Review Article

Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography

Ayssam Elkady and Tarek Sobh

School of Engineering, University of Bridgeport, Bridgeport, CT 06604, USA

Correspondence should be addressed to Ayssam Elkady, ayssam.elkady@gmail.com

Received 21 August 2011; Revised 15 January 2012; Accepted 29 January 2012

Copyright © 2012 A. Elkady and T. Sobh. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Autonomous robots are complex systems that require the interaction between numerous heterogeneous components (software and hardware). Because of the increase in complexity of robotic applications and the diverse range of hardware, robotic middleware is designed to manage the complexity and heterogeneity of the hardware and applications, promote the integration of new technologies, simplify software design, hide the complexity of low-level communication and the sensor heterogeneity of the sensors, improve software quality, reuse robotic software infrastructure across multiple research efforts, and reduce production costs. This paper presents a literature survey and attribute-based bibliography of the current state of the art in robotic middleware design. The main aim of the survey is to assist robotic middleware researchers in evaluating the strengths and weaknesses of current approaches and their appropriateness for their applications. Furthermore, we provide a comprehensive set of appropriate bibliographic references that are classified based on middleware attributes.

1. Introduction

Robot middleware is an abstraction layer that resides between the operating system and software applications (as shown in Figure 1). It is designed to manage the heterogeneity of the hardware, improve software application quality, simplify software design, and reduce development costs. A developer needs only to build the logic or algorithm as a component, after which the component can be combined and integrated with other existing components. Furthermore, if he wants to modify and improve his component, he needs only to replace the old one with the new one. Therefore, experiment efficiency will improve. In [1], the authors outline some of the problems faced in the development of some robotics middleware. A survey of robot development environments (RDEs) by Kramer and Scheutz [2] described nine open source, freely available RDEs for mobile robots, evaluated and compared them from various points of view with suggestions of how to use the results of the survey, and concluded with a brief discussion of future trends in RDE design. Mohamed et al. [3] provide a short overview of several research projects in middleware for robotics and their main objective. Mohamed et al. [4] provide an overview study of networked robot middleware and different criteria for evaluating networked robot middleware. Furthermore, in [5], some freely available middleware frameworks for robotics are addressed, including their technologies within the field of multirobot systems.

This paper is structured as follows. Section 2 presents an overview and the objectives of current middleware solutions. Some attributes, focusing on the architecture, simulation environment, standards and technologies, support for a distributed environment, security for accessing modules, fault detection and recovery, real-time and behavior coordination capabilities, and open-source and dynamic wiring for the most of the existing robotic middleware frameworks, are then discussed in the following sections. Each section describing an attribute for different middleware structures includes an embedded set of the appropriate bibliographic references to provide researchers with easy access to the current state of the art research in the area. The final section summarizes the survey findings.
The primary advantages of the robotics middleware are **Software Modularity**, **Hardware Architecture Abstraction** to hide the low-level device-specific details of the device in order to give the developers more convenient, standardized hardware APIs, **Platform Independence**, and **Portability** to be able to run on any platform with only changing in the system's configuration. Furthermore, the core of the middleware should not depend on the specific device or software algorithm. The middleware should be scalable and upgradable with the growing of its components. The middleware should be easy-to-use, robust, reliable, easy to maintain, efficient flexible, and support for parallelism and distribution systems. It should also allow for changing in the configuration of control flow and the data flow at runtime. It is favorable to be real-time system and provides some security aspects such as authentication, authorization, and secure communication.

### 2. Robot Software

#### 2.1. Middleware

Bakken et al. [6] defined middleware as follows: “a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system.” In this section, several robotics middleware such as Player, CLARAty, Player, Miro, and OpenRTM-aist and their attributes are described. For each attribute, a short description of the relevant research attribute is described and several representative references are provided.

