Distributed Robotic Computing

Lorenzo Natale

This is a post-peer-review, pre-copyedit version of an article published in Encyclopedia of Robotics, Springer. The final authenticated version is available online at: https://link.springer.com/referenceworkentry/10.1007/978-3-642-41610-1_4-1

Cite this Chapter as:
Natale, L. (2020). Distributed Robotic Computing. In: Ang, M., Khatib, O., Siciliano, B. (eds) Encyclopedia of Robotics. Springer, Berlin, Heidelberg.
https://doi.org/10.1007/978-3-642-41610-1_4-1

Synonyms

Robotic software middleware.

Definitions

Computing hardware in modern robotic platforms ranges from embedded microcontrollers to general-purpose CPU. To implement complex software applications, it is therefore common practice to separate computation in different components, whose execution is concurrent and distributed on the available hardware. Components in a distributed application exchange information – typically through a wired or wireless network – and are coordinated to obtain the desired behavior of the robot. Robotic

Lorenzo Natale
Istituto Italiano di Tecnologia, via Morego 30, 16163, Genova, Italy e-mail: lorenzo.natale@iit.it

researchers adopt software middleware to simplify the task of writing distributed applications and increase code reuse. The goals of this Chapter are to illustrate the general concepts of a software middleware, revise the specific middleware adopted in robotics and discuss open problems that remain to be addressed.

Overview

A robot application is often too complex to be executed on a single processing unit. This complexity may be due to the large amount of data that the robot needs to process in real-time, or, simply because the robot is interfaced with external devices or other robotic systems. Computation is therefore separated in components, which run concurrently on different CPUs on a set of interconnected hosts. This is a distributed system (Fig. 1).

Notice that in this Chapter the term *component* is used to identify any software unit which can be executed independently on a distributed system. This definition differs and is much broader than the one usually adoped in the field of Component Driven Software Development (Szyperski 2002). Finally, because components are often implemented as separate processes, these terms are used interchangeably.

In distributed systems software components are typically implemented as idependent processes, and a software middleware is adopted to simplify development. A middleware is a software framework which provides common functionalities for inter-process communication and primitives for synchronization among processes. Conventionally, computer programs are organized in functions or classes (depending on the programming paradigm). The latter are compiled and linked together and made accessible to one or multiple threads running in the same executable. Threads running on the same executable usually share memory, communication is therefore trivial. In distributed systems, on the other hand, computation is split among several processes which are executed on different hosts. Messages are exchanged to distribute data and aggregate result of the computation performed by the individual hosts that take part in the computation. A software middleware offers functionalities to support writing software following this paradigm.

The next section revises these functionalities and it introduces the problems solved by a middleware. For reasons of space this Chapter focuses only on the functionalities that are offered by the software middleware commonly used in robotics (the reader is invited to read (Tanenbaum and Maarten 2006) for a more complete treatment on the subject).

General concepts of software middleware

One of the goals of a software middleware is to provide the programmer with an easy to use API supporting communication between components. Operating systems

already provide functionalities for inter-process communication on the same host (IPC) or, using network protocols, across different hosts (inter-host). The former can be done using shared memory or message queues. The latter is typically done using sockets and TCP/IP protocols. These functionalities are relatively low-level, and are mostly concerned with transferring bytes. Relying on these functionalities to transfer messages is possible but leads to code that is complex and difficult to re-use across different operating systems, network protocols and medium.

The communication layer in a middleware is designed to hide details of specific communication protocols and to offer an abstraction which deals directly with complex structures including floating point variables and dynamically linked lists. Code on top of the communication layer is portable across different systems, communication medium and protocol. The majority of middlewares are also multi-platform, this allows porting code across different operating systems.

