2012 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 7-12, 2012. Vilamoura, Algarve, Portugal

MIRA - Middleware for Robotic Applications

Erik Einhorn[1], Tim Langner[2], Ronny Stricker[1], Christian Martin[2], Horst-Michael Gross[1]
[1]Neuroinformatics and Cognitive Robotics Lab, Ilmenau University of Technology, Germany
[2]MetraLabs Robotics GmbH, Ilmenau, Germany

Abstract— **In this paper, we present MIRA, a new middleware for robotic applications. It is designed for use in real-world applications and for research and teaching. In comparison to many other existing middlewares, MIRA employs novel techniques for communication that are described in this paper. Moreover, we present benchmarks that analyze the performance of the most commonly used middlewares ROS, Yarp, LCM, Player, Urbi, and MOOS. Using these benchmarks, we can show that MIRA outperforms the other middlewares in terms of latency and computation time.**

I. Introduction

Currently, most autonomous mobile robots still operate within the safe laboratories of their developers and are used by groups of researchers that work on single isolated robotic tasks like navigation or human robot interaction. However, with the recent improvements in these fields concerning methods, algorithms or new hardware components, we are now approaching an era where those autonomous mobile robots are more and more employed in the real-world for applications, where the robots are used to carry out certain tasks. For instance, mobile robots are already used as shopping assistants in public environments, such as shopping centers and home improvement stores [1], or as companion to assist elderly people in their home environments [2].

Those applications can become complex very quickly due to their large scale and scope. They range from low-level hardware driver functions up to higher level methods for robot navigation, user detection and tracking and finally the application logic, user interfaces and dialog systems that interact with the user in an intelligent way. It is not tractable to write all these capabilities as a single monolithic application, since it would result in complex and unmaintainable structures that require rewriting large portions of the application's code from scratch in case the development of a different application is required.

Consequently, the above mentioned capabilities and algorithms are usually written as separate software modules that can interact with each other by passing data and messages. In such a modular design, a so called *middleware* takes an important role. It acts as the 'glue' that ties the modules together and provides the required mechanisms for the communication between the software modules. Such a modular design is advantageous in many ways. It allows working on different parts of the application in parallel, makes the large complexity manageable [3], and results in a loose coupling of the software modules, which is a desired goal of structured software design to support high maintainability and reusability. On the other hand, the modular design results in some communication overhead due to the creation and transmission of messages between the modules. Therefore, it is important for a middleware to provide high-performance communication techniques with low CPU usage. Especially for robotic applications, efficiency and low latency is crucial since the computations need to be performed in real-time, and the applications must respond and react quickly.

In this paper, we introduce a middleware for robotic applications (MIRA) that we have developed during the last two years and describe the implemented concepts in detail. We compare MIRA with the existing middlewares that are most commonly used by the robotic community and show that it outperforms the existing middlewares in terms of latency, computation time and usability. Another important contribution of this paper is the first performance comparison of the different middlewares. These benchmarks reveal partially surprising results that need to be taken into account by researchers and developers when designing future robotic applications.

II. Related Work

Due to the described importance of middlewares for robotic applications, more than a dozen of such systems have been developed in recent years, where most of them are available as open source. A survey on this field and a comparison of the different projects is given in [4] and [5].

Subsequently, we give a short overview of the robotic frameworks which have the largest impact in the community. Here, we are especially focusing on those frameworks that we compare with MIRA in section V. A compact survey of all examined middlewares is given in TABLE I below.

CARMEN - The Carnegie Mellon Robot Navigation Toolkit [6] was one of the first middlewares. It supports different software modules that are realized as separate processes that communicate via Inter Process Communication (IPC) based on TCP/IP sockets. Since there have not been any updates since 2008, it can be considered as outdated and is only listed here to give a complete overview.

Player [7] aims to provide simple and clean interfaces to a robot's sensors and actuators. Device modules - so-called *drivers* - run in a server's process using one thread per driver and can be accessed by clients via TCP. By the use of interfaces multiple different devices can be manipulated by the same control client. The support for robot hardware

*This work has received funding from the European Community's 7th Framework Program (CompanionAble project under grant agreement no. 216487) and the German Federal Ministry of Education and Research (APFel project under grant agreement no. 13N10797).

TABLE I: Survey of the examined middlewares.

	MIRA	ROS	Yarp	LCM	Urbi	Player	MOOS
Communication Structure	decentralized	name-/parameter server	nameserver	decentralized	centralized	client / driver model	central database
Communication Mechanism	intra-process, TCP	intra-process, TCP, UDP	intra-process, TCP, UDP	UDP multicast	intra-process, TCP, RTP	intra-process, TCP	TCP
Data Transport	publisher / subscriber, RPC	publisher / subscriber, RPC	observer pattern	publisher / subscriber	change / access notification	publish / read, interfaces	store / fetch
Message Types	any C++ types and classes	IDL using PODs	nearly any C++ types and classes	IDL using PODs	C++ types, UVars, UObjects	IDL using PODs	string, double
Supported Languages	C++, Python, JavaScript, ...	C++, Java, Python, ...	C++	C++, Java, C# Python, ...	C++, Java, urbiscript	C++, Java, Python, ...	C++, Java
Supported Platforms	Linux, Win OS X (planned)	Linux, OS X Win (partial)	Linux, Win, OS X	Linux, Win, OS X	Linux, Win, OS X	Linux	Linux, Win

can be extended by adding new drivers. Other advantages include that clients can be run on different platforms and can be implemented in many different programming languages. It also supports a virtually unlimited number of clients connecting to the server. However, the centralized server approach exposes a single point of failure.

