A Middleware for Collaborative Research in Experimental Robotics

Johannes Wienke Sebastian Wrede

Abstract— **This paper presents the Robotics Service Bus (RSB), a new message-oriented, event-driven middleware based on a logically unified bus with hierarchical structure. Major goals for the development of RSB were openness and scalability in order to integrate diverse components in the context of robotics and intelligent systems. This includes the ability to operate on embedded platforms as well as desktop computers, reduction of framework lock-in, and the integration with other middlewares. We describe the design of the RSB middleware and explain how it meets requirements which lead to a scalable and open middleware concept. These requirements are based on several application scenarios which are used to verify the applicability of RSB. Furthermore, we relate RSB to other middlewares in the robotics domain.**

I. Introduction and Motivation

The *Robotics Service Bus* (RSB) is a new lightweight and flexible middleware, developed in the context of robotics and intelligent systems. The development has been driven by three major goals: *a*) Integration of heterogeneous existing and future systems and components on diverse (and sometimes restricted) platforms in a robotics-centric research environment. *b*) External integration in the sense of integrating components designed for other frameworks into RSB-based systems and exposing components designed for RSB into systems using other frameworks in order to foster reuse of components. *c*) Facilitation of system-level research (e.g. fault detection, reverse engineering) and empirical experiments (data set recording for HRI studies or evaluation of algorithms) in this environment by providing introspection support [1]. RSB contributes to robotics and intelligent systems research by facilitating the integration of otherwise separated systems. The results are generic interfaces for system-level development and research.

A. Requirements

The aforementioned goals give rise to a number of requirements a suitable middleware has to fulfill. These requirements are discussed in the following paragraphs.

One class of requirements can be summarized as *openness*. In the context of distributed software systems, openness is the property that services provided by a system adhere to standardized protocols which formalize their syntax and semantics [2]. According to this definition, openness of a system leads to the following desirable qualities:

Portability The property of being able to function in different execution environments without modification. Examples include different hardware and software platforms and their respective constraints regarding available (parallel) processing, memory and network resources. As a consequence, aspects like acceptable dependencies or the use of threads may be severely restricted.

Flexibility The ease with which the structure of a software system can be changed, e.g., by adding new components or altering behaviors of system parts.

Interoperability The ability to function in conjunction with other systems designed for the same domain. In the context of middleware, one enabler for interoperability is the existence of common data types, as they form the basis of every communication. Ultimately, components written for one middleware can be used in another middleware if both have sufficient interoperability qualities.

Another important requirement is *scalability*. In the context of middleware we consider the following aspects of scalability important (derived from [2]):

System size and distribution the system size and type which can be handled by a middleware. E.g. components in a single process, multiple processes on a single node or on distributed nodes in a network. This includes the number of components as well as their platform and runtime context. Scalable systems foster the integration of components. The ability to scale the size of a system implies sufficient efficiency in its processing.

Organization The amount to which a system can be administered, which is developed by several organizational structures with potentially overlapping or conflicting aims and guidelines.

The goal of enabling system-level research and data recording can only partially be fulfilled by implementing the requirements collected so far. Therefore, introspection support, the ability to access and analyze the state and processes of a deployed system, and the availability of sufficient relevant meta-data have to be considered as an additional requirement. This property also improves the usability of a middleware for a component developer by making the communications in a system traceable.

B. Why Yet Another Middleware?

Why yet another middleware is needed to fulfill the requirements mentioned above? Especially having in mind that several huge communities evolved recently for frameworks like ROS [3], YARP [4], OpenRTM [5] or Orocos [6].

This work was partially funded by the European FP7 project HUMAVIPS, grant no. 247525.

J. Wienke and S. Wrede are with Research Institute for Cognition and Robotics (CoR-Lab), Bielefeld University, 33615 Bielefeld, Germany {jwienke, swrede}@cor-lab.uni-bielefeld.de

This research was partially supported by the German Research Foundation (DFG), Center of Excellence “Cognitive Interaction Technology” (CITEC, EC 277), Bielefeld University, Germany.

Many thanks to Jan Moringen, Arne Nordman and Michael Götting for contributing to RSB and its concepts.

First, our evaluation of existing middlewares showed that the frameworks of our knowledge do not have sufficient portability across different platforms while providing the intended high-level functionality. Especially, existing systems often expose a dependency graph with multiple libraries, sometimes hard to (cross-) compile. In our research context with the constant development of new platforms, easy portability to them is a key aspect.

Another aspect we found for many middlewares which provide sufficient high-level functionality is that their implementation results in a higher level of *framework lock-in* than desired for our systems. We define framework lock-in as the amount to which client code is coupled (technically and conceptually) to the chosen framework. Reasons for this include unchangeable communication patterns and transport mechanisms, artificial framework data types which do not reflect the client domain but deeply spread into client code or a build system hard to replace with less coupled solutions.