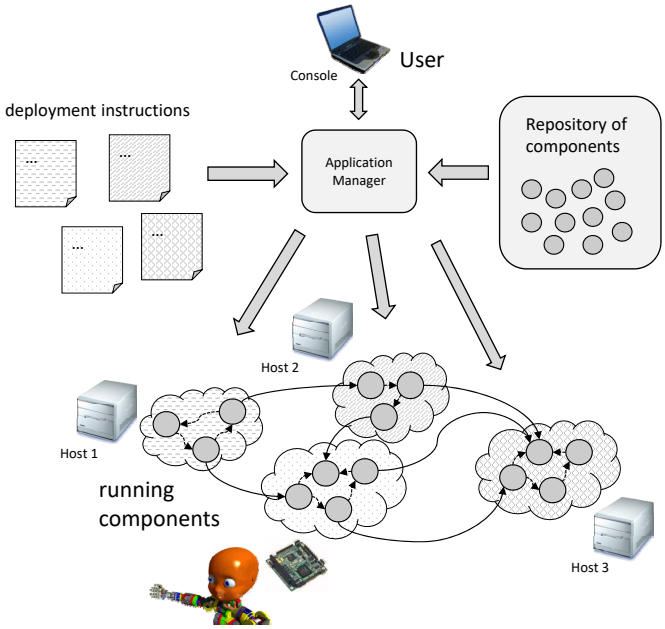

Fig. 1 A robot distributed system. Bottom: components (represented by circles) are executed on a set of computers (hosts). Components on the same host communicate using inter-process communication (IPC, dashed arrows), inter-host communication is obtained using network protocols (solid arrows). Top: deployment and monitoring. Some middlewares provide tools that allow the user to install, configure, execute and monitor components from a console.

Interface Description Language

Data travels on the network in the form of bits and bytes. Because complex data structures are rarely contiguous in a computer memory, one of the tasks of the middleware is to collect all the fields that make up a data structure (following pointers in case of dynamically allocated data structures) and transmit it on the network. This process is called serialization. The opposite process, de-serialization, collects bits and bytes received from the network and fill memory appropriately (Fig. 2). Because memory copy may be expensive for large data types, these operations need to be carefully optimized.

To proper serialize and de-serialize data structures the middleware needs to know how data is represented. It is the task of the programmer to proide this description using a special language made available by the middleware, called Interface Description Language (IDL). Data descriptions in IDL format are automatically parsed by an intermediate compiler which is responsible to produce language specific code that implements the data type, including serialization/deserialization routines (Fig. 2).

Some middlewares support remote object invocation, which is sometimes also referred to as Remote Procedure Call, or RPC (OMG 1997a; ZeroC Inc 2017). Remote object invocation allows a process to request execution of a function that is implemented in a given, remote object. In this paradigm the middleware keeps track of the remote objects and the functions they offer. The user code invokes functions as if they were linked locally in the process. The middleware catches such invocations, and is responsible for looking up (and eventually execute) the process that hosts the required object. It then dispatches the requests made by the user program to the remote hosts, collects the result of the computation and delivers it back to the calling program (this process is called parameters marshalling/de-marshalling). The code required for remote execution of objects, parameter marshalling and demarshalling is usually generated automatically from a compact description of the objects using specific keywords in the IDL.

The IDL of a middleware allows programmers to describe messages and component interfaces in a intermediate language. From this description, the IDL compiler can generate code for any target language. This greatly simplifies interoperability between components written in different languages (Fig. 2).

Communication paradigms

In a system that supports Remote Procedure Calls (RPC) a component can request the execution of a procedure (or subroutine) in a remote machine. An Object Oriented Middleware extends RPC systems, by supporting instantiation of remote objects and invocation of their methods. Communication in Object Oriented Middleware and RPC systems is typically synchronous (although asynchronous variants

exist), in which a sender waits for an explicit acknowlege from the recipient of a message.