ROS - The Robot Operating System [8] is nowadays one of the most widely used middlewares for robotic applications. It takes up many of the ideas and design decisions of Player. Even though it uses peer-to-peer TCP connections for communication between the software modules - called *nodes* - it needs a centralized server. This *master* is used for name look up and parameter storing and represents - like in Player - a single point of failure. ROS aims to support cross-language development by using a simple, language-neutral interface definition language (IDL) to describe the messages sent between modules [8]. The used IDL is suitable to describe plain old data structures (POD), i.e. collections of field values, arrays, etc. However, higher level object oriented concepts like inheritance and polymorphism can hardly be modeled. As each node is run in a separate process, the data transfer between nodes via TCP results in a drawback when exchanging large amounts of data. To overcome this limitation and to reduce communication overhead, the concept of *nodelets* was introduced lately. This concept supports running multiple modules within the same process. Each nodelet uses its own thread, and data can now be exchanged within the same process by sharing the same memory. However, the user has to take care of concurrent access and thread synchronization by themselves, e.g. by using mutexes or by allocating new memory whenever new data is written. This again results in a large performance penalty as shown in our results. An advantage of ROS is that it allows nodes to provide a service interface to others via RPC (remote procedure calls). Also, since it is widely used by the robotics community, many drivers, algorithms and software is available in public repositories as open-source.

Urbi [9] consists of two parts. The first part *UObject* provides a C++ API where components like drivers and algorithms can be designed and exposed to the second part - *urbiscript*. Urbiscript is an innovative event-based script language that is used to connect the components in an application. Urbi allows the user to decide whether to run the components within the same process or in different processes and on different systems transparently at runtime. In the latter case, TCP or UDP based RTP is used for data exchange. It also takes care of thread synchronization and concurrent data access. Another advantage is that there is support of a *0-copying* mode when running components in the same process that prevents unnecessary copying of data. However, it reveals the same disadvantages as in Player and ROS, since it uses a centralized approach where multiple clients connect to a master server for executing Urbiscript code.

LCM - The Lightweight Communications and Marshalling library [10] aims to simplify the development of low-latency message passing systems, especially for real-time robotics research applications. Like ROS, it utilizes the publisher/subscriber pattern to transmit messages between different processes using a platform- and language independent type specification language. LCM bases its communication on UDP multicast without the need of a central communication hub.

MOOS [11] is a cross platform middleware for robotics research. It uses a communication network with a star-shaped topology where each client has a single communication channel to a central server - the MOOSDB. Data is published as named messages by a client in either string or double format and stored in this database. Other clients can fetch not only the latest data but also the history of changes that were made to the data since the last read. It can be assumed that the lack of type-safety, the additional costs for parsing and sending human readable string data and the need to store data in a central database before sending it to a subscribed client leads to performance problems.

YARP - Yet Another Robot Platform [12] aims to minimize the effort devoted to infrastructure-level software development by facilitating code reuse, modularity in order to maximize research-level development and collaboration. Therefore, it includes support for a transport-neutral interprocess communication model based on so-called *Ports*. Ports can manage multiple connections for a unit of data either as input or output. Each connection allows sending and receiving of data supporting different data rates and different protocols (e.g. TCP, UDP or shared memory). A central server is used for maintaining a list of all ports and how to connect them.

This overview shows that many of the reviewed middlewares share the same ideas and basic concepts. Most of them also share the same disadvantage of a centralized approach. This poses a problem especially in multi-robot scenarios.

Here, each robot acts as an independent system that interacts with the other systems. If the central server fails, the whole system with all robots will stop working, even though each robot could continue its normal operation independently.

Although the aforementioned middlewares are widely spread within the robot community, no detailed performance benchmarks have been carried out yet. This is quite surprising since the performance of a middleware can have a significant impact on the overall performance of the robotic system. In [4] many robot frameworks including their provided algorithms for navigation and mapping are compared with each other also in terms of CPU and memory usage. However, in those tests complete robotic applications are benchmarked, that contain all necessary algorithms for navigation. Consequently, the obtained results reflect the performance of the navigation algorithms and not of the middlewares themselves.

Due to this lack of information, we present an elaborate performance benchmark in section V of this paper.