The Player project started at the University of Southern California [8–13]. Player is a “distributed device repository server” [13] for robots, sensors, and actuators. A client program communicates with Player, running on the robot, using a separate TCP socket connection for data transfer. Coupled Layer Architecture for Robotic Autonomy (CLARAty) [7, 14–26] is an attempt by NASA, through collaboration with the California Institute of Technology’s Jet Propulsion Laboratory, Ames Research Center, Carnegie Mellon University, and the University of Minnesota. OpenRTMiast [27–32] is a software platform developed by the National Institute of Advanced Industrial Science and Technology-Intelligent Systems Research Institute.

Miro [33, 34] is an object-oriented robotics middleware developed by the University of Ulm, Germany. Microsoft Robotics Developer Studio (MRDS) [35–37] is a Windows-based environment for robot control and simulation. Marie (Mobile and Autonomous Robotics Integration Environment) [38–41] was developed by the Mobile Robotics and Intelligent Systems Laboratory, University of Sherbrooke, Canada. Orca [42–44], and OPRobotS (Open Platform for Robotic Services) [45–47], designed by the IT R&D program of the Ministry of Knowledge Economy of Korea, are component-based software frameworks. The ERSP Software Development Kit [48] provides technologies for vision, navigation, and system development. Webots [49, 50] is a commercial robot simulation package for fast prototyping and simulation of mobile robots. Robot Operating System (ROS) [51] is a “thin, message-based, peer-to-peer” [52], robotics middleware designed for mobile manipulators.

The **Open Robot Control Software** (Orocos) [53–59] is a Real-Time Toolkit (RTT) that helps the developers to build C++ robotics applications. RSCA (Robot Software Communication Architecture) [60], developed in Seoul National University, is a robot middleware for networked home service robots. Skilligen Robot Behavior Learning System [61], RoboFrame [62–65], SmartSoft [66–69], iRobot AWARE [70], developed by iRobot, and ASEBA [71, 72] are some examples of robotic middleware platforms. RoboFrame [62–65] is an OO middleware developed using C++. Table 1 summarizes the objectives of some commonly used robotics middleware.

#### 2.2. Robot Toolkit

A robot toolkit is a set of software tools used by developers that provides them with the facilities to create their own robot applications, such as CARMEN and Pyro. Carnegie Mellon Navigation (CARMEN) Toolkit [74–76] is an open-source collection of robot control software. Pyro [77–81], written in Python, stands for Python Robotics which provides “a set of abstractions that allows students to write platform-independent robot programs.” [81].

### 3. Architectural Approach

In this section, the architecture of the previously discussed middleware platforms is described. CLARAty’s architecture has two distinct layers (as shown in Figure 2): the Functional Layer and the Decision Layer. The Functional Layer includes a number of generic components such as “digital and analog I/O, motion control and coordination, locomotion, manipulation, vision, navigation, mapping, terrain evaluation, path planning, estimation, simulation, and system behavior” [7]. Furthermore, control algorithms are implemented in this
layer, such as sensor-based manipulation, visual target tracking, and vision-based navigation. “The Decision Layer is a global engine that reasons about system resources, the state of the system and its environment, and mission constraints. It includes general planners, executives, schedulers, activity databases, and rover and planner-specific heuristics. The Decision Layer plans, schedules, and executes activity plans. It also monitors the execution modifying the sequence of activities dynamically, when necessary” [23]. The relation between the Decision and Functional Layers is a client-server model through a publish/subscribe communication scheme.

Webots [82] is organized as an OOP interface for robot control. Objects correspond to robot devices such as differential wheels, camera, distance sensor, laser range-finder, and accelerometer. As described in [64], two mechanisms are implemented in RoboFrame to exchange data between the modules: a blackboard architecture and mutual exclusion mechanisms for large data structure, and a message-based system for smaller data structures.

MIRO [33] consists of three layers (as shown in Figure 3): the MIRO device, MIRO service, and MIRO framework layers. The MIRO device layer, which is a platform dependent, provides interface abstractions for the low-level hardware details (all sensor and actuator devices). MIRO service layer provides service abstractions for sensors and actuators by means of CORBA interface definition language (IDL). “The Miro Class Framework provides a number of often-used functional modules for mobile robot control, like modules for mapping, self-localization, behavior generation, path planning, logging and visualization facilities” [33].