In a Message Oriented Middleware, on the other hand, the infrastructure provides primitives for sending and receiving messages, and components communicate in an asynchronous way, usually following a publish-subscribe paradigm. In this paradigm processes are not aware of other parties involved in the communication, but communicate through named entities called *topics*. A process that produces data advertises it by registering itself as a producer for a given topic in a shared database. A consumer that wishes to read this information registers itself as a subscriber for that topic . Therefore, in a publish-subscribe architecture a process that needs to read data from a source needs to know only its name in the shared database. The middleware takes care of establishing connections between publishers and subscribers when needed.

Publish-subscribe is widely adopted for the following reasons. Firstly, it is intrinsically many-to-many in that there are no limitations on the number of publishers and subscribers that are concurrently producing or reading data on the same topic. Secondly, it offers various degrees of decoupling between components (Eugster et al 2003). Publishers do not need to know the location (i.e. the host) in which they are executing (*space decoupling*). Communication is asynchronous. Publishers are allowed to continue execution immediately after they have published data. Symmetrically, subscribers do not need to wait for receivers but can check for new messages whenever they want. This level of decoupling allows running processes asynchronously and avoids dangerous timing dependencies when multiple receivers,

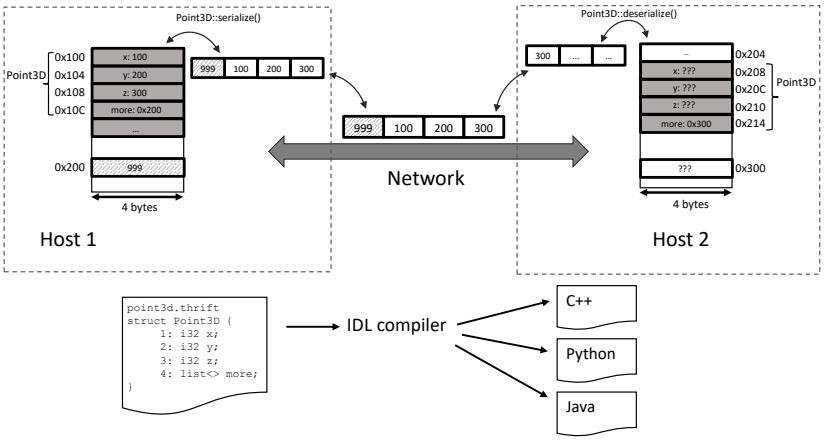

Fig. 2 Top: data travelling on the network needs to be serialized at the sender side and deserialized at the receiver side. This is especially important for complex structures that are not stored in a continuos chunk of memory (like the structure depicted in the figure). Bottom: the programmer describes data types using an IDL (e.g. the Apache Thrift). The IDL compiler automatically generates language specific code that defines the type, including serialization and deserialization routines. Notice that the same IDL descriptor can be used to generate code for different languages.

with different computational loads, exists (*synchronization decoupling*). This feature is particularly relevant for robotics, because introducing time dependencies between software components running on different machines may have negative effects. Finally, publish-subscribe does not require publishers and subscribers to be running to receive data. Published data can be stored by the middleware even when the publisher is no longer running and is delivered when a publisher subscribes (*time decoupling*).

Data persistency is an important property of communication, it allows fine grained control of how data is stored and in which order it is delivered. A middleware may provide different strategies for storing messages: First-In-First-Out (FIFO) stores all messages to guarantee delivery, while Oldest-Packet-Drop (ODP) attempts to minimize communication latency by dropping old messages as soon as new ones are available.

Finally, it is worth mentioning the Data Centric middleware, which is adopted in growing number of distributed applications that adopt the DDS standard (OMG 1997b). This approach is also asynchronous, and can be implemented on top of a publish-subscribe system.

Managing complexity: starting up and monitoring large applications

The procedure required to execute a distributed application is called *deployment* (Fig. 1). In general this requires the installation and update of the software on the hosts, the execution of all the components and, eventually, their deactivation or shutdown. Deployment of a large distributed application becomes easily cumbersome, and for this reason software middlewares provide dedicated tools that can perform at least some of these steps automatically.

