “Sensor Container” object.

Communication module is responsible for providing an interface to the rest of the system. This includes a Robot Command Service host and also a Robot Server Service client.

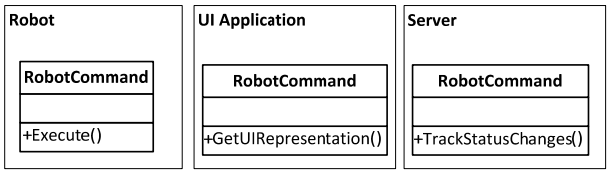

Fig. 6. Different behavior for RobotCommand at different parts of the system

III. Implementation challenges

There are several implementation challenges associated with the proposed architecture. One of the challenges in implementing distributed software architecture is to maintain the data structure that could be reused in several parts of the software. This is also the case with the proposed architecture.

Robot command can be viewed as an example – it is created at the UI side or robot server side and then sent to the robot, which executes the command. The status of the command is depicted on the UI side as well as monitored in the robot server that may have reasons to keep track of command status.

This means that different behavior and different data is associated with the command in each part of the system, e.g. on a robot side the command has a method named "Execute()" while on the UI side it has a method for obtaining its graphical representation (see Fig. 6). There are several ways to achieve this functionality. The first one is to create a common data structure library containing all the concepts and implementing all data and behaviors the concepts need in any part of the system. This results in data structures that are overwhelmed with data fields and operations and cannot be used without a context (e.g. a robot command that references infrared sensor at the robot side cannot be used without a reference to that sensor in the UI side).

Another option is to create a common data structure where each concept contains only the common data fields and no methods. Then there are two possible solutions: either to create a specification of each concept in each part of the system using inheritance or to create a wrapper class that specifies behavior and additional data for each concept in each part of the system. The former solution is faced with type casting issues because of the need to always create (or cast to) the correct type. This problem becomes more apparent when dealing with nested lists of concepts that also need to be specified. For example, imagine concept "robot" that has a list of "sensors". Now imagine casting items in that list from and to the accurate type every time you need to use it.

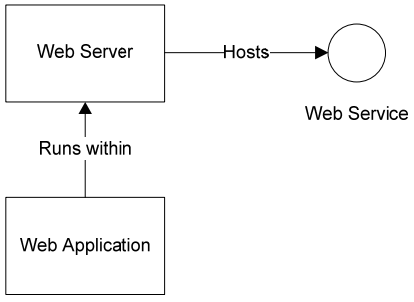

Fig. 7. Web service architecture

The latter solution including wrapper classes can easily become a "maintenance nightmare" in a long nested concept chain, where there is a need to maintain a wrapper around each element of the nested element of the nested sub-element and so on.

For example, imagine concept "robot status" that includes "command information" that includes "command history" that includes a list of "robot commands", each entry of which includes a list of "status history" and so on.

The next implementation challenge is dynamic command execution. Since robots are entities that operate in a dynamic world, robot commands may have some starting or ending conditions, e.g. to stop driving when robot hits the wall or to seek home base when battery charge is dropping below threshold. Such condition can be expressed using predicates. For example, the sentence "Robot R1 has to stop if it senses a bump with bumper sensors B1 and B2" can be encoded as "$Bump(B1) \wedge Bump(B2) \longrightarrow Stop(R1)$". The key is to provide robot client (i.e. user interface) with the ability to dynamically compose such expressions and send them to robot, thus offering the possibilities to dynamically create previously undefined command execution conditions.

This looks simple enough on paper, but the implementation usually consists of complex logic parsers or behavior-based approach that allows for hard-coded predefined behaviors that are stored on the robot side, thus giving a choice of already defined values. Robot world is not static and it is common practice to change robot sensors or adapt robot platforms for different tasks. So if one wants to send a command with the condition that has not been defined yet (i.e. you recently added a laser range finder sensor to the robot and want to send the robot a command to stop if an obstacle is within 1m)? The answer is simple – it is not possible. Even with a well-designed complex logic parser, one can only use the operands that are predefined – adding a new (type of) sensor requires rewriting and recompiling the parser. The task may be easier with a well-defined configuration file, but it still remains an issue.