Marie consists of three layers [40]: Application, Component, and Core Layers. Application/Component layers provide some tools for building applications/components. The Core Layer consists of low-level tools for communication, data handling, and I/O control. Furthermore, Marie uses a centralized control unit called the Mediator Design Pattern (MDP) to interact with each application independently, as shown in Figure 4. As described in [38], Marie uses four functional components for interaction and communication between the applications through a centralized control unit (CCU) based on the MDP: Application Adapter (AA), Communication Adapters (CAs), Communication Manager (CM), and Application manager (AM). The AA is responsible for communication between the CCU and applications.
For integration with the system, “each application should have its own AA in order to encapsulate communication mechanisms, services, and configurations. The CA is responsible for translating information between different communication protocols and mechanisms. It allows different applications to exchange data correctly” [40]. The common types of CA are Mailbox (“data buffer between asynchronous components” [40]), SharedMap (“It forwards incoming data to multiple outputs” [40]), Splitter (“a push-in/pull-out data structure used to store system states that can be accessed by many components” [40]), and Switch (“It sends only one of its inputs to the output port” [40]). “The CM is responsible for creating and managing communication links between Application Adapters. The AM is responsible for control and management of the whole system, by coordinating system states. It also configures and controls all components available in the system.” [40].
SmartSoft consists of three layers [67], as shown in Figure 5, Skill, Sequencing, and Deliberation Layers. A module of the Skill Layer, such as the path planner and the mapper, works simultaneously as client and server. The Sequencing Layer selects and synchronizes the required behaviors, according to the actual situation, to be executed. The Deliberation Layer contains time-consuming algorithms such as symbolic action planning.

ERSP [48] consists of the three primary layers: Hardware Abstraction Layer (HAL), Behavior Execution Layer (BEL), and Task Execution Layer. The HAL hides the low-level details of the robot’s hardware and operating system. The BEL contains sensing, decision-making, and autonomous action modules. “The TEL provides a high-level, goal-oriented method of programming and an interface to the BEL” [48]. Pyro consists of several modules, such as direct control, reactive control, behavior-based control, fuzzy logic, and finite state machine. Orca [43] is a component-based middleware. Communication between its components is handled by the Internet Communication Engine (Ice) which is a general-purpose communication middleware [83], as shown in Figure 6.

OpenRTM-aist [28] is a component-based framework. The RT Component may be a device unit, like a servomotor, and a sensor, or a combination of device units, such as a mobile platform and arm or software modules, such as control algorithm or image processing algorithm. The architecture of the RT Component is shown in Figure 7 [28]. Each RT Component has the interfaces, called Ports, to communicate with other components or exchange data. The RT system is constructed by connecting the ports of RT-components.

ORoCoS is a Real-Time Toolkit that provides the components to be able to run on a real-time operating system. As described in [53], it consists of the following libraries: the Orocos Components Library (OCL) that provides some ready-to-use control components, the Orocos Kinematics and Dynamics Library (KDL) that provides the real-time calculation of kinematic chains, and the Orocos Bayesian Filtering Library (BFL), such as Kalman Filters and Particle Filters.

Skilligent [61] is a modular-structured framework. The core of the Skilligent Robot Learning and Behavior Control System consists of various modules, such as Audio Recognition and Robot Learning, and Behavior Execution and Coordination. Furthermore, various external software modules can be attached to the core, as shown in Figure 8.

OPROs is a component-based platform supporting a client/server scheme for control flow and a publisher/subscriber scheme for data/event flow. ROS is a “metaoperating system” [51] for robot software consists of many small tools designed to work together. “The ROS system is a computation graph consisting of a set of nodes communicating with one another over edges” [51]. It consists of nodes (software modules), messages (passed peer to peer), topics, and services (analogous to web services). Nodes communicate together by passing messages through publish/subscribe model. Messages are not based on a specific programming language. “A node sends a message by publishing it to a given topic, which is simply a string. A node that is interested in a certain kind of data will subscribe to the appropriate topic.” [51].

