The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Efficient Communication in Autonomous Robot Software

Dirk Thomas and Oskar von Stryk, *Member IEEE*

Abstract— **Software for autonomous robots solving challenging tasks in research or application is becoming increasingly complex. System integration has to deal with various different functional components. To decouple those components from each other and to enable a modular and reuseable software architecture a robot middleware is typically used. But this intermediate layer introduces significant additional overhead during run-time. In this work a methodology is described to utilize specific application characteristics to improve communication efficiency between different robot software modules. By composing several components in a single thread memory copying or locking operations can be avoided, when data is exchanged between those parts. The optimization can be achieved without compromising the advantages of a communication layer. Still the modifications are transparent to the maybe already existing components. Experimental results in the scenario of autonomous soccer-playing humanoid robots are presented and exhibit remarkable reduction in communication overhead. Furthermore this approach can be implemented in or on-top of other communication layers.**

I. INTRODUCTION

In the past software developed for robotic systems was often tightly coupled with a single hardware and focused on a single task or scenario. It was mainly build for proof of concept of developed algorithms or designed hardware. Therefore a clear separation and reuse of different parts of the software was not mandatory.

But in the last years robotic applications are targeting a new domain. As the technology should be used in industrial applications, the goal is shifting towards developing integrated systems.

There are many ongoing projects in various different areas. For the DARPA Grand Challenges [1] vehicles are developed which are capable of autonomously driving in traffic and performing complex maneuvers. Other projects are the robot PR2 developed by Willow Garage [2] and the Care-O-bot developed by Fraunhofer IPA [3]. These mobile robots should be used in human environment and take over complex manipulation and transportation tasks. The international research and education initiative RoboCup introduced the concept of autonomous soccer-playing robots. This scenario works as an environment for providing standardized benchmarks for autonomous robots to foster artificial intelligence and robotics research where a wide range of methodologies technologies can be examined and integrated. The ultimate goal of RoboCup is to develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.

All of these scenarios have several aspects in common: The developed robot software applications integrate several different domain specific algorithms and functionality and provide an integrated overall system. The functional components vary based on the targeted objectives but cover different approaches for processing and filtering sensor data, self localization and mapping and behavior [4] and motion control. Each scenario has its own focus but since the requirements are continuously increasing the robot application software become more and more complex.

The combination of so many different parts in an integrated robot application software is a significant challenge. The effort of integrating, testing and debugging the overall system is tremendous. Thus the application design and general concepts from software engineering become increasingly important.

The software to connect those various components is typically called a robot middleware [5] as it shares similar functionality as typical middleware by providing methods for software components to interact with each other by exchange of data or function calls. On the one hand such a middleware eases the development of complex robot application software but on the other hand it is accompanied with an overhead especially during run-time. Also, a number of different requirements for robot software must be considered.

A. Communication layer

The middleware works as an intermediate layer - aka communication layer - and provides functionality to efficiently integrate components into a single application. The usage of such a layer should lead to a clear separation between the different parts of an application by explicitly declaring any interface between each other. An interface description allows an easier reuse and recombination of separate components for different tasks or in different scenarios. Sometimes when the interface definition is modeled in an abstract way it also supports connecting different components developed in different program languages. Additionally the explicit definition of interface among the components ease testing and debugging since replacing not-to-be tested components with stub implementations becomes simple.

Before continuing, the terminology of a *component* should be specified. A component is a piece of software which provides a specific functionality to fulfill the application's tasks. To foster a later reuse in a different context it should be as small as possible. Since commonly any algorithm depends on some input or output each component is coupled with a

D. Thomas and O. v. Stryk are with the Simulation, Systems Optimization and Robotics Group, Computer Science, Technische Universit"at" Darmstadt, Darmstadt, Germany {dthomas, stryk}@sim.tu-darmstadt.de

set of incoming or outgoing data. But on the other hand it should be fully decoupled from any other component.