III. Overview of MIRA

Keeping the weaknesses of the aforementioned middlewares and frameworks in mind and taking our long lasting experience with our previously used architecture [13] into account, we developed an alternative middleware for robotic applications (MIRA) that is designed to be used in real-world applications as well as for research and teaching. Therefore, we pursued the following design goals some of which are also mentioned in [14]:

1) *High performance* and low latency: the communication techniques must have a low CPU footprint to be able to use the middleware on robots with low computational power and to leave enough capacity for the actual algorithms.

2) *Easy to learn and use*: for this reason, all functionality should be realized by using features of the designated programming language (C++, Java, etc.) only. In contrast to ROS, no additional interface description meta-language is necessary.

3) *Small number of different concepts* that are used for a large number of different features to assure ease of use, e.g. our serialization concept that is described below.

4) *High usability*: the features of the middleware must be intuitive and accessible with little code and configuration overhead.

5) *Foolproof*: programming mistakes like type mismatches when using the communication techniques should be reported by the compiler or at run-time instead of leading to unexpected results.

6) *Robust and reliable*: the MIRA software system must not contain a central component that - in case of failure - might stop the entire system from working. Moreover, a high software quality is assured by strict software reviews and by many automated test cases.

Similar to ROS, MIRA supports fully distributed applications, i.e. the application can consist of several different processes that can even be located on different machines. Each process runs a *MIRA Framework* that provides all functionality of the middleware. In each such process, one or more software modules can be located (see Fig.1). Each module is called a *unit*. A unit usually implements the necessary algorithms to solve a certain task, e.g. robot navigation or person tracking, hence a unit is comparable to a ROS-Node/Nodelet. In contrast to ROS, however, units can be freely placed with other units in any process at run-time without any code changes. If several units are located within one process, each unit runs in its own thread. MIRA will automatically take care of multi-threading and data synchronization.

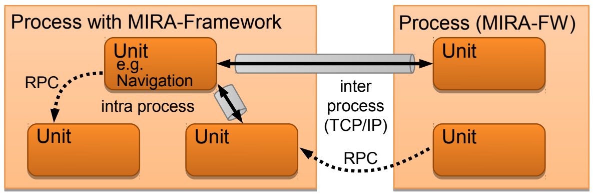

Fig. 1. In MIRA software modules (units) can be distributed across different processes, where each process runs a MIRA Framework that provides the middleware. MIRA automatically chooses methods for intra- or inter-process communication that are fully transparent to the units.

MIRA itself is written in C++, but it is designed to be interoperable with other programming languages like Java, Python and others.

A. Serialization and Reflection

A central concept of MIRA that is used in many different use cases is serialization and reflection. *Reflection* is also known from higher level programming languages like Java and C#. It allows both to retrieve information on the structures of the program at run-time and to query the names and types of variables, and methods of classes. This information is used by MIRA to realize *serialization*. This is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be ”resurrected” (deserialized) later in the same or another computer environment. Unfortunately, C++ does not support reflection and serialization natively. For this reason, MIRA provides an easy to use mechanism that adds both features. An arbitrary class can be reflected and serialized by adding a reflect method as in the following example:

```
class Foo {
  int value;
  Bar* ptr;

  template<typename Reflector>
  void reflect(Reflector& r) {
    r.member("Value",   value, "An int member");
    r.member("Pointer", ptr, "polymorphic pointer");
  }
};
```

This mechanism is used for efficient serialization of the transported data when inter-process communication is used, for parameter marshalling in remote procedure calls, persistence, i.e. storing and loading of states of objects, real-time modification of the parameters of algorithms for easy online “parameter tuning” and many more. If a class again uses non trivial members they are reflected recursively. This enables users of MIRA to compose, store and transmit complex objects, objects with pointers and, by the use of a class factory, even instances of polymorphic classes. It allows to use the complete object-oriented programming paradigm all over the whole distributed application, which offers the realization of completely new ideas: beside simple data now very complex objects like robot-models, GUI-components or even software modules can be transported to the remote side where they add new functionality. That means, beside simple data, functionalities can now also be transmitted. In other middlewares, object orientation always ends at the point when data is transmitted and where they have to fall back to PODs that are restricted to carry plain data only.

B. *Interoperability*

The use of the serialization technique makes it easy to add interoperability with other programming languages and middlewares. At the moment the serialization formats XML, JSON and binary are supported. Through JSON MIRA offers connectors to Python and JavaScript. In order to make the variety of algorithms provided by the ROS community accessible and to allow one to use our middleware without the need for porting existing software an adapter to ROS's topics is available using serialization.

IV. Communication Mechanisms

As already stated before, the major role of a middleware is providing communication mechanisms. For this purpose, MIRA offers two different techniques: message passing and remote procedure calls (RPC). Both will be described in this section. MIRA will automatically choose high performance intra-process communication if the communicating units are located within the same process and inter-process communication via TCP/IP if the units are distributed across different processes (see Fig.1).

The details of the underlying communication technique are fully transparent for the units. Hence, no changes in the code are necessary if a unit is used together with other units in a single process or in a distributed application. This is a big advantage in comparison to ROS which differentiates between *nodes* and *nodelets*.