There are four main components in MRDS [36]: Concurrency and Coordination Runtime (CCR), Decentralized Software Services (DSSs), Visual Programming Language (VPL), and Visual Simulation Environment (VSE). The CCR is a .NET-based concurrent library for coordinating between the multiple sensors and robot actuator and managing asynchronous, parallel tasks through messages. DSS is a service-oriented runtime allowing multiple services to work together in order to achieve certain tasks and behaviors. VPL is integrated with Visual Studio to give the developers the facilities to implement a program through drag and drop blocks (activities or services) onto the design surface. A program, implemented with VPL, can collect and process data from the various sensors and create autonomous behavior for any robot that has a distance-measuring device and differential drive system. Finally, VSE is a simulation environment (Figure 9).

Player [13] is a device server (application server) with a collection of dynamically loadable device-shared libraries. It is a queue-based message passing system, consisting of two parts: the Player Core and the Transport Layer. “The core system includes the device and driver classes, the dynamic library loading code, configuration file parsing and the driver registry” [13]. Each driver has a single incoming message queue and can publish messages to the broadcast data and to all other drivers’ subscribed client queues. The core library defines message syntax and coordinates the passing of these messages. The Transport Layer is independent of device drivers and is based on TCP communication protocol using sockets (and message queues).

Irobot Aware is implemented as a data messaging system designed through a publisher/subscriber scheme. As described in [70], Irobotware consists of the following: base tools and component, robotics, and operator control unit (OCU) frameworks. The base tools are open-source software packages dealing with the low-level details of the OS and hardware. Component/Robotics/OCU frameworks provide the software applications with infrastructure for autonomous networked systems/mobile robot-specific applications/real-time OCU controls and displays.

RSCA “consists of a real-time operating system, a communication middleware, and a deployment middleware called core framework” [60]. CARMEN [84] consists of three layers: the base layer to hide the low-level details of the hardware, the navigation layer to provide navigation primitives, such as motion planning, localization, and dynamic object tracking, and “the third tier is reserved for user-level tasks employing primitives from the second tier”.

4. Simulation Environment

The simulation environment is very important for fast prototyping and educational purposes, although it may have some drawbacks and limitations, such as lack of noisy data existing in the realistic world and simulated models that may
be incomplete or inaccurate. Table 2 shows whether or not each middleware supports a simulation environment.

Most of the middleware, such as iRobotAWARE [70], CLARAty [16], Orca, Miro [2], ASEBA, MRDS [35], ROS [51], RoboFrame [63], OpRoS [47], and CARMEN [76], provides simulation environment, although Orocos, Skil- ligent, RSCA, and ERSP do not come with a simulation environment. The Player [9] project provides a graphical, two-dimensional device simulator called Stage, which supports research in multirobot systems by using socket-based communication and a high-fidelity, three-dimensional sim-ulator called Gazebo. In the Player [50], it is easy to simulate nonexistent devices for research in device design. Marie [38] provides interfaces to Stage, Gazebo, the ARIA, and CARMEN simulators. OpenHRP3 is a dynamics simulator based on OpenRTM-aist. Webots is a simulation environment. MuRoSimF is a simulation framework developed in RoboFrame. SmartSoft uses Player/Stage 2D simulator and also Gazebo 3D simulator. Pyro [80] is integrated with several existing robot simulators, including Robocup Soccer, Player/Stage, Gazebo, and the Khepera simulator.

5. Standards and Technologies Used

Table 2 summarizes the standards and technologies used in each middleware. Miro [33] and Orocos [85] are implemented using the Common Object Request Broker architecture (CORBA) standard. CORBA allows interprocess and cross-platform interpretability for distributed robot control. Although Orca [43] uses ICE, CORBA and ICE provide the basic functionality for component interaction. Smartsoft [68] provides two reference implementations, one based on CORBA (ACE/TAO) and one based on ACE only. Webots [50] uses Open Dynamics Engine (ODE) for detecting collisions and simulating rigid body dynamics. In fact, OpenRTM-aist and OPROS implement the same standard, OMG’s Robot Technology Component Standard.