The usage of a communication layer completely separates the components from each other. Therefore changes in some components or executing those on distinct hardware can be made transparent to others as long as the exchanged data stays the same.

Some of the most widely used tools are Player [6], Orocos [7], MSRS [8] and ROS [9]. Another approach named RoboFrame [10], [11] was developed in the authors' group. The majority of communication layers are founded on a message-based communication concepts [12].

In a message-based communication messages correspond to what interfaces are in an object-oriented language. Therefore to decouple components from each other, a set of messages for exchanging data must be defined. The interaction between components is only handled through those messages and never directly. This way the components are only coupled with the messages they receive and send, but no more with other components. This approach allows flexible recombination of components — of course only when the message are well designed.

Another task of a middleware beside the communication is the execution of the functional components. Commonly the components can be triggered both on incoming messages and on a timer event. For some solutions this is done by the components itself. But the decision when to trigger each component can be different based on the concrete application it is used in. Therefore the execution order and interval should not be specified in the component itself, but be defined in the concrete application consisting of several components.

B. Overhead due to separation

The abstraction and flexibility provided by a communication layer comes with an additional expense. There are two main aspects were a middleware brings overhead along: copied memory and mutual exclusion.

When messages are exchanged between components the data must be copied in many cases since each component may be executed concurrently or may run on a separate computer. This may involve a plain copy of an object's memory or even a serialize/deserialize cycle. E.g. when using Player or ROS each component is running as a separate process and therefore any exchanged message must be copied for every receiver. Neither of the common tools currently provides a copy-on-write semantic. The overhead for these memory operations are scaling with the amount of exchanged data, so large amount of data results in a large overhead due to the communication layer.

Additionally since the middleware has to deal with multiple processes and/or threads and synchronize sharing of global resource it requires mechanisms for mutual exclusion. The overhead for those locking operations increases with the frequency of communicated data, so for high repeat rates this aspect become more severe.

Even when a communication layer tends to keep these overheads low there, the provided gain of abstraction and flexibility will inevitably increase the overall resource consumption of the middleware.

C. Requirement for efficiency for mobile robots

When looking to the domain of mobile robots the requirements for efficiency are crucial. The lower the payload of a hardware platform is, the more restricted are the resources. Many mobile robots have therefore quite limited resources in terms of CPU power. Such a scenario demands for as few overhead as possible.

But creating single monolithic applications without clear separation and interfaces just for the sake of minimal overhead would not be beneficially either. The benefits of a communication layer are still meaningful.

A good solution should therefore bring the best of both approaches together. It should foster the flexibility advantages of a middleware, but whenever possible minimize additional overhead. Any possible optimization to improve performance should be transparent to not undermine the reusability and decoupling of components.

II. Example scenarios

As example scenarios for the later described optimization two different scenarios from the authors' group are described. Both originate from the context of RoboCup. Currently the authors' group is participating in two leagues: the humanoid soccer and the rescue league.

Fig. 1. Darmstadt Dribblers (in the foreground, magenta jerseys) playing a 3-on-3 soccer game during RoboCup 2009 [13].

The humanoid robot team of the Darmstadt Dribblers [14] is the RoboCup 2009 and 2010 champion. Dynamic walking, kicking the ball while maintaining balance, localization and team coordination are among the basic research issue in this league. Furthermore visual perception of the ball, other players, and the field, self-localization and team play are investigated in the Humanoid League. Since the field of view is restricted by rule to human-like 180 degrees the modeling of recently seen objects is an important requirement. The robots play fully autonomously in three-on-three matches (cf. Fig. 1).

While at first glance playing soccer may not appear as a too difficult task as it seems not to be too sophisticated

for humans, the required skills to act in such a dynamic environment are fairly complex. For a rough comparison the required behavior control of the soccer-playing autonomous humanoid robots requires more than 120 hierarchically organized state machines (realized in XABSL [4]). By contrast the behavior of an autonomous car participating in the DARPA Urban Challenge typically consisted of only a dozen state-machines. Indeed the involved sensors and algorithms for sensor processing are not comparable.