Deployment tools require that the user provides information describing the available hardware (number of nodes, their hardware capabilities and the topology of the network), the required components and instructions on how they should be executed (their parameters and the identifiers of the hosts on which they should run). These tools may ideally provide features for monitoring the CPU usage of the hosts on the network, and even perform automatic allocation of components on hosts, depending on their current CPU usage or hardware requirements.

Robotic Middleware

This section discusses some additional, key features that characterize software middlewares that have been developed by the robotic community. In addition it revises some of the frameworks that have been adopted in the robotics community.

Real-time performance

Robotic software must deal with timing constraints, especially when implementing control loops. To ensure proper operation and safety checks (like maximum torque or joint limits) the low-level routines for motor control are implemented using real-time systems (either embedded or not). Hard real-time systems are designed in such a way to guarantee time constrains. To provide such guarantees a software middelware must rely on an operating system that provides hard real-time capabilities (examples of real-time operating systems are QNX, RTAI Linux and Linux Xenomai). The majority of distributed systems are soft real-time or best-effort, in that the system has no way to guarantee a requested level of performance, but it does its best given the available resources.

Timing aspects that are important for robotics distributed applications are communication latency and jitter. Latency measures the delay introduced by the communication layer, while jitter is its variance. The literature focuses on the effects of communication latency (Zhao et al 2015), however jitter may even be more dangerous, because it prevents latency compensation and prediction in control loops.

A simple way to reduce latency in control loops is to shorten message queues. For this reason several middleware (Natale et al 2016; Huang et al 2010; Dantam et al 2015) provide buffering policies that drop old messages to minimize communication latency (Oldest-Packet-Drop).

Another way to reduce latency is to minimize communication overhead. This can be achieved by designing efficient encoding mechanisms that minimize the bytes sent on the network. To save CPU cycles a middleware can implement efficient inter-process communication using shared memory. In this case messages are not copied to the memory areas of the individual processes, but are stored in a shared memory space from which they can be directly read or written. This approach is also called zero-copy and is implemented in the middelware Ach (Dantam et al 2015) and aRDx (Hammer and Bäuml 2013). A similar mechanism allows the middleware to copy a message only once (copy-once) when it is received from the network, avoiding extra copies (Hammer and Bäuml 2013). These approaches become important when transmitting large packets, like, for example, images (Fig. 3).

A source of variable latency in distributed systems come from conflicts arising when processes compete to get resources. This can be at the level of CPU and on the network. Inside a computer different processes and threads compete to get CPU time. How CPU is assigned may affect communication performance. A way to reduce communication jitter is to assign higher priority to those threads that handle the communication. This is clearly a viable solution and it prevents that low-priority tasks have a negative impact on the threads that manage the communication of high priority tasks.

In a distributed system there are however other sources of conflicts that affect communication performance (Fig. 4). Data packets compete to travel on the network. Outgoing packets are queued before they reach the network card and are dispatched on the network. Similarly, incoming packets are queued when they are received from the network, before they are dispatched to the applications. In a

switched network (a typical network topology in a robotic system) data packets from one computer compete with data coming from other sources when they traverse the ports of the switch. These conflicts cannot be avoided, but there are ways to assign priorities to packets so that some connections have precedence over others. This technology is called Quality of Service (QoS) Control and it is used, for example, to route voice traffic on the internet and avoid interruptions (Almquist 1992).

Robot abstraction layer

Robot components are made from different vendors and there are no accepted common standards. Code written for one robot cannot run on another. This problem sometimes affects even consequent versions of the same robot. One of the functionalities offered by robotics middleware is a robot abstraction layer for hardware