We think that a new middleware or at least the extension of existing middlewares is required to address all of the aforementioned issues. However, we also specifically address the issue of integrating with other solutions as stated by the openness definition. Last but not least, while being a completely new implementation, the ideas of RSB are not entirely new, but instead are based on experiences with our previous middleware XCF [7] and its information-driven integration [8] principles.

The remainder of this paper is organized as follows: After describing prototypical application scenarios of RSB in Sec. II we will continue with a description of RSB's architecture in Sec. III. Afterwards, selected concepts of RSB are discussed and evaluated along use cases from the previously mentioned scenarios in Sec. IV. Finally, Sec. V compares RSB with related middlewares.

II. Application Scenarios

This section introduces a number of projects in which RSB is already applied and which guided the development of RSB. Therefore, specific requirements of each application scenario are related to the generic ones outlined above. The different applications especially outline the bandwidth of distribution types and system sizes for which RSB is currently used.

In the FP7 EU project AMARSi[1] we are developing a hierarchical motor control architecture based on dynamical systems components. The resulting architecture needs flexible deployment as some of the components run in a single control process on the embedded system of a new compliant quadruped robot [9]. Others are deployed remotely on dedicated workstations. Furthermore, the currently used RoBoard RB 110[2] requires high efficiency and minimal overhead as it features a single core CPU and threading has a notable impact on performance.

Another platform on which RSB is used and which has been the inception for developing the approach is the NAO humanoid robot [10]. For experimental research, e.g. in the context of the FP7 EU project HUMAVIPS[3], an alternative approach for system integration was needed than what was provided by the NAO SDK (see Sec. V for a more detailed discussion of NaoQi).

Fundamentally, RSB offers remote access to NAO's sensors and actuators for one or more remote computers. In this context, efficiency is again a requirement with the 500 MHz CPU of NAO as well as the limited network bandwidth, especially for video streaming. To achieve the optimal system performance, these limitations require the ability to easily change the deployment scheme for components from remote computation to on-board computation on the robot and the other way around. As deployment to the robot requires cross-compilation, RSB and its dependency graph should not complicate this process. Furthermore, existing functionality of the robot's SDK (e.g. inverse kinematics or the walking algorithm) is imported into the RSB-based architecture, resulting in a use case for middleware-compatibility. Another important aspect of the activities with NAO is the ability to generate complete data sets for the training and evaluation of algorithms as well as the analysis of HRI topics. RSB's introspection support is used in this context to generate data sets without needing to program recording components for this specific use case. Recording includes external devices like cameras to observe the scene as well as the system internals. Complete data sets require the availability of sufficient meta-data, e.g. to synchronize different streams (i.e. sensors and processing results). Moreover, the manual annotation process should be facilitated, e.g. through a common storage format which reduces the need to use many different tools. Replay of recorded data should be possible with the same middleware interface in order to later analyze and improve the behavior of single components without interface modifications for the replay trials.

On a larger deployment scale, RSB is used for teaching activities at Bielefeld University and provides the technological basis for a course in which about 25 students in the Intelligent Systems Master program implement new functionality for a smart-room environment. Besides the diversity of integrated applications in this context, the opportunity to empirically asses the usability of the framework from a developer perspective is given.

RSB is also applied in an online learning scenario where goal-babbling is used to learn a kinematic model of a novel robotic manipulator [11], the Festo Bionic Handling Assistant. From an architectural perspective, a particular requirement in this scenario is the online integration of ground truth data acquired through a motion tracking system (Vicon Nexus [12]) with low latency and high frequency. For instance, endeffector positions as input for the feedback controller are streamed with 200 Hz while in parallel CAN Bus control commands and joint information as outputs of the controller are streamed with a frequency of 100 Hz.

[1]http://www.armarsi-project.eu
[2]http://www.roboard.com/RB-110.htm
[3]http://www.humavips-project.eu

III. Architecture and Implementation

In essence, RSB[4] can be described as a message-oriented, event-driven middleware for research systems based on a logically unified bus. As a result, the native communication semantic is $m : n$ broadcast. However, other communication patterns based on this structure can be added. An asynchronous request/reply pattern is already provided with RSB.

The basic functionality required for communicating in a distributed system is contained in a concise core layer. This core currently exists as full implementations in C++, Java, Python and Common Lisp with the aim to provide a natural interface in each language. Following this principle, dependency footprints of the core layer implementations are as small as possible and Linux, Mac and Windows are supported.