The other team Hector was founded in 2009 and is reusing some of the already developed components while adjoining several other domain specific components. Since the rescue team is a joint team of several different groups the aspect of a clear separation between the different domain specific algorithms was mandatory.

In both scenarios the mobile robots act fully autonomously. In the soccer league the aspect of playing with multiple robots per team further increases the complexity of the robot application software. Both projects are based on RoboFrame and share a common code base and therefore many components.

A. Components and execution order

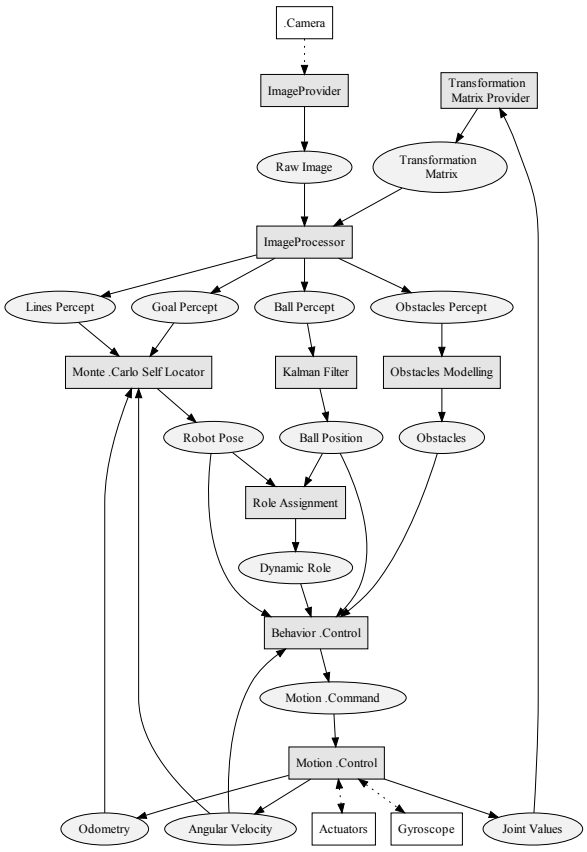


Fig. 2. Application layout of Humanoid RoboCup team. Rectangles illustrate the components, ellipses the exchanged messages.

To give a detailed look to the overall application layout Figure 2 depicts the most important components and there exchanged messages.

As depicted in the figure the application consists of several components which all focus on a single functionality. E.g. one component – the ImageProvider – has the only task to provide images. Those images could be acquired from a real camera, provided by a simulator or loaded from a log file. Afterwards they are processed by the ImageProcessor component which outputs some messages with information about the detected objects in the image – called Percepts. To determine the position of detected objects in robot coordinates the image processor additionally requires the position and orientation of the camera when the images was taken. These data – called transformation matrix – is provided from the TransformationMatrixProvider which itself requires the current joint values to determine the cameras position and orientation.

The execution order of the various components is mostly defined by the incoming data. Some of the components do process the incoming data when certain criteria match. E.g. the image processing is only executed for new images but not for new transformation matrices.

B. Amount of exchanged data

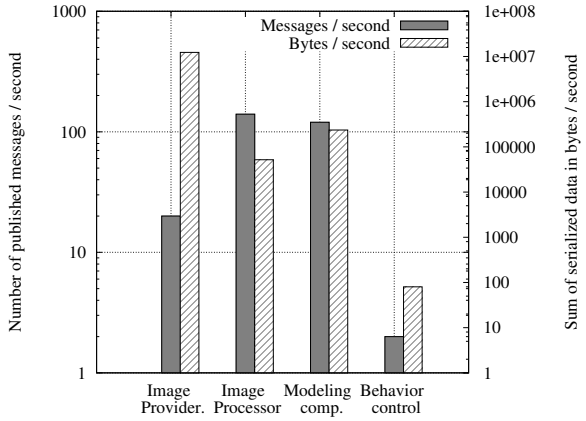


Fig. 3. Number of published messages (solid) and the size of serialized data in bytes (shaded) for some components.