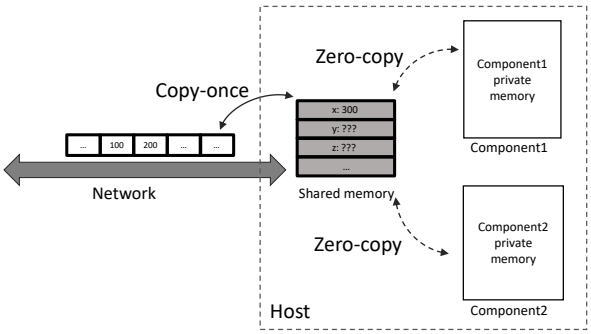

Fig. 3 To reduce overhead some middlewares rely on shared memory to avoid copying messages for inter-process communication (zero-copy). The middleware can also use shared memory to store data from the network, making it available, without further copies, to all the components that have requested it on the same host (copy-once). In contrast, in conventional approaches, a copy of the message would be copied to the private memory of all the components.

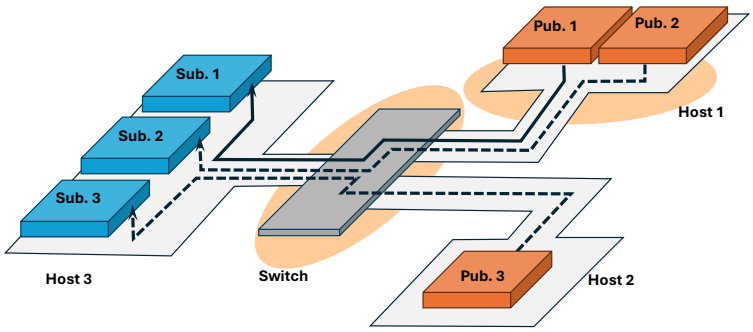

Fig. 4 Sources of conflicts in the communication of a distributed system (Paikan et al 2015).

devices and simulators. Similarly to the communication abstraction layer, the robot abstraction layer provides an interface that is independent of the robot and it allows recycling code across different robotic systems and simulators.

Examples of such an abstraction layer are ROS Control (Tsouroukdissian 2014) and the device interfaces in YARP (Natale et al 2016). These abstraction layers define a set of hardware interfaces for reading the status of arrays of joints and control them in position, velocity or torque mode. They also define interfaces for devices like force and torque sensors or inertial units. The YARP middleware implements objects that automatically provide remote invocation of such functions in a distributed state of computers. H-ROS (Acutronic Robotics 2017) is a recent effort that aims to develop a standard framework for developing interoperable robot components. In addition to sensing and actuation it defines three other classes of devices, namely communication, cognition and hybrid.

Comparison of software middleware for robotics

General purpose middlewares like CORBA (OMG 1997a), ICE (ZeroC Inc 2017) or ZeroMQ (iMatix Corporation 2014) are mature frameworks for developing distributed systems. They have been rarely used in robotics – with some exceptions, e.g. ICE has been used on the ARMAR robots (Vahrenkamp et al 2015). This is due to the fact that they have a steep learning curve and provide functionalities that were not considered fundamental for robotics. For many years, the robotics community preferred to develop custom solutions, sometimes adopting general purpose middleware as transport layers, and exposing only a subset of their functionalities to the user. This trend started with frameworks like OROCOS (Bruyninckx et al 2003), Smartsoft (Schlegel and Worz 1999; Stampfer et al 2016), GenoM (Mallet et al 2010), Player (Collett et al 2005), and YARP (Metta et al 2006; Fitzpatrick et al 2014; Natale et al 2016), until ROS (Quigley et al 2009) established itself as the de facto standard.

The main features of OROCOS are that it provides real-time support and a component model. Communication in OROCOS includes RPC and asynchronous publish-subscribe paradigms, implemented with transport layers for real-time interprocess communication and inter-host communication (based on CORBA, ROS and YARP). LCM (Huang et al 2010) is a framework based on UDP multicast. In TCP/IP networks multicast allows efficient one-to-many communication without retransmission. LCM leverages on UDP multicast to provide an efficient publish-subscribe mechanism and tools for logging and message inspection without overhead. OpenRDK (Calisi et al 2012) proposes a communication framework based on a shared memory concept (called white-board) in which processes can read and write data. The framework ensures that data written on the white-board is distributed and maintained consistent among all hosts in the network. Ardx (Hammer and Bäuml 2013) provides a communication framework which aims at maximizing performance by reducing unnecessary memory copies and by providing efficient protocols for inter-

process communication (IPC). In addition, Ardx provides support for real-time systems and it allows tuning priority of channels by controlling QoS parameters. Ach (Grey et al 2013; Dantam et al 2015) also provide real-time support and efficient IPC communication with minimal latency.