The third implementation challenge is the communication infrastructure. The proposed architecture implies services as the basis for communication but it does not specify them. A service can range from a TCP (Transmission Control Protocol) endpoint used to send and receive strings of bytes that are then interpreted in the appropriate way to a web service that defines high-level operations an endpoint can perform. Nowadays the industry is leading towards web services and service oriented architectures [12]. The main constraint web services imply is that they must be hosted using some kind of web-server. A web-server is usually an application that hosts web related content (such as web-pages and web services) placed in the specially configured virtual directories. This means that the application willing to support web-service interface must be integrated within a specially designed web-application that starts and stops together with the web-server (see Fig. 7). If the web-server is malfunctioning for some reasons, the application will malfunction as well.

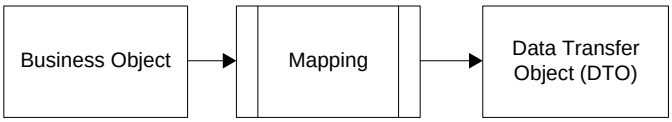

Fig. 8. Mapping business objects to data transfer objects

Such situation makes nearly impossible design of standalone applications that provide web-service interfaces. Although modern programming languages (such as Java and C#) provide the possibility to host http server in a standalone application, the tools provided are only viable for very simple or proof-of-concept projects. Trying to implement a mechanism for serving thousands of http requests per second is like reinventing the wheel and will most likely end up with a system that works "sometimes", not to mention that doing so is cumbersome and not straightforward. It means the developer is stuck with the concept of integrating the application under consideration within a web-application.

Another problem with web services is the need for data transfer objects (DTOs). A typical business object consists of both data and operations. A DTO is an object that is transferred using a web-service from server to client and vice versa. A DTO typically contains a subset of the business object fields and contains no methods – it is just a placeholder to transfer data from point A to point B. For every business object that needs to be transferred over the wire, a corresponding DTO object has to be created. This leads to the need of mapping between business objects and their corresponding DTOs (see Fig. 8). Such mapping is often called "the maintenance nightmare", because changes in the original business object involve changes in the corresponding DTO and also in the mapping algorithm.

IV. Solving Implementation Challenges in .NET

.NET platform brings a few things into table, providing alternative solutions to the implementation challenges described in the previous section. Let us start with the data structure. In .NET there is a possibility of creating partial classes. A partial class is a class that is defined in several files [13]. At compile time the final class is produced, linking all these files together in a single one. This feature was originally introduced to support multiple developers working on the same class – developers could declare the class as partial and one developer could implement one part of the class, while another developer implements another part, both doing that independently of each other [13].

Partial classes can also be used for a slightly different purpose – to create a common data structure and allow specifications of the corresponding classes where needed, thus providing development flexibility, increasing code reuse and modularity. The algorithm is as follows. First, one should create a library containing classes marked as "partial" with the attributes and possibly methods that are common to the whole system. Wherever it is needed to use those classes in another project that is part of the system, one links them to that project. If some of the classes need specialization (e.g. additional fields or methods) in the context of a current project, one creates a new class with the same name as the original and marks it "partial". In the "new" class one can add the necessary specialization. At the compile time the "new" class and the common class files are linked together into the single class. The key is that the common class file can be (re)used in multiple projects and is not bound to some specification. This way a common data structure can be created, providing the appropriate specification where necessary, thus solving the data structure challenge of the proposed architecture.

Let us proceed to the next challenge – dynamic command execution. In .NET there is a feature called Lambda Expressions [14] and a type of such expression is a predicate. Lambda Expressions provide the possibility to write predicates that are very similar in syntax with predicates used in logic [15]. Those predicates are used to specify some sort of conditions. An example usage is list filtering, e.g. to get all persons such that $\forall p(Salary(p) > 10000)$, one can use the following line of C# code:

```
var richPersons = persons.Where(
                p => p.Salary > 10000);
```

The logic expression and the program code are quite similar, allowing use of the out-of-the-box programming language constructs to describe logic expressions. This may be used in describing dynamic robot command conditions. Let us return to robot command example described with the expression "$Bump(B1) \wedge Bump(B2) \longrightarrow Stop(R1)$". This expression contains two parts: condition $Bump(B1) \wedge Bump(B2)$ and action $Stop(R1)$ that follows when condition is met. Such conditional action can be translated into the following C# code:

```
ConditionalAction abortOnBumper = new
ConditionalAction();
abortOnBumper.CommandAction = new
CommandActionAbort();
abortOnBumper.SetExpression(c =>
c.RobotStatus.Sensors.Bumper.BumpLeft &&
c.RobotStatus.Sensors.Bumper.BumpRight);
```

Predefined command action "CommandActionAbort" is used to describe the action to be performed when the corresponding expression evaluates to true. This approach provides flexibility in the sense of compositing the expressions and provides a feasible solution to dynamic command execution implementation challenge.