While some of the shown messages only contain small amount of data others are quite large, e.g. the images from the camera. Notably each single image is about 600 KB and is getting fetched and processed with about 20 Hz (for the described measurements). Copying these large amount of data inside of the communication layer means a significant overhead. To identify the messages and their sizes some statistics have been collected during a ten minute half time of a soccer test game as seen in Figure 3. This gives a rough overview how many messages and data are exchanged in such modular applications. In the measured scenario about 500 messages were exchanged with a payload of approximately 14.5 MB per second.

Since each component is running in a separate thread – as it is done for all of the above mentioned communication layers – the whole data must be copied (at least once). The overhead of those memory copy operations is highly dependent on the hardware platform used. In this example an AMD Geode processor with 500 MHz is used.

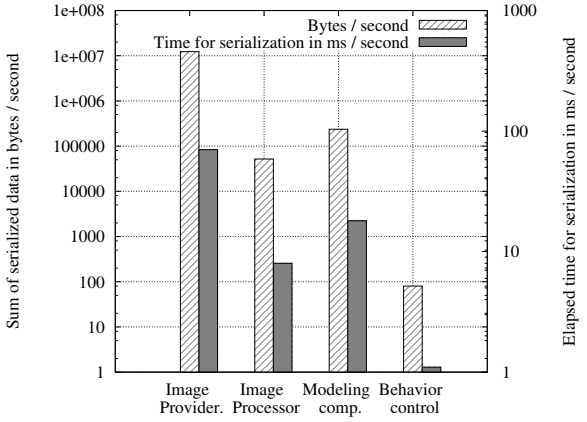
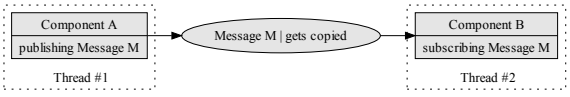


Fig. 4. Size of serialized data in bytes (shaded) and the elapsed time to serialize these messages (solid).

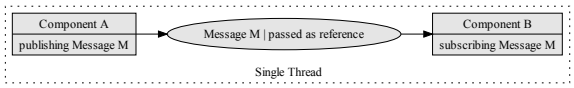
The overhead in terms of time spend for serializing these messages is shown in Figure 4. Neither any locking overhead nor the deserialization is taken into account. The required time need not be proportional to the size of the data, since different kind of data may require different serialization strategies. E.g. the data of an image can be copied on block while complex objects have to be serialized member by member.

For the described scenario the overhead just for serializing the messages is a significant amount of approximately 15 percent of the overall application run-time.

III. UTILIZE APPLICATION SPECIFIC CHARACTERISTICS



(a) Messages getting copied between components



(b) Messages passed as references inside a single thread

Fig. 5. Message exchange with/without copying memory.

When looking at the execution order of the various components as shown in Figure 2 they are not at all arbitrary. E.g. the ImageProcessor is executed if and only if the ImageProvider component has been executed before and provided a new image. The order of execution is application specific, but defined through the set of component communicating with each other.

Usually the execution order of the components is resolved by the middleware during run-time depending on exchanged messages between the components. But in many cases the execution order does not change during run-time. This application specific characteristics can be utilized to improve the efficiency of the communication layer.

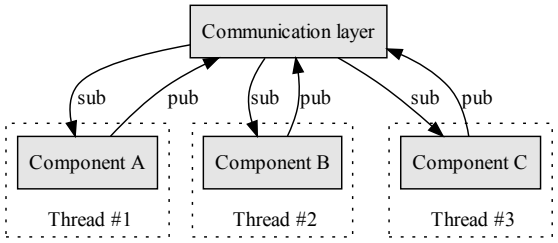
When the order of execution of components is preassigned based on the application specific flow of data, then some components could be executed one after another in a single thread (cf. Fig. 5 (a) and (b)). As long as the messages are not requested by any other component outside that thread, memory copyiny and locking operations are not required at all. E.g. the ImageProvider passes the image as a references to the ImageProcessor which can use them without worrying about the data being modified by another components concurrently. And due to the reference passing any overhead for copying the data is circumvented.