In contrast to ROS, Urbi and other existing robot middlewares, MIRA does not need a central server for name lookup and other management tasks. It is fully decentralized, and the different MIRA processes connect to each other in a peer-to-peer architecture. Hence, there is no single point of failure. This is an important advantage in terms of robustness and reliability.

A. *Message Passing and Channels*

Units can communicate with each other and exchange messages and data via named *channels*. Channels have globally unique names and are strongly typed, i.e. when storing data in the channels the type information will be maintained. If a user accidentally mixes types, this will be reported by the compiler or at run-time and will not lead to unexpected results. A unit that provides information and data can publish it to a channel. On the other hand, units that are interested in a certain kind of data can subscribe to the corresponding channel using either polling or automatic notification via callbacks whenever new data is available. There can be one or multiple publishers and subscribers for the same channel. Hence, channels allow one-to-one, one-to-many and many-to-many connections.

When multiple publishers and subscribers access the same data, MIRA will automatically take care of concurrent multithreaded access. For optimal performance, we developed a mechanism that avoids unnecessary copying of data and blocking whenever accessing the data for reading or writing. Therefore, each channel is realized as a queue of data. Each queue entry is called *slot*. It is large enough to store one element of the data. The slot concept allows units to access the same channel for reading and writing without interfering each other.

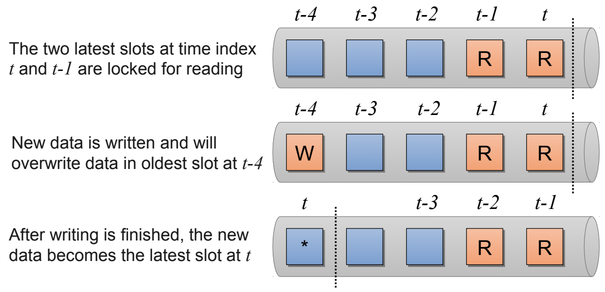
t-4
t-3
t-2
t-1
t
The two latest slots at time index t and t-1 are locked for reading
R
R
New data is written and will overwrite data in oldest slot at t-4
W
R
R
After writing is finished, the new data becomes the latest slot at t
*
R
R

Fig. 2. Each channel consists of a queue of slots. In the upper row, the two newest slots are locked for reading. In the second row, new data is written. Therefore, the oldest slot is recycled and replaced by the new data. In the third row, the newly written data now becomes the newest slot.

If a unit wants to read data from a channel, the MIRA Framework will return a read-accessor to the most current slot. Additionally, that slot will be read-locked for concurrent access, i.e. read access is allowed for other units whereas write access is forbidden (first row in Fig. 2). If a unit wants to publish data to that channel, the MIRA Framework looks for a non-blocked slot first of all, i.e. a slot where no other unit is reading from or writing to (second row in Fig. 2). This slot will be locked exclusively, and a write-accessor is returned that allows the unit to write its data directly into the slot. To avoid unnecessary memory allocation, the memory of a slot can be reused if the size of the data does not change. A unit that e.g. captures data from a camera can store the image data directly and thus, no data is copied unnecessarily. If no free slot is available for writing, a new slot is created and the channels queue will grow. After finishing writing, the slot becomes the newest element for the channel and can be read by other units (third row in Fig. 2). This concept completely avoids blocking of the publisher of the data even if other readers are still consuming. Vice versa a reader will always be able to obtain data for reading without blocking. It is essential in the use case of a single fast producer and many or even slow consumers. Consumers can do time consuming operations on the data without the need for copying it and thus avoid blocking of the producer. Obviously, the described slot-concept seems to lead to an additional memory consumption for holding multiple instances of the data in the slots. In the worst case, there will be as many slots as readers and writers. However, in other middlewares that do not use such a slot-concept, as for example ROS, each reader or writer holds its own copy of the data anyway, leading to exactly the same memory consumption. Consequently, in comparison to other concepts, our concept does not pose any additional memory consumption overhead. In practice, it even reduces the memory usage: in typical scenarios with one publisher and two or more readers only two slots are needed since several readers can share a single data slot.

Another advantage of this slot-concept is achieved by keeping the slots of the channel's queue in the correct temporal order. This is done by using the queue as a ring buffer, where the oldest element is replaced if new data is written (second row in Fig. 2). This allows read access to data from the past or even reading whole intervals of continuous data. This is an essential feature for transmitting audio data where it is important that no audio frame is dropped. Moreover, such a history allows to interpolate or filter data since the filter or interpolation function can access multiple consecutive values. Currently, we support linear- and cubic-interpolation as well as a moving average-filter that, for instance, allows to compute the average of the channel's data within the last second.