6. Distributed Environment

The different software modules of an application should be able to exchange data and be able to run in different machines, from which each one is able to obtain its maximum efficiency. Table 2 summarizes whether or not each middleware supports distributed environment. In CLARAty, there are modules that support distributed processing: ACE/TAO (CORBA), sockets, published subscribed mechanisms, and so forth, but CLARAty modules cannot be readily distributed without either existing ACE/TAO wrappers or additional software development (clarified via correspondences with the author Dr. Issa Nesnas).
7. Security for Controlling Access

Data transportation and user access should be secure so that no one can control the robots other than the user. Some Middleware platforms, such as Orocos [53] and MIRO [33] (based on CORBA), do not support security for control access, but CORBA [86] itself supports SSL for its communication. Table 2 summarizes whether each middleware supports security capabilities for controlling access.

8. Fault Detection and Recovery

Fault detection and recovery capabilities are necessary to provide the framework with the ability to be used in real, critical situations. A failure in one module should not damage the whole system. There is always the possibility of a fault at runtime. The faults in the robot’s framework should be detected and localized, and also, the robot should be able to complete its mission or at least to proceed to a safe mode. Table 2 summarizes the fault detection and recovery capabilities for each middleware. ORCA, MIRO, OpenRTM-aist, and Player do not provide any explicit fault tolerant capabilities on the system level apart from the exception list, which may indicate service failures. CLARAty [87] offers a broad variety of low- and high-level means for fault tolerant and robust system performance (state monitoring and recovery from some faults, resources checking, state estimation, verification, test, simulation classes, (clarified via correspondences with the author Dr. Issa Nesnas). CARMEN programs are “robust to a variety of failures” [75] (in communications and of other programs). In ROS [51], there is additional fault tolerance as crashes are isolated to individual nodes.

9. Real-Time Capability

Real-time capability of a robot middleware means that the reactivity of a robot is guaranteed by the real-time
system by providing real-time capabilities for the component communication and process in the framework. Table 2 summarizes the real-time capability for each middleware. RSCA [60] provides real-time capability, if the operating environments support RT-CORBA and POSIX RT profile (PSE52) (clarified via correspondences with the author Dr. Seongsoo Hong). In CLARAty, most modules are real time and operate under VxWorks and QNX RTOS (clarified via correspondences with the author Dr. Issa Nesnas).

10. Behavior Coordination

Behavior Coordination is not part of RoboFrame (but robot coordination can be provided by integrating XABSL as a component, running on top of RoboFrame) (clarified via correspondences with the author Dr. Dirk Thomas). In RSCA, the behavior coordination should be implemented as an RSCA application or a set of RSCA components (clarified via correspondences with the author Dr. Seongsoo Hong). Table 2 summarizes the behavior coordination capability of the middleware.

11. Open Source

OPRoS, OROCOS, ROS, OpenRTM-aist, MARIE, ORCA, ASEBA, RSCA, SmartSoft, Pyro, MIRO, and Player are open-source software, although ERSP, Skilligent, Webots, and iRobotAware are commercial software products. MSRDS is commercial but is free of charge for research and hobbies. In CLARAty, most modules have been approved for public release. Only about 10% has been released due to funding constraints (clarified via correspondences with the author Dr. Issa Nesnas). RoboFrame is not open-source but available for research use for free (clarified via correspondences with the author Dr. Dirk Thomas).

12. Dynamic Wiring

This feature allows dynamic configuration of connections between services of components at runtime, making both control flow and the data flow configurable. As described in [88], “the automatic configuration facility in Marie is not available so communication setup should be static. It definitely is not well suited for all the requirements of dynamic scenarios.” Autoconfiguration is not supported in ORCA [42], although ORCA relies on the ICE [83] Naming Service for delivering location transparency. But the dynamic wiring is supported in Orocos and SmartSoft [66] through scripting, XML, and run-time parameter setting. In SmartSoft [69], the wiring pattern provides a consistent mechanism for dynamic wiring of client parts of communication patterns from outside a component. CLARAty partially supports the dynamic wiring, Only some components that required this feature support runtime reconfiguration. But that is not a general rule that applies to all components in CLARAty (clarified via correspondences with the author Dr. Issa Nesnas).