The last challenge is the communication infrastructure. .NET platform offers WCF (Windows Communication Foundation) services for communication. In contrast with traditional web-services, a WCF service can be hosted in a standalone application, and using basic http binding can provide interpretability with standard web-services, while other binding methods provide more functionality, but might be limited to WCF usage only. An example of this is binary .NET TCP (Transmission Control Protocol) binding. The data is sent in binary format, providing higher performance

compared to standard http binding, where data is sent in xml format.

Fig. 5. Experimental robotic platform

This binding method also allows for stateful services and most importantly – callbacks, allowing the server to initiate data transfer to a client without the need to setup a WCF server at the client side. Callbacks can be used to notify a client about status changes, thus allowing implementation of a push instead of pull mechanism.

Regarding DTOs, WCF services can benefit from the partial class usage idea described earlier. Since WCF uses serialization to send objects over the wire, i.e. object is serialized at point A, send to point B and de-serialized there, each class in the common data structure project should be marked for serialization. Attributes of those classes should also be marked for serialization. Assuming that both client and server (e.g. robot and UI application) have access to the common data structure project, it serves as a DTO (Data Transfer Object) library. The burden associated with mapping DTOs to their corresponding business objects (partial classes that contain specializations, in our case) is eliminated in the following way.

At the server (e.g. robot) side, one can use server specialization of the common data structure within the service host. Since it is possible to reuse types in referenced assemblies when generating a service client reference, one can reuse client specialization of the data structures at the client (e.g. UI application) side. Assuming that fields belonging to the specializations are not marked for serialization, the following actions take place when sending a specialized object from point A to point B. At point A the object is serialized as the common object, since only the fields belonging to the common object are marked for serialization. Object is then transferred from point A to point B, where it is de-serialized as specific to B. Only common attributes of the object are de-serialized (since they are the only ones that are marked for serialization), but the resulting instance is specific to B. This is possible, because additional fields (added via partial-class specialization) are ignored in the serialization process.

Thus, a clever use of out-of-the-box partial class feature of .NET framework together with WCF services can address the communication infrastructure implementation challenge of the proposed architecture.

V. Implemented prototype

To illustrate the proposed ideas in practice a prototype multi-robot system has been implemented. The prototype is built to carry out area-cleaning tasks. The basic idea is to join multiple existing vacuum-cleaning robots in a single multi-robot system to clean large areas that cannot be cleaned by a single robot in reasonable time.

iRobot Roomba [16] is used as the robot platform. To provide robot with the necessary computational capabilities, a laptop is installed on top of the robot (see Fig. 5). Robot localization is achieved by a combination of artificial landmarks recognized by an on-board camera and robot wheel encoders. Communication between the laptop and Roomba is provided through serial interface. C# is used as the programming language.

Implementation of the proposed architecture is straightforward. Each application defined in the proposed architecture becomes a standard Windows Forms application and each service becomes a WCF service. WCF services are hosted in their corresponding applications. A common data structure project is created and specifications of the necessary classes are created by using the partial class feature described in the previous section. One of the main data structures used in the system is “Robot Status” (see Fig. 5).

The partial class feature used allows defining common fields of the robot status data structure in the common data structure project, while providing the necessary specialization such as methods and members to read sensor data at the robot side as well as additional fields, such as “Sensor status text” at the UI side. The UI implementation is done by using a mix of Windows Forms and WPF (Windows Presentation Foundation) techniques, thus allowing both for a classic-style user interface and an option to define a UI structure in XAML (Extensible Application Markup Language). This provides a credit for the ease of development from the implementation perspective, since the only thing to do to reflect the current robot status in the UI is to set the robot data structure (received from robot via WPF service) as the data context of the corresponding XAML UI element.