The following description of the architectural concepts of the RSB core is structured along three topics. First, the basic event structure will be explained, second, the notification model which transports events to participants is described, and third, the observation model specifies how participants receive relevant events. For the description, we adopt the definitions of [13], in which an event is *a detectable condition that can trigger a notification*. A notification, in turn, is *an event-triggered signal sent to a run-time defined recipient*. The explanations are supplemented by Fig. 1 which gives an overview of the high-level concepts implemented in RSB.

A. Event Model

An event is the basic unit of exchanged data in RSB. Hence, all information required to fully specify and trace the condition it represents need to be present in the event. To fulfill these requirements, our event model consists of the following components:

Payload The payload of an event is a user-defined object of the respective programming language which contains the major information specifying the condition the event represents. It can be of an arbitrary domain type which reduces the framework lock-in by means of an early transition from framework types to domain objects (cf. III-B for technical realization).

ID A unique ID for each event in an RSB-based system to make events addressable and foster traceability.

Meta Data Each event is supplemented by meta data. It consist of the event sender's ID and several timestamps that *a*) specify timing information relevant to the condition represented by the event (user-extensible) *b*) make the the processing of the event within RSB traceable. Besides these framework-supplied items, a key-value store for string-based additional meta data items is available for the client and user-defined timestamps can be added.

Causal Vector This vector allows to represent the causing events of a given event, as proposed in [14]. It facilitates automatic system analysis and debugging.

[4]RSB is available as open-source software at: `https://code.cor-lab.de/projects/rsb`

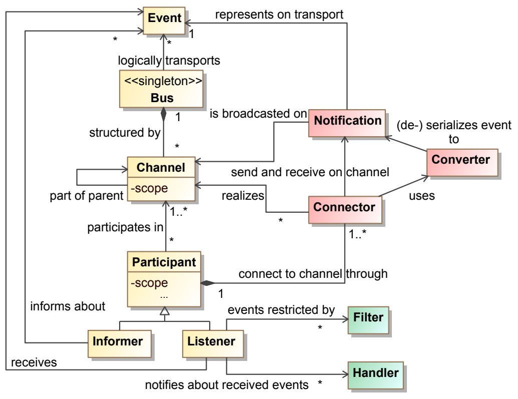

Fig. 1. Conceptual overview of the RSB model. The notification model parts are color coded in yellow and red to distinguish between client level and backend realization. The observation model is depicted in green.

Destination Scope Specifies the recipients of the event notification (cf. Sec. III-B) by restricting the visibility of event notifications [15].

B. Notification Model

The notification model defines how events are communicated in RSB. RSB forms a logically unified bus across different transport mechanisms. Different *Participants* connect to this bus. *Informers* send events, whereas *Listeners* receive events. From a logical perspective, no point-to-point connections are established.

In order to structure the communication via the bus – or stated differently, restrict the visibility of events for participants – RSB utilizes a hierarchical channelization scheme. This scheme is best explained by it's declarative representation as a *Scope*, which is represented in RSB with a hierarchical notation compatible with the path component of URIs [16]. E.g. sending an event with destination scope (cf. III-A) `/robot/camera/left/` will make this event visible in the channels [13] represented by scopes `/robot/camera/left/`, `/robot/camera/`, `/robot/`, and `/`. Consequently, `/` represents a channel where all events of the system are visible. Each participant is associated to one channel, but multiple participants can participate at the same channel ($m : n$ semantics). The yellow parts of Fig. 1 visualize these domain concepts.

The chosen hierarchical channel layout provides benefits for logging purposes and provides a first-class means of the framework to structure the data space, e.g. with subscopes for different services. However, it also increases the chance that a listener receives unexpected data, because a new informer appeared on a sub-scope of the listener's scope. RSB's filter mechanism (cf. Sec III-C) allows clients to efficiently specify which events they expect.

To realize the logically unified bus, several constraints need to be addressed in order to meet our requirements:

- Different transport mechanisms (transports) may be used in one system. E.g. an in-process connection provides high performance for several components,

whereas others are connected through diverse network-based protocols. It is helpful to maintain communication even inside single components using RSB in order to make them introspectable with system-level tools.
- The payload of events is a user-defined programming language object (cf. III-A). Thus, no common (de-) serialization mechanism can be used.
- If different transports are used, they may require different (de-) serialization strategies.
- Different domain objects should create compatible serialization formats to allow standardization and compatibility between different applications and libraries (e.g. a participant implemented in C++ sends OpenCV's images[5] and a receiving participant uses PIL images[6]).
- Depending on the application, client-level code may pose different quality-of-service and performance requirements on the transmission of events.