B. Remote Procedure Calls

Remote procedure call (RPC) is a concept for invoking procedures and methods from remote as if it were local ones. It is a form of inter-process communication. Our approach for RPC can be used for both inter- as well as intra-process execution of a unit's subroutines. Each unit exposes some of its functions as a named service available for RPC to the framework. Most RPC systems make use of an interface description language (IDL) to describe the layout of the methods. In others, the methods must follow a special syntax or have special types for parameters and return values. In MIRA, return values and parameters are strongly typed as we use our serialization concept for marshalling them. It allows to publish any global or class member function with an arbitrary number of parameters. The only requirement is that the return value and the parameters use serializable data types. This has the advantage that the user can exploit existing methods to the RPC system later with a single line of code. Registration of methods is done analogue to the serialization of members in the reflect method.

```
class Foo {
  template<typename Reflector>
  void reflect(Reflector& r) {
    r.interface("IAverage");
    r.method("computeAverage", &Foo::computeAverage, this,
             "Computes the average of all vector elements");
  }
  float computeAverage(const std::vector<float> data) {
    ...
    return sum / data.size();
  }
};
```

For obtaining the returning value of a remote procedure, MIRA utilizes *futures* that act as a proxy for the result of the asynchronous call. The use of futures dramatically reduces latency for RPC, since the caller can decide whether to wait for the result or to query for it later on. For example, one can invoke a call to a remote procedure which may take some time to complete. In the meantime, one can process or compute something else to use the time efficiently while waiting for the result of the call.

```
std::vector<float> data;
data.push_back(1.0f);
data.push_back(2.0f);
std::string service = waitForServiceInterface("IAverage");
RPCFuture<float> future = callService<float>(service,
                                   "computeAverage", data);
// do some time consuming computation here
...
// now obtain the result of the call by blocking
// until it is available
float result = future.get();
...
```

We have also enhanced the functionality of standard RPC facilities by object-oriented features. Units can implement interfaces so that other units may query for a list of services that provide a certain interface without knowing the name of the service provider. Interfaces enable code reuse, i.e. a navigation algorithm can control a variety of different robotic drivers that implement an "IRobot" interface. Units can even wait for a required interface to become available. Additionally, we support exposing of virtual member methods that can be reimplemented in derived classes.

Another feature of RPC in MIRA is *exception passing*. That is, errors and exceptions that occur on the service side will be transported back to the caller. When the caller tries to access the return value of the call, an exception will be thrown that contains the marshalled error. Here again, MIRA only relies on built-in features of the C++ language.

As mentioned before, calling service methods is done by serializing passed parameters and return values. This allows for using fast binary serialization as well as calls with JSON-RPC encoded messages, e.g. via Javascript over a website.

V. Results

In the following, we compare the performance of MIRA with several other commonly used robot middlewares ROS, Yarp, Urbi, Player, LCM and MOOS. Thereby, we concentrate on the performance evaluation of the communication techniques only. In general, we analyze the communication overhead that is imposed by the middlewares.

Since each middleware offers slightly different concepts for communication, the performance of those concepts cannot be compared directly with each other. Instead, we defined simple scenarios that were implemented for each middleware using the available techniques that are provided by the middleware. The scenarios also define the performance metrics that are evaluated, e.g. delay, latency, memory usage, etc. The source code for all benchmarks is written in C++ and adapted to each tested middleware. Moreover, it was compiled with full compiler optimizations for all middlewares. It can be downloaded at `http://www.mira-project.org/` for own evaluations.

All benchmarks were carried out on the a single machine with an Intel Core i7-2620M, 2.70GHz processor and 4GB of RAM. For timing measurements we used the "RDTSC"-instruction to access the Time Stamp Counter of modern processors. It counts the number of CPU cycles since system start and, therefore, has an extremely high resolution of less than one nanosecond on the testing machine. In the timing results below, we have converted the measured CPU cycles into milliseconds.

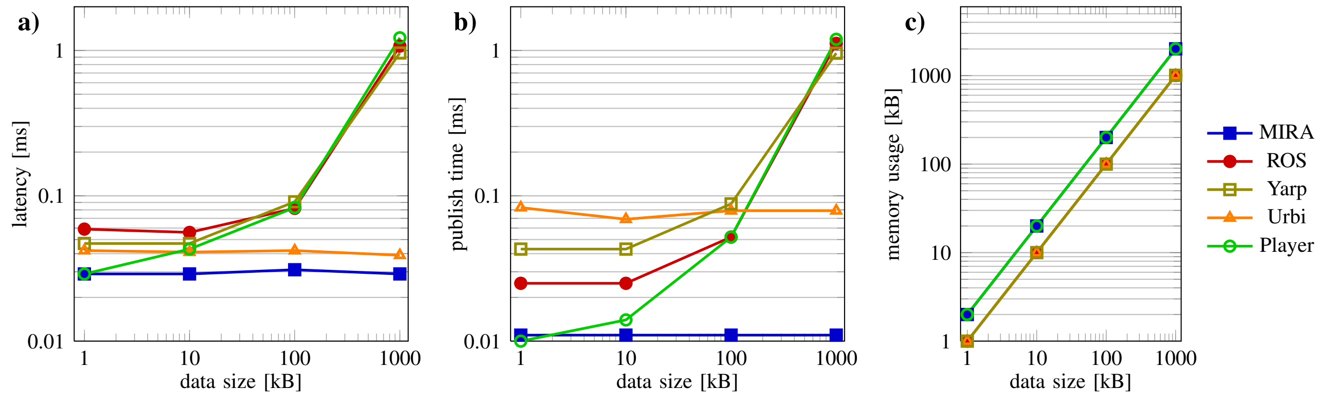

Fig. 3. Benchmark results for *intra-process communication* where publisher and subscriber are located within the same process. **a)** Latency for transmitting data depending on the size of the transmitted data. **b)** Time for publishing data depending on the size of the transmitted data. **c)** Memory usage caused by data transmission depending on the size of the data.