ROS is a middleware which was developed in 2005 for a robotic project at Stanford University. It was initially supported by Willow Garage and later by the OSRF. ROS offers communication through *topics* in a publish-subscribe system and, through *services*, with RPC. A central server takes care of registering topics and establishes connections. ROS has an IDL that allows describing data types and RPC services. It is strongly typed in that it ensures that publishes and subscribers exchange exactly the same data type. Data is exchanged directly by components without intervention from the central server. ROS implements a simplified form of publish-subscribe, in which senders and receives need to be running to receive data (although a topic can be configured to store the latest published data for later delivery). In addition a parameter server allows components to exchange data in the form of parameters. This centralized form of communication is inefficient but it provides an easy way to feed parameters to ROS components.

YARP is a communication middleware in which components exchange data via objects called *Ports*. YARP promotes asynchronous communication and it is close to a publish-subscribe system like ROS. However, in this case connections are established dynamically and can be re-routed at runtime without restarting components. Ports are named entities that are registered on a central name server. YARP is not strongly typed. Messages describe themselves using tags and can be interpreted and translated at run-time. Custom data types can be defined with an IDL (the Apache Thrift IDL). It supports stream-based communication and RPC for services. Services can have complex interfaces and can be specified with the IDL. YARP includes built-in protocols (i.e. TCP, UDP, multicast, XML/RPC, MJPEG and HTTP with JSON), and it includes a plug-in system for dynamically loading custom protocols at runtime (Fitzpatrick et al 2014; Paikan et al 2014). For interoperability with the ROS system, YARP provides ROS-compatible protocols and it is able to interpret messages written with the ROS IDL. It also allows tuning priorities for individual data channels, to reduce jitter and latency of time-critical loops (Paikan et al 2015). Finally, YARP includes a robot abstraction layer which defines interfaces for reading sensors and for motor control in joint and operational space.

The Data-Distribution Service (DDS (OMG 1997b)) is a standard specification by the OMG for a publish-subscribe system. DDS adopts a data centric approach, in which components describe the data they share and it is responsibility of the middleware to maintain a globally consistent memory space that is shared by all the components in the network. DDS provides an IDL for defining data types and includes a standard API and a standard protocol (RTPS). There are in fact many, compatible, implementations of DDS. DDS is fully distributed and there is no need for any central server for naming and establishing connections. DDS allows fine control on how data is transmitted, stored and delivered. This is achieved by assigned QoS parameters (or policies) to data writers and readers. DDS requires that senders and receivers agree on the QoS of the data they exchange. This prevents an application

that need data at fast frame rate to be connected to a slow producer. DDS has been designed for aerospace applications. It is getting popularity in robotics following its adoption for the next release of ROS, ROS 2.0 (Fernandez et al 2014).

To conclude this Section, Table 1 compares various robotics middlewares, summarizing their main features.

Table 1 Comparison of middleware for robotics.