To address the application of different transports in one system, RSB introduces the concept of *Connectors*. A connector implements sending and receiving of events for a specific transport mechanism. These connectors realize the logically unified bus and each participant uses one or more of these connectors as its backend. Internally, connectors exchange *Notifications*, which are serialized events according to the connector's protocol. This structure is visualized by the red parts in Fig. 1. Currently, three transports are implemented. For complex distributed systems over a network connection, a transport using the Spread Toolkit [17] is available. A more lightweight network-based transport using sockets with TCP communication can be used with less configuration overhead. Finally, an in-process transport implements efficient event exchange without serialization overhead. Connectors can be chosen through a configuration mechanism or programmatically.

Concerning the dynamics of sending events, RSB provides an interface to implement different strategies [18]. For instance this allows to integrate a queuing strategy for temporal decoupling in the case events are generated with a higher frequency than the connector can achieve for sending, but the client code should not block. The default strategy, however, is synchronous. Moreover, clients may specify reliability and ordering requirements for events. These two properties have a strict ordering in a way that e.g. a higher level of reliability includes all aspects of lower levels. For different transport mechanisms it cannot be generally stated where these requirements need to be specified (sending or receiving side). Therefore, they can currently be passed to sending and receiving connectors. Only the minimum specification for each sending and receiving connector pair is the resulting guarantee for the client. E.g. specifying [reliable, ordered] at the sending side and [unreliable, ordered] at the receiving side will effectively result in [unreliable, ordered].

The (de-) serialization of event payloads from/to notifications is the responsibility of *Converters*. RSB comes with a set of pre-defined converters for fundamental data types (payloads of events) like numbers and strings. This set can be extended by the user to integrate user-defined payload types. Converters vary in 3 dimensions and have to be chosen along these aspects: *1*) the serialization format they operate with[7] *2*) the concrete serialization they operate with, given their serialization format[8] *3*) the data type they operate with, e.g. IPL image or PIL images. Given a specific connector, only those converters matching the serialization format of the underlying transport mechanism can be used to (de-) serialize events. However, no clear selection criterion for a certain converter exists which also covers the other two aspects. For instance, the created serialization is a design decision of the overall system and, in the case of deserializing, multiple converters multiple converters can exist which produce different data types. Therefore, RSB uses exchangeable strategies to select converters. Default implementation are provided in cases where only a limited and unambiguous set of converters is registered or user-supplied predicates are sufficient to describe the selection.

To properly initialize participants in RSB with their scope, required connectors and other options, RSB provides a configuration procedure using configuration files and environment variables as well as programmatic access to the configuration. Moreover, the chosen scope syntax is the basis for a URI-based configuration scheme, which provides a useful addition to the aforementioned system. As an example, `spread://localhost:4803/nao/vision/left/` configures a participant using a spread-based connector for scope `/nao/vision/left/` with the daemon running at port 4803.

C. Observation Model

Clients of RSB receive events by registering *Handlers* at listeners. These callbacks are invoked asynchronously and implicitly whenever a new event was received and decoded by a connector. Each listener can have multiple handlers registered.

Besides the scope, which is already defined by the listener, further filtering may be required for clients to receive only the relevant information. Hence, dedicated *Filters* can be installed at each listener to further restrict the set of received events. Several filters are connected as a conjunction, thus the first filter which does not match an event leads to a complete rejection of the event. Filter of one listener affect all registered handlers. RSB implements the concept of content-based [13], client-side filtering, where filters are explicitly allowed to inspect the user-defined payload of events. Users can provide additional filters according to their requirements. As a default, the payload of the event is first deserialized using an appropriate converter and afterwards filtered. However, this may result in a high computational load on the

[5]`http://opencv.willowgarage.com/wiki/`
[6]`http://www.pythonware.com/products/pil/`

[7]E.g. socket-based communication can be based on binary encodings, whereas a SOAP-based transport usually requires XML serializations

[8]I.e. the byte layout for a binary encoding or the XML schema. This information has to be carried in each notification since channels are not restricted to a particular serialization.

```
message FaceDetection {
    message FaceBox {
        required BoundingBox box = 1;
        required double confidence = 2;
    }
    repeated FaceBox detections = 1;
}
```

Listing 1. An exemplary Protocol Buffers-based data type definition for face detection results.

receiving side without visible actions for RSB clients if many events are filtered based on their content. To mitigate the impact of client-side filtering, connectors of each listener are informed about the attached filters. Each connector can implement selected filters already at the transport level before decoding an event from a notification. This allows a flexible optimization. Imagine e.g. the case that a transport encodes the data type of the user payload outside of the serialized notifications. In this case the transport can implement a type filter even before decoding the notification. A static view of the relevant concepts is visualized with green color in Fig. 1.