The communication layer may use a shared memory to pass the data from one component to another. But even when the memory copying is avoided locking mechanisms are required due to the concurrent execution of each component. In this case the described approach of executing multiple components in a single thread makes even the locking overhead obsolete.

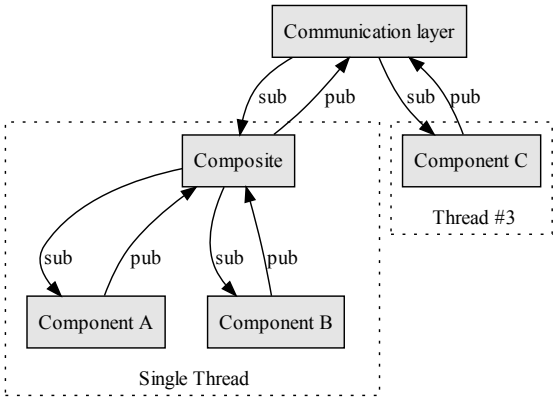
A. Composite pattern

The described optimization equates to the *composite pattern* [15], which is a partitioning design pattern known in software engineering. It allows a group of components nested in a composite to be treated in the same way as a single instance of a component.

In this context multiple components (called *leafs* according to the terminology of the pattern) are *composed* in a meta-component (called *composite*). The composite implements the same interface as the Leafs and can therefore be treated homogeneously.



(a) Components directly interacting with the communcation layer



(b) Composite composing multiple leafs in a single meta-component

Fig. 6. Interaction of components with the communication layer.

When using a common publish/subscriber messaging paradigm the composite would ”take over” all publications

and subscriptions of its leafs (cf. Fig. 6 (a) and (b)). When message are received from the composite they are stored in the composite and references are relayed to the corresponding leafs. Since no concurrency can take place inside a composite and it's leafs passing references is possible and no copying or locking is required.

Obviously the API must assure that a subscriber can not manipulate the received message since it may be used by multiple leafs. This is achieved by only making constant references to the messages available for the receiving components.

For publications the same relaying is done. The message can again be passed by reference to other leafs. The data must only be send to the middleware and therefore serialized / copied when other components beyond the composite have subscribed to the message.

The approach should be transparent to the functional components so that no modifications have to be done to apply the described optimization. But to implement this pattern the used communication layer must be designed to support such an approach of dependency injection between the middleware and the components.

E.g. when looking to ROS any node implements the interaction with the publish/subscriber infrastructure itself. Therefore a potential composite would not have any chance to intercept these calls and redirect them through the composite itself. Any kind of dependency injection is therefore inhibited.

B. Implementation in RoboFrame

In RoboFrame the interface between the communication layer and the components is therefore especially designed to allow the usage of composites. Every component consists of member variables which act as proxies to the real data which can be used to either send or receive messages. These proxies are named InBuffer for received and OutBuffer for sent messages.

During the initialization phase of the application the published and subscribed messages are queried from the components using a method defined in the interface for components. For every component the framework allocates the memory for the set of messages specified. References to these data are than handed over to the proxies inside the components.

With the described interface applying the composite pattern is easy (cf. Fig. 7). A composite component encapsulates a set of other components in a single thread. When the framework queries for published and subscribed messages the composite dispatches this call to all subcomponents and returns the aggregated set of subscribtions and publishments to the middleware. This approach is transparent for the components as well as for the framework. Neither of both sides is aware that a composite is in between. But effectively all components inside a composite share the same message references as illustrated in the last figure.

One very important aspect is, that every received message can only be read but not modified. Therefore the proxies for received messages provide only constant references to the data. Furthermore the sender of a message should not keep a reference to the sent data or rather is not allowed to modify it after publishing.