13. Other Platforms

There are several other robotics software platforms available, such as Yet Another Robot Platform (YARP) [89, 90],
Table 2: Comparisons of the attributes of the main robotics middleware.

<table>
<thead>
<tr>
<th>Name</th>
<th>System model</th>
<th>Control model</th>
<th>Fault tolerance</th>
<th>Simulator</th>
<th>Linux Windows</th>
<th>Standards and technologies used</th>
<th>Open source</th>
<th>Behavior coordination</th>
<th>Real time</th>
<th>Distributed environment</th>
<th>Dynamic wiring</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLARAty</td>
<td>2-layer AM; decentralized data; client-server; platform independent abstractions and interfaces for various robotic components for motion control, coordination, mobility, manipulation, perception, estimation, navigation and planning.</td>
<td>Supports adaptation for centralized and distributed control; event driven</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Only cygwin</td>
<td>Partially</td>
<td>Most modules are real time</td>
<td>Yes</td>
<td>Partially</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Player</td>
<td>No particular architectural constraint, client-server, decentralized data.</td>
<td>Not applicable; but on module level can be considered centralized, since it relies on polling model</td>
<td>No explicit fault handling capabilities</td>
<td>Yes</td>
<td>Yes</td>
<td>Stage is a 2D simulator, Gazebo is a 3D simulator</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ORCA</td>
<td>No particular architectural constraint, peer to peer, decentralized data, component-based robotic systems.</td>
<td>Not applicable; not clear on component level</td>
<td>No explicit fault handling capabilities</td>
<td>Yes</td>
<td>Yes</td>
<td>Ice</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MIRO</td>
<td>3 layers, client-server, decentralized data</td>
<td>On object level there is an emphasis on event–driven control</td>
<td>No explicit fault handling capabilities</td>
<td>Yes</td>
<td>Yes</td>
<td>TAO Middleware C++ implementation of the CORBA standard</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Name</td>
<td>System model</td>
<td>Control model</td>
<td>Fault tolerance</td>
<td>Simulator</td>
<td>Linux</td>
<td>Windows</td>
<td>Standards and technologies used</td>
<td>Open source</td>
<td>Behavior coordination</td>
<td>Real time</td>
<td>Distributed environment</td>
<td>Dynamic wiring</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
<td>-----------------</td>
<td>-----------</td>
<td>-------</td>
<td>---------</td>
<td>---------------------------------</td>
<td>-------------</td>
<td>----------------------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>OpenRTM-aist</td>
<td>Component-based frame work, model-driven architecture, platform-independent model (PIM), Platform-specific model (PSM)</td>
<td>Component-based</td>
<td>Supported by RT-component model</td>
<td>OpenHRP3 is a dynamic simulator</td>
<td>Yes</td>
<td>Yes</td>
<td>CORBA</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ASEBA</td>
<td>Event-driven distributed control</td>
<td>Event-based control</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Event-based middleware, Virtual machines Mediator Interoperability technology, ACE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MARIE</td>
<td>Component-oriented engineering approach, 3 layers</td>
<td>Centralized control unit</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>POSIX. 13, CORBA, RT-CORBA vl. 1</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RSCA</td>
<td>Event-based control</td>
<td>Event-based control</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OPRoS</td>
<td>Component-based frame-works, validation/test tools</td>
<td>Client/server mechanism for control flow and the publisher/subscriber mechanism for data/event flow.</td>
<td>Being developed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>event-driven</td>
<td>Yes</td>
<td>No</td>
<td>Being developed</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ROS</td>
<td>Component-based frame-work, publisher/subscriber</td>
<td>Message oriented frame-works</td>
<td>Not Explicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial functions</td>
<td>Message oriented, RPC services</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MRDS</td>
<td>Component-based, REST</td>
<td>Distributed messaging</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>.NET/SOA</td>
<td>Comm.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OROCOS</td>
<td>C++ libraries: OCL, KDL, BFL</td>
<td>Event-based control</td>
<td>No, but it has Orocos Simulik Toolbox</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>ACE/TAO, CORBA</td>
<td>Open source</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Name</td>
<td>System model</td>
<td>Control model</td>
<td>Fault tolerance</td>
<td>Simulator</td>
<td>Linux</td>
<td>Windows</td>
<td>Standards and technologies used</td>
<td>Open source</td>
<td>Behavior coordination</td>
<td>Real time</td>
<td>Distributed environment</td>
<td>Dynamic wiring</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>-------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>-----------</td>
<td>------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>SmartSoft</td>
<td>Service-oriented, component-based software, model-driven architecture, platform-independent model and platform-specific model</td>
<td>Client/server, publish/subscribe, master/slave, arbitrary control models within component hull</td>
<td>Being Developed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Two reference Implementations, one based on CORBA (ACE/TAO) and one based on ACE only</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, runs with RTAI-Linux and QNX</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ERSP</td>
<td>Skilligent</td>
<td>3 layers</td>
<td></td>
<td></td>
<td>No</td>
<td>Yes</td>
<td>TCP Socket interface, Open Dynamics Engine, Ogre 3D</td>
<td>Comm.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Webots</td>
<td>Multiprocess architecture</td>
<td></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Comm.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Irobotaware</td>
<td>Layered architecture</td>
<td>Publish/subscribe, Messaging</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Comm.</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Carmen</td>
<td>3T hybrid architecture</td>
<td>2D simulator</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td>Socket based, using TCP protocol, IPC</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RoboFrame</td>
<td>Message-oriented publish/subscribe and shared memory communication mechanisms</td>
<td>Message-based</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Pyro</td>
<td>Architecture independent</td>
<td></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Socket based using TCP protocol, XML, SOAP, OpenGL, HTTP</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
SPICA [88, 91, 92], BABEL [93, 94], Dave’s Robotic Operating System (DROS) [95], Intelligent Robot Software Platform (IRSP) [96], K-MIDDLEWARE [97], the Washington University Robotics Development Environment (WURDE) [98], OpenRDK [99, 100], OpenJAUS [101], Open Robot Controller Computer Aided Design (ORCCAD) [102–104], Pyro [77–81], Robot Intelligence Kernel (RIK) [105, 106], MissionLab [107–110], and Mobile Robot Programming Toolkit (MRPT) [111].