From a dynamic perspective, RSB includes an interface to implement strategies that dispatch received events to registered handlers of a listener. These strategies also apply the filters (except those optimized by connectors). The default strategy uses a small, fixed-size thread pool to serve all registered handlers.

IV. Discussion and Evaluation

The aim of this section is to outline how the chosen design of RSB meets the requirements posed by the application scenarios introduced in Sec. I.

Starting with practical issues, RSB's small dependency graph facilitated the application on our NAO robots. For this platform, no complete system administration access for the embedded Linux PC is provided by the manufacturer. Nevertheless, several standard libraries like Boost [19] are available on the system. RSB (for the C++ implementation) explicitly states to rely only Boost for its core. Moreover, the Spread-based connector requires the Spread Toolkit, which is an easy to compile C-library, and Google Protocol Buffers for internal serialization. This small footprint allowed the application of RSB on NAO without a much more complicated cross-compilation setup.

A. *Assessment of Openness*

As stated in Sec. I-A, an important aspect of scalability is the availability of common data types. This is an essential requirement for coupling components originally developed for different systems. ROS [3] provides a collection of these data types in the common_msgs package[9]. However, we argue that the technical realization with the custom IDL results in a coupling to ROS which prevents the general use of this message definition technique outside ROS. For this reason, RSB does not include a custom IDL for data exchange and does not require its application. Instead, an additional project RST[10] provides common data definitions

[9]http://www.ros.org/wiki/common_msgs

[10]Available at https://code.cor-lab.de/projects/rst

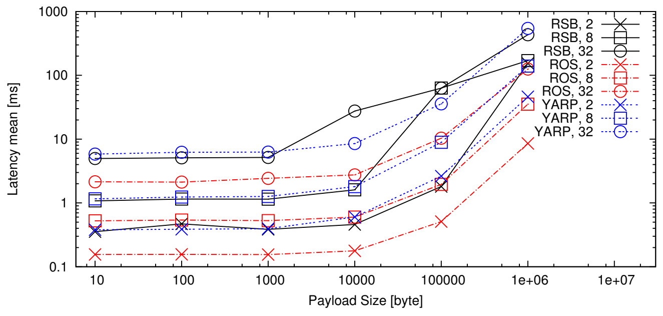

Fig. 2. Roundtrip latency for RSB, ROS and YARP scaling over data size and the number of components that generate the pong reply. Please note the logarithmic scales.

using an external IDL (Google Protocol Buffers [20]). This IDL is well-known outside the robotics community and is not inherently coupled to a broader framework, nor is the RST project itself limited to RSB. In order to access the data types already defined by ROS for robotics (cf. openness), we evaluated an automatic translation which proved to work properly. Listing 1 shows an exemplary data type definition from RST.

To validate the intended facilitation of interoperability by architectural choices, we prototypically implemented a dynamic bridge component that translates between RSB-based and ROS-based systems. Structurally, the component consists of additional connectors that implemented the communication protocols used within the ROS framework, namely TCPROS and the XMLRPC-based connection setup and converters for the ROSMSG serialization mechanism. The component operates by creating a listener or informer using these connectors and converters and participating in the ROS-based part of the system. Another informer or listener participates in the RSB-based part of the system. Using the converter mechanism, messages/events received in one part of the system are first (de-) serialized into a common representation and then sent to the other part of the system after a second (de-) serialization.

B. *Assessment of Scalability*

In order to scale a system, the efficiency of the processing must be sufficiently high. We performed a benchmark of the RSB C++ implementation against well-known competitors in robotics, ROS and YARP (cf. Sec. V), to quantitatively validate that the performance of RSB is comparable and sufficient for robotics use cases. For this purpose, RSB's Spread-based transport was compared against ros_comm[11] and YARP in a roundtrip scenario. A sending component sent a ping message to a varying number of replying components, which replied with a pong. The latency between sending the ping message and receiving the pong message was measured for different message sizes. All receivers and the sending component were started in unique processes on the same node and connection was established through the loopback device. This computer was equipped with an Intel® Core^{TM}2 CPU with 2.40 GHz and 4 GB of RAM. The results of this

[11]http://www.ros.org/wiki/ros_comm

benchmark are depicted in Fig. 2. As visible, ROS provides the fastest roundtrip performance while RSB and YARP operate with a comparable performance. We conclude from this that RSB's current performance is sufficient for robotics use cases as YARP is successfully applied in several scenarios. However, we will further try to improve the performance of RSB and evaluate the progress by continuously running the benchmarking suite[12]. Please note that the advantages of the multicast mechanisms implemented by Spread are not exploited by the current benchmarking setup. Several connected workstations would be required to make them visible.