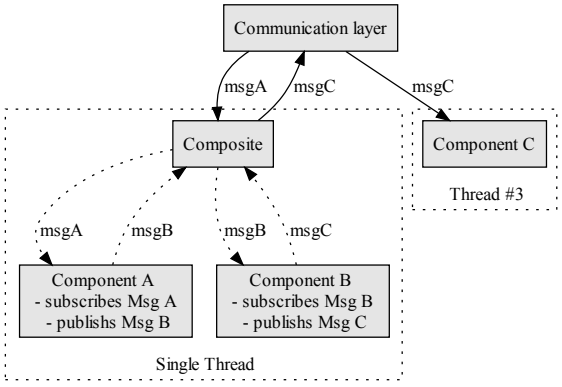


Fig. 7. The composite pattern applied in RoboFrame. Dotted lines describe access to messages by reference instead of copying.

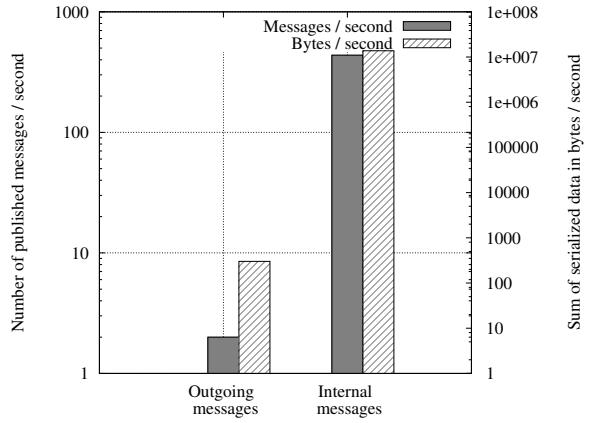
Obviously since all components inside a composite are executed sequentially, locking operations are obsolete. Likewise no serialization/deserialization or memory copy is required since all components share the same referenced messages using their own proxy member.

IV. RESULTS

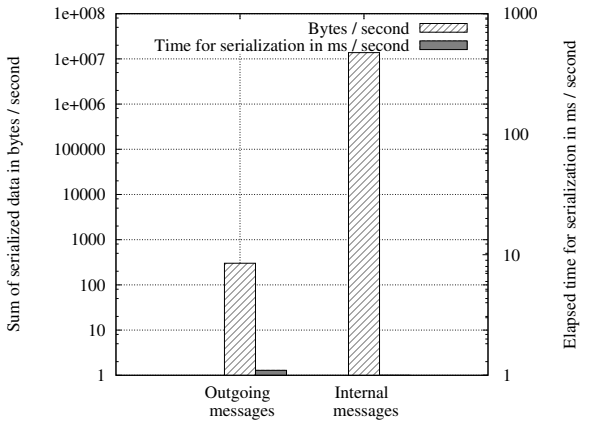
Due to the characteristics of the application used in RoboCup all components are split into only two different composites. One first encapsulates the components which read the sensors and control the actuators - this thread is called *Motion*. The second composite - which is called *Cognition* - encapsulates all other components which perform image acquisition, transformations matrix calculation, image processing, self localization, modeling of balls, obstacles and roles and finally the behavior control. All these components can be executed sequentially in our scenario because a) the CPU has only one core and b) it would not make much sense to run the behavior decision step again if no new image is available, since this is the only external sensor.

The capability to group components in which each is executed sequentially is obviously highly dependent on the application. But even if it is not possible to group as many components together as in the described scenario it is always applicable to at least group components for e.g. sensor data acquisition and processing. Since the raw sensor data often cover the major amount of exchanged data the advantage is in particular valid for grouping these components and therefore applicable to most other scenarios and applications.