14. Conclusions and Bibliography

Access Information

In this survey, we outlined the architecture and some important attributes, with the appropriate bibliographic references for most of the existing robotic middleware, such as Player, CLARAty, ORCA, MIRO, UPnP, RT-Middleware, ASEBA, MARIE, RSCA, OPRos, ROS, MRDS, OROCOS, SmartSoft, ERSP, Skilligent, Webots, Irobotaware, Pyro, Carmen, and RoboFrame. All references listed in this paper can be found in http://www1bpt.bridgeport.edu/~aelkady/Survey.bib. They are stored in a BIBTEX format file: survey.bib.

Acknowledgments

The authors would like to sincerely thank Oskar von Stryk (ROBOFrame), Stphane Magnenat (ASEBA), Reid Simmons and Nesnas, Issa (CLARAty), Francois Michaud and Carle Cot (Marie), Ando Noriaki (OpenRTM-aist), Alex Makarenko (ORCA), Richard Vaughan (Player), Jean-Christophe Baille (URBI), Sang C. Ahn (Upnp), Douglas.Few (RIK), and Byoundyoul Song (OPROS), for their help and support while compiling this survey. They also would like to express their gratitude to Christian Schlegel (SmartSoft), Alex Makarenko (ORCA), Kasper Stoy (Player), Olivier Michel(Webots), Ronald (Ark MissionLab), Byoundyoul Song (OPROS), and Dirk Thomas (RoboFrame) who reviewed and provided them with very useful comments regarding Table 2, which summarizes a comparison of the attributes of main robotic middleware designs.

References