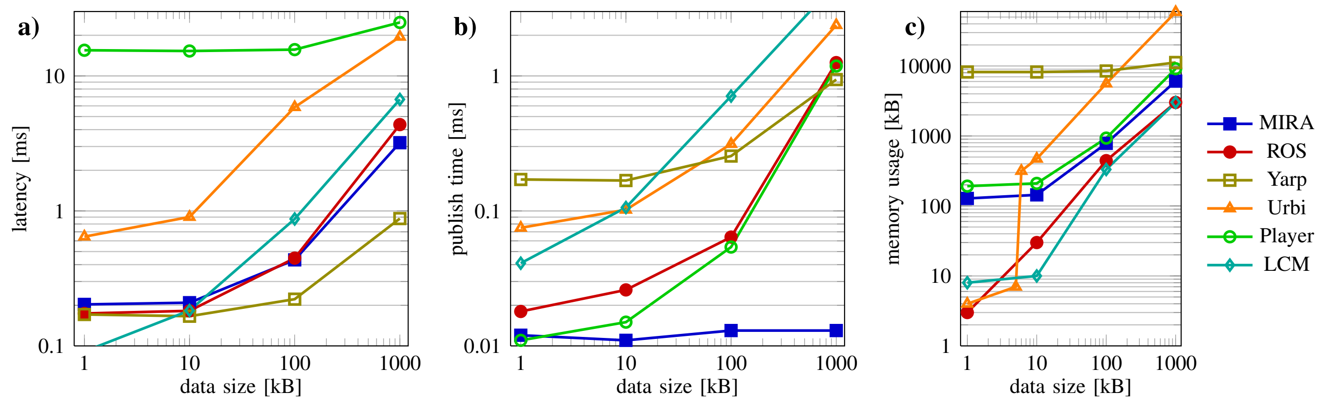

Fig. 4. Benchmark results for *inter-process communication* where publisher and subscriber are located within different processes. **a)** Latency for transmitting data depending on the size of the transmitted data. **b)** Time for publishing data depending on the size of the transmitted data. **c)** Memory usage caused by data transmission depending on the size of the data.

A. Intra-Process Communication

The first scenario consists of a publisher that sends data to a subscriber. Both the publisher and the subscriber are located within the same process. The scenario was executed several times - each time with a different size of the transmitted data. The size was varied within a range from 1 kB to 1000 kB. In practice, a size of 1 kB corresponds to small objects like odometry data, while a size of 1000 kB corresponds to 640×480 color images. Therefore, both are typical sizes for data that need to be transmitted by robot middlewares. To achieve reliable measurements we averaged the benchmark results of 100 consecutive transmissions which were sent with a rate of 10 transmissions per second.

1) Latency between Publisher and Subscriber: As first performance metric the time duration was measured from the moment when the data was sent by the publisher until it was received by the subscriber. For a robotic middleware this latency should be small to enable highly reactive applications. Moreover, the latency is an indicator for the CPU usage of the transmission as it contains the time needed for the serialization, transport and potential copying of the data - operations that are very CPU intensive.

The left plot in Fig.3a shows the latency depending on the size of the transmitted data for the different middlewares. Please note that logarithmic scales are used in those plots. MIRA and Urbi have a constantly small latency that is independent from the data size. The latency of MIRA on average is 0.03 ms, which is slightly lower than Urbi's latency of 0.04 ms. In contrast, the latency of ROS, Yarp and Player increases with the size of the transmitted data. In Player this is caused by copying the data.

For ROS and Yarp on the other hand, one would not expect the latency to increase with the size of the data at first glance, since both simply share the memory between the publisher and the subscriber. However, in practice the publisher has to make sure to not interfere with the subscriber. Therefore, the publisher has to create a new shared memory block whenever it wants to write new data, since the subscriber may still be reading from the previous shared data. This is common practice in most ROS nodelet implementations. Therefore, we treat this implementation detail as necessary part of the data transmission and had to take it into account in our benchmarks, where we allocate new memory before the transmission of the data. Consequently, for large amounts of data, allocating the memory results in a large performance penalty for ROS as shown in Fig.3a. For Yarp the same technique has to be applied when intra-process communication is used. For large amounts of data, ROS, Yarp and Player are up to 40 times slower than MIRA and Urbi due to the allocation or copying overhead.

2) Overhead for publishing data: In addition, we have measured the time a software module blocks when it is publishing data. For a publisher this time should be as small as possible to avoid performance penalties, especially when data is written with high frequencies.