Having a look at optimization possibilities in RSB, the content-based filtering approach (cf. Sec. III-C) can be tuned by implementing parts of the matching inside connectors. One example of content-based filtering consists of applying XPath [21] expressions to serialized protocol buffer messages, e.g. from RST. In order to achieve sufficient efficiency, the filtering is performed in the transport layer prior to deserialization and dispatching with the optimization strategy introduced in Sec. III-C. To this end, an approximate DOM-interface has been specified for protocol buffers messages to enable application of an XPath engine. Message fields required by the XPath engine are lazily deserialized to minimize overhead. Consider the following example based on the message definition in Listing 1: `node()/detections[@confidence>0.5]`. This XPath expression would result in handlers being only called for face detection messages which contain at least one reliable detection. Applying XPath facilitates scalability in two aspects. On the one hand, it provides clients with flexible access to data which is an essential requirement in constantly changing and growing systems. On the other hand, it allows the efficient application of binary encodings while exposing a high-level interface to clients. For scaling a system, XPath expression can be made a configuration property of the system instead of requiring code changes and recompilation.

On the level of integrating diverse components with defined runtime environments, we could assess RSB's scalability while implementing several GStreamer[13] plugins. These plugins provide sinks and sources for RSB-based communication with GStreamer pipelines and thus are maintained through GStreamer's life cycle management. Hence, the explicit lack of a component life cycle model in the RSB core is an essential requirement for this kind of integration. This example also demonstrates how RSB can be used within different organizational structures which might impose different life cycles components. RSB does not enforce one of them and hence can be integrated in any of the systems, an advantage of reduced lock-in.

C. Assessment of Introspection Support

RSB's introspection support has been used in a human-robot interaction study centered around the NAO robot. Control of the robot and streaming of its sensor information was implemented using a set of components running partially on the robot and partially on a workstation. All RSB-based communication, including the sensor data streams generated on the robot, was recorded and stored using introspection-based tools and with a common format. Based on this format, a generic replay component was implemented and addresses the requirement to use the same components for online and replay purposes (the same RSB interfaces are served). For this purpose, the management of timestamps outside of the user data in framework-supplied data structures provides the replay tool with the ability to analyze and also automatically adapt these timestamps without needing knowledge about the structure of the user data. Moreover, due to the rich meta-data attached to every RSB event, no manual augmentation with timestamps or other meta-data was required for the recording. Finally, assuming synchronized clocks of the participating computers, a synchronization of recorded data is possible solely and on these meta data in a generic way.

V. Comparison to other Middlewares

Subsequently, we will briefly review related approaches for system integration with a focus on robotics middleware. As a complete overview is beyond the scope of this paper, we refer the interested reader to recent surveys, e.g., about robotics middleware [22] or complete development environments for mobile robots [23]. For our comparison we selected NaoQi [10], ROS and YARP as representatives of the state-of-the-art in robotics system integration, demonstrated by their wide adoption in mobile and humanoid robotics. Please note that the following assessments focuses on the major differences and may be subject to a strong bias on the requirements introduced in Sec. I.

A. NAOqi

The humanoid robot NAO is delivered with a the SDK NAOqi, which consists of a middleware core and functional components specifically for the robot. The middleware in NAOqi is based on a broker architecture and implemented using SOAP over TCP. In contrast to RSB, NAOqi comes with a component model which consists of modules. A client component needs to implement the module interface. NAOqi uses request-reply-based communication as fundamental communication pattern. A module can implement $1 : n$ publish-subscribe only manually by managing a list of callbacks for interested modules. The RSB-like $m : n$ anonymous broadcast semantics are realized through a central storage component called *ALMemory*, which raises signals to subscribed listeners.

NAOqi's data type model is based on a generic data container called *ALValue*, comparable to JSON values [24]. All transfered data must be expressed with this container. No early transition to client domain data is available.

Even though NAOqi uses SOAP as its underlying communication mechanism, no standardization efforts are visible and the implementation is inherently coupled to NAO. Middleware and functional modules are delivered as a single

[12]The whole benchmarking code is publicly available at `https://code.cor-lab.de/projects/rsbench`.
[13]`http://gstreamer.freedesktop.org`

package with specialized interfaces for functions of the NAO robot. Also, no introspection tools are available and extension points to change communication behaviors do not exist.

B. ROS Communication System

Willow Garage, the makers of ROS (Robot Operating System), aim to provide an open-source, meta-operating system supporting robotics software development in different robotics domains with a focus on mobile manipulation. Over the last years, ROS gained wide community support. It provides access to a large number of software libraries for building robotics systems, which expose their external interface using features of the ROS communication stack called ros_comm. Due to the wider focus of ros, our approach mainly compares to this communication subsystem.