Conditional actions of robot commands are implemented as a combination of predicates that provide a way to dynamically describe the conditions, and predefined actions (such as “abort”) to perform when the corresponding conditions evaluate to true. These conditions can be defined within one part of the system (e.g. the UI application), serialized, sent to another part of the system (e.g. the robot application), then de-serialized and interpreted properly.

VI. Results and discussion

This paper demonstrates that a general purpose development platform like .NET framework can be successfully applied to multi-robot management system development. Moreover, the implementation challenges identified can be addressed using out-of-the-box features of .NET framework. This leads to conclusion that .NET framework is a considerable choice when dealing with multi-robot system implementation.

Use of a general purpose implementation platform has several advantages over specific frameworks for multi-agent and multi-robot system development, such as JADE, Player-Stage and Microsoft Robotics Studio. First, it is not required to invest time and money in researching available frameworks or specific development environments. An experienced .NET programmer most likely already has the required knowledge to build a multi-robot system.

Second, it is straightforward in means of saving development time and computational resources to use an existing programming language feature over third party framework that provides similar functionality. Programming language features, however, tend to provide only the basic building blocks for system development. That is why third party frameworks are often used. Nevertheless, this is not the case, since the previously described implementation challenges can be addressed using out-of-the-box features of .NET Framework. The developer is saved from the burden of implementing the basic building blocks of the system.

Third, the use of a general-purpose development platform greatly enhances the possibilities of code reuse among different software projects. The use of partial classes together with WCF services allows creating reusable common data structures that can be specialized where needed. Developers can reuse both code and knowledge from previous projects regarding WCF services, partial classes and LINQ expressions to build a multi-robot system.

An important aspect of the proposed architectural solution is the dynamic command execution approach. First, it allows for creation of previously undefined commands using decomposition. Assuming that a basic set of commands is defined, one can create custom hierarchical commands using a combination of the basic commands. There is no need to predefine them in code or in some configuration file – it is possible to create such commands on the fly, send them to robot and the robot will know what to do. One is not limited by a set of commands provided by the robot interface. In contrast with behavior-based approaches (like [17]), robot behavior is defined by currently executing command. Since the command can be hierarchical and each command can contain a number of conditional actions, one can easily implement such behaviors as “avoid obstacle” by defining the appropriate nested commands and their conditional actions.

Together with decomposition, each command may have one or more conditional actions. The actions themselves are predefined and refer to the command management (e.g. to abort current command when some condition holds). The important part of conditional action is the condition. The proposed approach allows defining conditions on-the-fly by using predicate expressions provided by .NET framework. This gives the developer the flexibility in constructing conditions and also eliminates the need to invest time and money in creating logic parsers.

Everything comes at its price and .NET framework is no exception. A general purpose personal computer (PC) is required to run the full .NET framework. This factor may seem to be a constraining factor when it comes to hardware cost of the robot. However, a term “PC” does not always refer to the combination of most powerful (and thus – most expensive) devices on the market, such as systems based on Intel Core i7 processors. In our prototype system we experimented with an Intel Atom board based system, which total cost did not exceed 300$ (including power supply, memory and SSD drive). In contrast, a microcontroller board with Ethernet shield and Wi-Fi bridge module costs about 200$. There is also the possibility to build and solder everything from scratch, but one must weigh the cost of off-the-shelf components against the time spent on building an identical item.

An alternative to using full .NET framework is .NET Micro Framework that is a subset of .NET framework designed for highly resource constrained devices. According to Microsoft [18] the .NET micro framework can run on devices having as little as 300kb of RAM and a processor running at 27 MHz.

The processing power available while using PC differs dramatically from the processing power available in a microcontroller. Thus, using a PC instead of a microcontroller offers a greater potential for robot autonomy – localization, mapping and planning can all be executed on an onboard robotic platform. This differs, however, from application to application. A PC might be a complete overkill in one application and an absolute necessity in another.

ACKNOWLEDGMENTS

This research has been partly supported by the European Social Fund within the project “Support for the Implementation of Doctoral Studies at Riga Technical University” and also supported by ERAF European Regional Development Fund project 2010/0258/2DP/2.1.1.1.0/10/APIA/VIAA/005 Development of Intelligent Multiagent Robotics System Technology.