Fig.3b shows the time that is needed for publishing data depending on the size of data that is written. In MIRA publishing data constantly takes about 0.01 ms, i.e. a software module is able to publish data with a rate of up to 100 kHz. This is significantly faster than the other middlewares. For ROS, Yarp and Player, the time again increases with the size of the transmitted data, caused by the memory allocation (ROS and Yarp) or memory copying (Player). On the other hand, when using Urbi, a publisher blocks for a constant amount of time, similar to MIRA. However, the time is significantly higher. This is a result of the non-copying mode, where the publisher has to wait until all subscribers have finished processing the data.

B. Inter-Process Communication

In a second scenario, we tested the inter-process performance. In contrast to the first scenario, the publisher and the subscriber were now located in two different processes on the same machine. The results are shown in Fig.4.

1) Latency between Publisher and Subscriber: Fig.4a again shows the latency for the transmission of a data packet. It is not surprising that the latency is higher for all middlewares as inter-process communication creates more overhead. The inter-process performance of MIRA and ROS is almost identical. Both are significantly faster than Urbi and Player.

The graph of the MOOS middleware is not included in the diagrams, as its performance is very poor. With 100 ms the latency of MOOS is 100 times higher compared to MIRA. Beside the slow string-based data exchange, this is caused by the low frequency that is used for fetching the data from the central database, although this frequency was already set to the maximum of 200 Hz in our tests. It is questionable if such a middleware is suitable for realtime robotic applications, that need to respond quickly.

Beside MOOS, Player also has a poor inter-process performance since it does not support this kind of communication natively. Instead, it requires the use of a special 'pass through'-device which results in significant overhead.

In contrast, Yarp shows an extremely good inter-process performance and achieves even smaller latencies than ROS and MIRA for larger amounts of data although all three middlewares use TCP for the inter-process data transport.

The LCM middleware also performs well, especially for small amounts of data. For larger amounts of data its latency becomes up to two times larger than that of MIRA and ROS. This is caused by the used UDP multicast, which seems to perform worse on a single Linux machine compared to TCP.

In general, the inter-process communication is about 10 times slower than the intra-process communication for all middlewares that support both, especially when large amounts of data are transferred. This needs to be taken into account when images and other large data packets are exchanged between software modules.

2) Overhead for publishing data: As shown in Fig.4b, in ROS, Yarp, Player, Urbi and LCM the blocking time increases with the size of the transmitted data when publishing data. This is because these middlewares block until the whole data is transferred into the transportation buffer (TCP stack, etc). However, in MIRA the publishing overhead remains constantly low. This is a result of the employed slot-concept which ensures non-blocking access for reading and writing. The actual transfer into the TCP stack is done by a separate thread. Therefore, in MIRA the communication stays fully transparent no matter if inter-process or intra-process communication is used.

C. Memory Consumption

In the above tests, we additionally measured the memory usage during the transmissions. Measuring the total memory consumption of a process is difficult as it contains the code size of shared library and the process itself and would, therefore, lead to incomparable results. For this reason, we first obtain the memory usage of each process before starting the transmission and again afterward. The difference of both values yields the additional memory usage that is solely caused by the data transmission. Again, these measurements were made for different sizes of the transferred data packets and for intra-process communication (Fig.3c) and inter-process communication (Fig.4c). In the latter case, we computed the memory usage for both involved processes and added them.

When using intra-process communication, ROS, Yarp and Urbi use exactly the amount of memory that is allocated by the data packet. There is almost no additional overhead. MIRA and Player require two times more memory here. In MIRA this is caused by the slot-concept which allocates memory for two data packets. In Player the additional memory is required since there is a data buffer for both the publisher and the subscriber, where the data is copied to.

When using inter-process communication, the memory overhead is significantly higher for all middlewares. ROS and LCM have the lowest memory consumption followed by MIRA, Player, Urbi and Yarp.

Interestingly, the memory usage of ROS, LCM and Urbi is significantly lower than that of MIRA, Yarp, and Player as long as small amounts of data are transferred. Apparently, MIRA, Yarp and Player have a minimum size for their transportation buffers that create some overhead in these cases. This constant overhead is the highest for Yarp. On the other hand, when using large data packets the memory overhead of Urbi rises dramatically, up to 6-10 times more than the other middlewares.

Nevertheless, the general memory usage characteristics are similar for all middlewares and none of them really wastes memory. Moreover, on current architectures memory usually is not the limiting resource.

D. Latency of Remote Procedure Calls

In a third scenario, we compared the performance of remote procedure calls. Therefore, one software module offers a service that is called by another module. To measure the influence of the amount of data that is transported together with the RPC call, the service method takes a single parameter where the data is passed. Again the latency is evaluated, i.e. the time that elapses from sending the RPC call until the invocation of the service method is measured.

Due to the limited or missing RPC support of Urbi, Player, Yarp and LCM, the benchmark can be performed for ROS and MIRA only. Fig.5 shows the latencies of RPC calls for both middlewares.