The ros_comm stack implements a type-based, anonymous publish/subscribe model where each logical connection between a set of publishers and subscribers is bound to a symbolically identified *Topic* which prescribes the type of exchanged data with this channel. Conceptual differences to the concepts presented in Sec. III are the lack of hierarchically organized topics[14] and the restriction to a single data type per topic whereas RSB allows polymorphic channels. Polymorphic channels remove the necessity to couple namespaces with data type design. More importantly, this facilitates the development of generic software components such as the RSB/ROS bridge described in Sec. IV. While ROS also provides content-based filtering with the message_filters package[15], installed filters are not visible to the transport layer and hence cannot be optimized like for RSB.

Besides publish/subscribe, ROS supports *Services* which offer remote procedure calls. In contrast to RSB, ROS services are conceptually outside of the channel system described above. In RSB an RPC-like request/reply pattern is logically implemented on top of the unified event bus. This facilitates easier implementation of advanced event-based RPC patterns with optional feedback such as the *Task-State* pattern [25] and allows us to apply the same toolchain, e.g., for introspection or recording. Performance drawbacks caused by this implementation can be mitigated if necessary by using special connectors which implement RPC interaction natively at the network-level. However, in contrast to ROS and other approaches, this optimization is not visible to client code or even framework tools.

The ROS communication stack may very well scale to larger distributed robotics systems. However, it remains unclear how it is able to support more tightly integrate component interactions. To this end, no direct equivalent to the presented in-process transport for collocated optimization [26] is available. The comparable features we could identify, in particular *Nodelets* and *intra-process* publishing seem to require changes at the level of the client code which would prevent seamless reconfiguration of ROS components to scale down to embedded platform and integration of components in more tightly coupled feedback controllers. The RSB architecture allows this kind of configuration change without code-level changes.

While the aforementioned differences influence (among other aspects) the scalability of a system integration solution, one of the primary goals of the ROS communication library as well as RSB is to be as lightweight as possible. Here, both frameworks are on an equal footing with the Boost libraries as their primary dependency and additions for communication with the ROS Master (XML-RPC) in ROS or Protocol Buffers as the internal serialization mechanism in RSB. However, ROS lacks clearly defined extension points for important functionality, e.g., for 3rd party serialization formats or different transports.

C. YARP

YARP is another robotics middleware which has been used for the system integration in a number of projects on advanced robots [4] such as Kismet, COG or currently in the context of the iCub humanoid robot[16]. With our approach it shares the motivation to focus only on the middleware aspects of robotics system integration. Besides its dependency on ACE, YARP is a lightweight framework which allows the execution on platforms with limited processing capabilities. Openness is additionally supported by a number of available *Carriers* which realize different transports such as TCP, UDP or also local transport.

Using YARP, an integrated system is organized as a number of independent components which expose their interface via *Ports* that allow to send data to other participants. Port communication implements a distributed observer pattern, which is similar to the event bus concept introduced in Sec. III. However, compared to the presented implicit invocation architecture, ports are directly connected to each other through an explicit connection which has to be established manually. While a hierarchical name scheme typically appears in port identifiers, e.g., in the iCub interface, their syntax is rather unspecified and the hierarchical structures do not impose any semantics.

The default serialization in YARP is the *Bottle* format. At client-level, the bottle concept provides a simple interface where native data types, strings, lists and dictionaries can be added and removed to a bottle instance. While this represents a straightforward concept, the lock-in at code-level would be very high if used extensively. Furthermore, no IDL for specifying data types and no generators for encoding or decoding code exist. That said, YARP supports a limited number of predefined data types natively, e.g., OpenCV images, and users may write their own serialization engines. Besides that, content-based filtering is not directly supported neither at client-level nor within the framework.

[14]ROS allows a hierarchical syntax in topic identifiers but does not enforce or associate any semantics with it. Redirections into a common topic or sub-topic would require developers to know beforehand every possible topic to be aggregated.

[15]`http://www.ros.org/wiki/message_filters`

[16]http://www.icub.org

VI. Conclusion and Outlook

This contribution presented the Robotic Service Bus (RSB) middleware for system integration in robotics and intelligent systems. The described structure features (i) a *logically unified and hierarchical bus* based on exchangeable transports, (ii) allows the *integration of domain-specific data with filtering at transport level* and (iii) facilitates *introspection and analysis* of an RSB system utilizing a well defined set of event meta data.

By strictly making openness and scalability a key requirement, RSB is usable on diverse platforms and within varying research contexts. This explicitly includes the ability to integrate with existing components for different middlewares in order to strengthen software reuse and foster collaboration in research projects. This integration is further facilitated by the rich set of extension points within the described architecture and has already been demonstrated by the dynamic bridge discussed before and with a TCP-based ROS connector (so far implemented in Common Lisp) that can be transparently replaced by our native connector implementations.