	Comm. paradigm	Protocols	IDL	QoS	Operating systems	Notes
Smartsoft	pub/sub, RPC	CORBA	custom		Windows, Linux, RTAI, QNX	IDE for component development and deployment
OROCOS	pub/sub, RPC	CORBA, YARP, ROS	CORBA IDL		Windows, Linux, OSX, Xenomai, RTAI	
OpenRDK	white-board	TCP, UDP, IPC			Windows, Linux, OSX	
LCM	pub/sub	Multicast			Windows, Linux, OSX	tools for logging
Ach	pub/sub	TCP, UDP, IPC				low-latency IPC, logging
aRDx	pub/sub	TCP, UDP, IPC		channel	Linux, QNX	low-latency IPC
ROS	pub/sub, RPC	TCP, UDP	rosmsg		Linux	tools for remote execution and logging
YARP	pub/sub, RPC	TCP, UDP, Multicast, user-defined	Thrift, rosmsg	channel	Windows, Linux, OSX	GUI for remote execution and monitoring, logging
DDS	pub/sub	RTPS	OMG IDL, XML	data		QoS policies allow specifying data persistence, reliability and priority

Open problems and conclusions

The development of software middleware for robotic applications has been driven by the need of frameworks that are simple to use and have only the minimal set of functionalities required by the robotic community. This is clearly demonstrated by the success of ROS and its establishment as a de facto standard in robotics. The steady maturation of robotics and consequent push for commercialization is going to challenge this view, with an inevitable shift towards systems that are more complex.

None of the middlewares developed specifically for robotics has security features. While some attempts have been done to extend ROS with security features (White et al 2016), secure technologies have not been adopted in robotics so far. Future applications in the domains of healthcare, safety and domestic service will put stringent requirements to ensure protection against denial-of-service and data spoofing. This may come with a cost because adopting data encryption technology may increase communication latency and CPU usage. This may be affordable in modern, high-end CPUs, but could be a serious issue in embedded systems (Morante et al 2015).

Commercial applications will also require fault-tolerant systems, especially across unreliable, wireless communication medium. Robotic middlewares have not been designed to handle these cases. Removing single point of failures and adding reliable and efficient one-to-many protocols will prove to be important features to support future applications. Fine control of data persistency was not considered fundamental in robotics and it is partially implemented in robotics middleware. It is easy to imagine, however, how this feature becomes critical for fault-tolerant systems.

Despite the wide adoption of ROS as a common middleware, there are still no accepted standards for describing software components, hardware devices, and maybe more importantly, the data they exchange. This is preventing the birth of a mature *app* ecosystem in which applications can be made by composing components developed by independent developers. The adoption of DDS as transport layer for ROS 2.0 has the potential to change the situation significantly, hopefully enabling the development of interoperable robotic components.

Cross-References

- Real-Time Computing in Robotics
- Service-Oriented Computing in Robotics
- Robot Software Programming
- Embedded Computing in Robotics

References

Acutronic Robotics (2017) Hardware robot operating system. https://www.h-ros.com/, accessed: 2017-01-26

Almquist P (1992) Type of service in the internet protocol suite

Bruyninckx H, Soetens P, Koninckx B (2003) The real-time motion control core of the Orocos project. In: IEEE International Conference on Robotics and Automation, pp 2766–2771

Calisi D, Censi A, Iocchi L, Nardi D (2012) Design choices for modular and flexible robotic software development: the OpenRDK viewpoint. Journal of Software Engineering for Robotics 3(1):13–27

Collett THJ, MacDonald BA, Gerkey BP (2005) Player 2.0: Toward a practical robot programming framework. In: Proceedings of the Australasian conference on robotics and automation (ACRA 2005), p 145

Dantam NT, Lofaro DM, Hereid A, Oh P, Ames A, Stilman M (2015) The Ach library. Robotics and Automation Magazine 22(1):76–85

Eugster PT, Felber PA, Guerraoui R, Kermarrec AM (2003) The Many Faces of Publish/Subscribe. ACM Comput Surv 35(2):114–131, DOI 10.1145/857076.857078

Fernandez E, Foote T, Woodall W, Thomas D (2014) Next-generation ROS:building on DDS. https://roscon.ros.org/2014/wp-content/uploads/2014/07/ROSCON-2014-Next-Generation-of-ROS-on-top-of-DDS.pdf, accessed: 2017-01-26