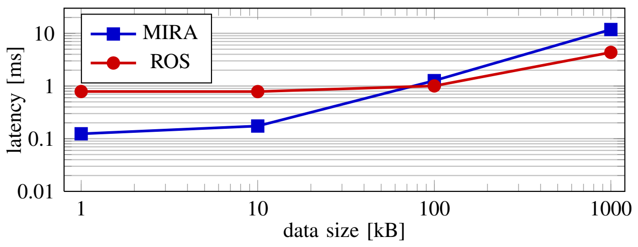

Fig. 5. Latency when performing a remote procedure call depending on the size of the transmitted data.

For RPC calls where a small amount of data is transferred, the latency of MIRA is similar to using channels for inter-process communication. When using ROS on the other hand, RPC calls are significantly slower compared to the other communication technique using ROS topics. In a direct comparison with MIRA, the latency of ROS RPC calls is 7 times higher if only a few bytes or kilobytes are transmitted. However, with an increasing size of data, this difference is getting smaller. For very large amounts of data the latency of ROS even becomes smaller than that of MIRA. This drawback is a result of unnecessary data copying within MIRA's RPC framework and will be fixed in the next release. Nevertheless, RPC calls are usually not used to transport huge amounts of data. Channels should be preferred for this purpose.

VI. Conclusions and Future Work

In this paper, we presented a new middleware for robotic applications. Although many middlewares already exist for this purpose, most of them apply similar communication techniques that lead to the same disadvantages. With our slot-based communication technique, we presented a novel, very efficient approach that avoids blocking and unnecessary copying of data when concurrent read and write access comes into play. Using this technique, our middleware is able to outperform other middlewares that are regarded as state of the art. The high performance and the fully decentralized and therefore reliable architecture allows to apply MIRA in real-world applications. While other middlewares are strongly focused on the transport of plain data, MIRA's serialization concept allows to transport complex objects as easily as normal data and, therefore, allows to apply the object-oriented programming paradigm within the whole distributed application.

This enables new techniques like the migration of units at run-time. Depending on the workload within the distributed application, units can move from one process to another or even to a different machine. This offers completely new possibilities for robotic applications and even self-organized applications. The current implementation already supports migration, and we want to extend this feature within the next releases.

Its performance and the new features mentioned in this paper make MIRA an valuable alternative to the existing middlewares. To increase the distribution of MIRA within the robotic community, it is available as open source for download at `http://www.mira-project.org/`.

Another main contribution of this paper is the first ever performance comparison of the most commonly used middlewares ROS, Player and Urbi. On the one hand the results of these benchmarks can assist developers and designers of robotic applications to choose the right middleware depending on their needs, and on the other hand it may help to improve the performance of the overall application by choosing the right design decisions, e.g. by using intra-process communication where applicable.

Acknowledgements

We thank all our colleagues for their contributions, for implementing certain parts of MIRA, for their feedback and of course for the various code reviews, especially Michael Volkhardt, Christian Vollmer, Steffen Müller, Christoph Weinrich and Christof Schröter.

References

[1] H.-M. Gross, H.-J. Böhme, C. Schröter, S. Müller, A. König, E. Einhorn, C. Martin, M. Merten, and A. Bley, "Interactive Shopping Guide Robots in Everyday Use - Final Implementation and Experiences from Long-term Field Trials," in *Proc. IEEE/RJS International Conference on Intelligent Robots and Systems (IROS)*, pp. 2005–2012, 2009.

[2] H.-M. Gross, C. Schröter, S. Müller, M. Volkhardt, E. Einhorn, A. Bley, C. Martin, T. Langner, and M. Merten, "Progress in Developing a Socially Assistive Mobile Home Robot Companion for the Elderly with Mild Cognitive Impairment," in *Proc. of the IROS*, pp. 2430–2437, 2011.

[3] F. Beck and S. Diehl, "On the Congruence of Modularity and Code Coupling," in *Proc. of the 19th ACM SIGSOFT symposium and the 13th European Conf. on Foundations of Software Engineering*, pp. 354–364, 2011.

[4] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[5] A. Elkady and T. M. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, 2012.

[6] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit," in *Proc. of the IROS*, pp. 2436–2441, 2003.

[7] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *Proc. of the 11th Int. Conf. on Advanced Robotics*, pp. 317–323, 2003.

[8] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[9] `www.urbiforge.org`.

[10] A. Huang, E. Olson, and D. Moore, "LCM: Lightweight communications and marshalling," in *Proc. of the IROS*, pp. 4057–4062, 2010.

[11] `www.robots.ox.ac.uk/ pnewman/TheMOOS/`.

[12] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet another robot platform," *Int. Journal on Advanced Robotics Systems*, pp. 43–48, 2006.

[13] C. Martin, A. Scheidig, T. Wilhelm, C. Schröter, H.-J. Böhme, and H.-M. Gross, "A New Control Architecture for Mobile Interaction Robots.," in *Proc. of the ECMR*, pp. 224–229, 2005.

[14] W. D. Smart, "Is a Common Middleware for Robotics Possible?," in *Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, 2007.