A second key property of RSB is the reduction of framework lock-in. This has been achieved by an early inclusion of user data types, the (intended) lack of explicit component models and is also supported by the lightweight dependency graph with standard libraries.

Besides scalability and reduced framwork lock-in, RSB features a number of functional concepts not found in other robotics middleware so far. In particular, the consequent adoption of an event-based model, the hierarchcial message distribution, content-based filtering at user and transport level as well as the flexible mapping from serialization to domain-specific types are beneficial features for system integration.

While the presented framework is already usable for a broad range of tasks, several additions are planned to address further use cases or make them more convenient. Currently, solutions to the question how to correctly form the logically unified bus across several transports are under design and development. Several transports are often involved when performance is critical (e.g. by choosing in-process communication for parts of the system) and hence the obvious solution of sending all events to all transports is not feasible. The current RSB model already provides a solution to this problem by installing custom sending strategies for events, which e.g. could send only every n-th event on expensive transports. However, we will further develop more formal ways of bridging between several transport mechanisms.

In the context of performance optimization we will also integrate signals which enable informing components to be aware of listeners in order to prevent expensive computations when no listener is available. More generally, we will integrate a distributed name service which allows access to participants of each scope and provides contextual information at runtime about an RSB system.

With its focus on openness, we acknowledge the existence of widely used approaches such as ROS and thereby facilitate integraton with these in collaborative projects. That said, we are convinced that innovative features explored in smaller communities can provide hints also to widely accepted approaches such as ROS and that the presented concepts are applicable also in system integration tasks beyond robotics.

References

[1] S. Vinoski, "A Time for Reflection," *IEEE Internet Computing*, vol. 9, pp. 86–89, Jan. 2005.

[2] A. S. Tanenbaum and M. v. Steen, *Verteilte Systeme*. Pearson Studium, 2003. German.

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

[4] G. Metta and P. Fitzpatrick, "YARP: yet another robot platform," *Journal on Advanced Robotics*, vol. 3, no. 1, pp. 43–48, 2006.

[5] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Proc. of the 1st Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR '08, (Berlin, Heidelberg), pp. 87–98, Springer-Verlag, 2008.

[6] P. Soetens, "A software framework for real-time and distributed robot and machine control," 5 2006. `http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf`.

[7] S. Wrede, J. Fritsch, C. Bauckhage, and G. Sagerer, "An XML based framework for cognitive vision architectures," in *Proc. 17th Int. Conf. on Pattern Recognition, 2004. ICPR 2004.*, pp. 757–760, IEEE, 2004.

[8] S. Wrede, *An Information-Driven Architecture for Cognitive Systems Research*. Phd thesis, Bielefeld University, 2008.

[9] S. Rutishauser, "Cheetah: compliant quadruped robot," tech. rep., Biologically Inspired Robotic Group, 2008.

[10] Aldebaran Robotics, "NAO SDK V1.10 User Guide," 2011.

[11] M. Rolf and J. J. Steil, "Continuum Kinematics Simulation of the Bionic Handling Assistant," in *IEEE Int. Conf. on Robotics and Automation*, (St. Paul, Minnesota, USA), IEEE, 2012. submitted.

[12] Vicon Motion Systems, "Vicon Nexus 1.7." `http://www.vicon.com/products/documents/nexus_17_techsheet.pdf`, 2011.

[13] T. Faison, *Event-Based Programming*. Apress, may 2006.

[14] D. C. Luckham, *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[15] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[16] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 2396: Uniform resource identifiers (URI): Generic syntax." `http://www.ietf.org/rfc/rfc2396.txt`, 08 1998.

[17] Y. Amir and J. Stanton, "The spread wide area group communication system," Tech. Rep. CNDS-98-4, Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 2007.

[19] B. Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*. Pearson Education, 2005.

[20] K. Varda, "Protocol buffers: Google's data interchange format." `http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html`, 6 2008.

[21] "XML path language (XPath) version 1.0." `http://www.w3.org/TR/xpath/`, 1999.

[22] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *Robotics Automation and Mechatronics 2008 IEEE Conference on*, no. Ram, pp. 736–742, Ieee, 2008.

[23] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, pp. 101–132, Dec. 2006.

[24] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*. JSON.org, 07 2006.

[25] I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, and S. Wachsmuth, "Coordination and Control for Complex Robot Software Systems: The Task-State Pattern," *Journal of Software Engineering in Robotics*, 2011. submitted.

[26] M. Valente and R. Palhares, "Collocation optimizations in an aspect-oriented middleware system," *Journal of Systems and Software*, vol. 80, pp. 1659–1666, 10 2007.