Fitzpatrick P, Ceseracciu E, Domenichelli DE, Paikan A, Metta G, Natale L (2014) A middle way for robotics middleware. Journal of Software Engineering for Robotics 5(2):42–49

Grey MX, Dantam N, Lofaro DM, Bobick A, Egerstedt M, Oh P, Stilman M (2013) Multi-process control software for HUBO2 plus robot. In: 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), pp 1–6, DOI 10.1109/TePRA.2013.6556374

Hammer T, Bäuml B (2013) The highly performant and realtime deterministic communication layer of the aRDx software framework. In: IEEE International Conference on Advanced Robotics, IEEE, pp 1–8

Huang AS, Olson E, Moore DC (2010) LCM: Lightweight communications and marshalling. In: IEEE/RSJ Intelligent robots and systems, pp 4057–4062

iMatix Corporation (2014) ZeroMQ Distributed Messaging. http://zeromq.org/, accessed: 2017-01-26

Mallet A, Pasteur C, Herrb M, Lemaignan S, Ingrand F (2010) GenoM3: Building middleware-independent robotic components. In: IEEE International Conference on Robotics and Automation, IEEE, pp 4627–4632

Metta G, Fitzpatrick P, Natale L (2006) YARP: Yet Another Robot Platform. International Journal of Advanced Robotics Systems 3(1):43–48, DOI 10.5772/5761

Morante S, Victores JG, Balaguer C (2015) Cryptobotics: Why Robots Need Cyber Safety. Frontiers in Robotics and AI 2, DOI 10.3389/frobt.2015.00023

Natale L, Paikan A, Randazzo M, Domenichelli DE (2016) The iCub Software Architecture: Evolution and Lessons Learned. Frontiers in Robotics and AI 3(24), DOI 10.3389/frobt.2016.00024

OMG (1997a) Corba. http://www.corba.org/, accessed: 2017-01-26

OMG (1997b) DDS Portal – Data Distribution Services. http://portals.omg.org/dds/omg-dds-standard/, accessed: 2017-01-26

Paikan A, Fitzpatrick P, Metta G, Natale L (2014) Data flow port monitoring and arbitration. Journal of Software Engineering for Robotics 5(1):80–88

Paikan A, Pattacini U, Domenichelli D, Randazzo M, Metta G, Natale L (2015) A best-effort approach for run-time channel prioritization in real-time robotic application. Proc IEEE/RSJ International Conference on Intelligent Robots and Systems, Hamburg, Germany

Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng AY (2009) ROS: an open-source Robot Operating System. In: ICRA workshop on open source software, vol 3, p 5

Schlegel C, Worz R (1999) The software framework SMARTSOFT for implementing sensorimotor systems. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, vol 3, pp 1610–1616

Stampfer D, Lotz A, Lutz M, Schlegel C (2016) The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software. Journal of Software Engineering for Robotics 1(7):3–19

Szyperski C (2002) Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Tanenbaum AS, Maarten vS (2006) Distributed Systems: Principles and Paradigms (2Nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA

Tsouroukdissian AR (2014) ROS control, an overview. https://roscon.ros.org/2014static_replacement/wp-content/uploads/2014/07/ros_control_an_overview.pdf, accessed: 2017-01-26

Vahrenkamp N, Wächter M, Kröhnert M, Welke K, Asfour T (2015) The robot software framework ArmarX. Information Technology 57(2):99–111, DOI doi.org/10.1515/itit-2014-1066,

White R, Christensen HI, Quigley M (2016) SROS: securing ROS over the wire, in the graph, and through the kernel. CoRR abs/1611.07060, URL http://arxiv.org/abs/1611.07060

ZeroC Inc (2017) Ice Comprehensive RPC Framework. https://doc.zeroc.com/, accessed: 2017-01-26

Zhao Y, Paine N, Kim KS, Sentis L (2015) Stability and Performance Limits of Latency-Prone Distributed Feedback Controllers. arXiv:150102854 [cs] ArXiv: 1501.02854