REFERENCES

[1] Collier, R., et al., *Beyond Prototyping in the Factory of Agents*, in *Multi-Agent Systems and Applications III*, V. Marík, M. Pechoucek, and J. Müller, Editors. 2003, Springer Berlin / Heidelberg. p. 1068-1068.

[2] Vaughan, R. and B. Gerkey, *Reusable Robot Software and the Player/Stage Project*, in *Software Engineering for Experimental Robotics*, D. Brugali, Editor 2007, Springer Berlin / Heidelberg. p. 267-289.

[3] ellifemine, F., C. Giovanni, and D. Greenwood, *Developing Multi-Agent Systems with JADE*2004, Chichester: John Wiley & Sons Ltd. 286.

[4] Vecchiola, C., et al., *Agentservice: A framework for distributed multiagent system development.* International Journal of Computers and Applications, 2009. **31**(3): p. 204-210.

[5] Campo, A., et al., *Artificial pheromone for path selection by a foraging swarm of robots.* Biological Cybernetics, 2010. **103**(5): p. 339-352.

[6] Grosso, A., et al., *A Context Aware Multi-robot Coordination System based on Agent technology*, in *WOA 2010, From Objects to Agents*2010: Rimini, Italy.

[7] Chaimowicz, L., et al. *ROCI: A distributed framework for multi-robot perception and control*. 2003. IEEE.

[8] FIPA. *FIPA Home Page*. [Online] 2012 [Accesed 2012 May. 10]; Available from: http://www.fipa.org/.

[9] Ahmadi, M. and P. Stone. *A multi-robot system for continuous area sweeping tasks*. 2006. IEEE.

[10] Coltin, B., M. Veloso, and R. Ventura. *Dynamic User Task Scheduling for Mobile Robots*. in *2011 AAAI Workshop*. 2011. San Francisco, California, USA.

[11] Franchi, A., et al., *The sensor-based random graph method for cooperative robot exploration.* Mechatronics, IEEE/ASME Transactions on, 2009. **14**(2): p. 163-175.
[12] Haines, M.N. and M.A. Rothenberger, *How a Service-Oriented Architecture May Change the Software Development Process.* Communications of the ACM, 2010. **53**(8): p. 135-140.
[13] Microsoft. *Partial Classes and Methods (C# Programming Guide).* [Online] 2012 [Accesed 2012 May 12]; Available from: http://msdn.microsoft.com/en-us/library/wa80x488.aspx.
[14] Microsoft. *Lambda Expressions (C# Programming Guide).* [Online] 2012 [Accesed 2012 Jun 10]; Available from: http://msdn.microsoft.com/en-us/library/bb397687.aspx.
[15] Russell, S.J., et al., *Artificial intelligence: a modern approach*. Vol. 74. 1995: Prentice hall Englewood Cliffs, NJ.
[16] iRobot. *Product Catalog*. [Online] 2012 [Accesed 2012 Mar. 15]; Available from: http://store.irobot.com/home/index.jsp.
[17] 17. Gifford, C.M., et al., *A novel low-cost, limited-resource approach to autonomous multi-robot exploration and mapping.* Robotics and Autonomous Systems, 2010. **58**(2): p. 186-202.
[18] Microsoft. *.NET Micro Framework.* [Online] 2012 [Accesed 2012 May 20]; Available from: http://www.netmf.com/.

Aleksis Liekna

Received his Bc.sc.ing degree in 2008 and his Mg.sc.ing. degree in 2010 from Riga Technical University. At the moment he is a PhD student at Riga Technical University. His major field of study is computer science.

He is working as a Research Assistant at Riga Technical University. His research interests include artificial intelligence and multi-agent systems.

He was awarded by the Latvian Foundation for Education for his bachelor's thesis "Development and Implementation of Reinforcement Learning Model".

Agris Nikitenko

Received his Dr.sc.ing. in 2006 from Riga Technical University as well as Bc.sc.ing. and Mg.sc.ing. focusing on artificial intelligence in his thesis.

Currently he is docent in the Department of Systems Theory and Design of Riga Technical University and vice dean of study affairs in the Faculty of Computer Science and Information Technology. His scientific interests cover artificial intelligence and autonomy as well as their application in robotics.

He has received award of Verner fon Siemens for his doctoral thesis in 2006. At present he is member of IEEE and ACM as well as represents Latvia in NATO RTO AVT panel.