Since in the described scenario only few messages are exchanged between the two composites most of the data exchange is done using references. The impact on the number of copied messages and therefore serialized/deserialized bytes is therefore quite significant (cf. Fig. 8(a)). The amount of data is reduced by over 97 percent compared to exchange all data between independent components. As expected the CPU time required to handle the communication drops



(a) Most messages are passed internally, only a small fraction is communicated to external components



(b) Any time for serialization is omitted for internal messages

Fig. 8. Messages exchanges for the Cognition composite.

approximately about the same ratio (cf. Fig. 8(b)). Only a fractional amount of time is spend for serialization when the composites handle most of the exchanged messages by passing references.

Using the described optimization the communication overhead in the example scenario was reduced from over 100 ms to only one ms per second. This is achieved withpout changing the components and therefore without compromising most of the advantages of the communication layer.

One disadvantage should also be noted: this solution is obviously not applicable across language boundaries. This limitation originates directly from the concept of passing references of objects between components.

V. CONCLUSIONS AND OUTLOOK

In this paper an approach has been described to enhance the efficiency of the communication layer used in complex application of autonomous robot software. Therefore the application specific characteristics have been utilized to reduce the communication overhead. The composite pattern is applied to nest components in a single-threaded composite and thus eliminating any need to serialize, deserialize, copy data or use locking operations as required by other approaches. It has been demonstrated that in the described scenarios the gained improvements in communication efficiency are significant. Additionally the basic principles for an API have been described, which supports the usage of composite components without altering existing components.

Depending on the specific scenario the concept of composition may be useful for other applications as well. The described optimization should be considered for future (re-) design of communication layer APIs. Other existing robot middleware can examine how to integrate the concept of composition into their software and estimate the efficiency improvements coming along.

In the future, it would be fruitful to adopt the concept of modified interfaces for other common robot middleware. The possibility of using composite components is only one kind of dependency injection between the middleware and the components.

ROS [9] is most likely the best candidate for such an adoption, since its publish-subscriber concept is quite similar to RoboFrame. Additionally its growth and adoption by several groups in the last years makes it a sustainable platform for the future.

In an ongoing work other patterns besides the composite are applied to inject different kind of functionality between those two layers. Some of the applied patterns are filter, gateway and strategy. Each of those can provide a specific kind of functionality which is either more efficient if provided by the middleware or even not achievable without such a dependency injection mechanism.

REFERENCES

[1] DARPA Grand Challenge, http://www.darpa.mil/grandchallenge/
[2] Willow Garage PR2, http://www.willowgarage.com/pages/robots/pr2-overview
[3] Fraunhofer Care-O-bot, http://www.care-o-bot.de/
[4] M. Lötzsch, M. Risler, M. Jüngel, *XABSL – A Pragmatic Approach to Behavior Engineering*, Proc. of IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2006, Beijing, China, pp. 5124-5129,
[5] Definition of middleware, http://en.wikipedia.org/wiki/Middleware
[6] B. P. Gerkey and R. T. Vaughan and A. Howard, *The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems*, Intl. Conf. on Advanced Robotics (ICAR), Portugal, 2003.
[7] H. Bruyninckx, *Open robot control software: the Orocos project*, In: IEEE International Conference on Robotics and Automation (ICRA), pages 2523-2528. IEEE Press, 2001.
[8] Microsoft, Microsoft robotics studio, http://msdn2.microsoft.com/en-us/robotics/default.aspx
[9] ROS, open-source meta-operating system for robots, http://www.ros.org/
[10] S. Petters, D. Thomas, O. von Stryk, *RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots*, Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2007.
[11] S. Petters, D. Thomas, M. Friedmann, O. von Stryk, *Multilevel Testing of Control Software for Teams of Autonomous Mobile Robots*, Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008) 2008, pp 183-194.
[12] Message-oriented middleware, http://en.wikipedia.org/wiki/Message-Oriented-Middleware
[13] Photography by David Kriesel, http://www.dkriesel.com.
[14] Darmstadt Dribblers, Humanoid Kid-size league, RoboCup, http://www.dribblers.de
[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: abstraction and reuse of object-oriented design*, 2002, pp 701-717.