Player 2.0: Toward a Practical Robot Programming Framework

Toby H. J. Collett and Bruce A. MacDonald
Electrical and Computer Engineering
The University of Auckland
New Zealand
{t.collett,b.macdonald} _at_ auckland.ac.nz

Brian Gerkey
Artificial Intelligence Center
SRI, International
Menlo Park, California, USA
gerkey _at_ ai.sri.com

Abstract

Player/Stage has become a de facto standard in the open source robotics community. We describe recent work on restructuring the Player robot device server into a system that more closely aligns with the idea of a robot framework. The general requirements for a robot framework are also discussed.

Player 2.0 is a major improvement in two basic areas, simplicity and flexibility. The driver API has been vastly simplified with many more parts of the communication being hidden from the user. The flexibility of the new system is realised in the library division that allows for different transport layers (or no transport at all), and in the new device address structure that allows for inter-server device subscriptions.

1 Introduction

All work, including robotics research, is impacted by the tools that are used. Good tools simplify common tasks, while bad tools complicate them. The assumptions that are built into a set of tools bias the researcher who uses them toward particular kinds of solutions. The availability of flexible, reliable, and reusable tools for robot programming is crucial to the research community. We call this set of tools a "robot programming framework," or simply, "robot framework." In this paper we present a set of design goals that an ideal robot framework should achieve.

We also describe an implemented system that satisfies many of these goals. In support of our own research, we have developed a set of Open Source tools, distributed by the multi-institution Player/Stage/Gazebo (P/S/G) Project, that a significant portion of the community has found useful and to which many have contributed. The P/S/G tools have become a de facto standard in the Open Source robotics community [Vaughan *et al.*, 2003]. There is a large community built around these tools (over 200 members on the users' mailing list), offering support and improving the code. We have recently completed the overhaul of Player, which is the central tool of the P/S/G system. This restructuring was guided by our experience with the shortcomings of earlier versions and driven by our goal of developing a robot framework. We describe the result of this effort, which is Player 2.0.

2 Related work

IPC (Inter Process Communication) along with TDL (Task Description Language) are two software packages developed at CMU [Simmons and Apfelbaum, 2004]. TDL extends the standard C++ syntax to provide semantics for task management and it uses IPC for sending messages between servers and clients. CARMEN [Montemerlo *et al.*, 2003], also based on IPC, provides a navigation toolkit for robotics. SIMOO-RT [Becker and Pereira, 2002] is an integrated development environment for real-time systems consisting of a software development framework and model editing tools. CLARAty is an architecture designed for robot automation systems from the Jet Propulsion Laboratory [Nesnas *et al.*, 2003]. It uses a two-tiered design to separate functional and decision layers of a robot application. Coolbot [Dominguez-Brito *et al.*, 2004] is a component oriented framework using a port automata model for components. ROCI [Chaimowicz *et al.*, 2003] uses an integrated circuit pin analogy for connecting components, and focuses on programming robot teams.

The research projects above create custom middleware systems for interaction between distributed components. Other projects base their work on existing high-level middleware such as CORBA. MIRO [Utz *et al.*, 2002] is a distributed object-oriented framework for mobile robot application developments. It is based on CORBA and therefore it is also a multi-platform and programming language independent development system. Woo proposed a three-tier software infrastructure [Woo *et al.*, 2003] based on CORBA. It is designed around the CORBA Trader Service and relies on CORBA to provide

the underlying development environment. Kuo proposed a real time software framework for distributed robotics applications, based on real time CORBA [Kuo and MacDonald, 2005]. OROCOS [2005] is a software framework for robotic control software. It aims to integrate low-level real-time services provided by operating systems (OSs) and provides a high-level software framework for robotic control software development. Smartsoft [Schlegel, 2003] provides component communication patterns and dynamic wiring on top of OROCOS. The Orca project [Brooks *et al.*, 2005] is a multi-transport component system recently evolved from OROCOS. Marie [Côté *et al.*, 2004] aims to integrate APIs and middleware for robotics. Web protocols may be used to expose robot services [Jang *et al.*, 2005]. RT-Middleware [Ando *et al.*, 2005] is designed to provide standard interfaces and allow modular software components for robotic systems. It is based on CORBA, with a standardised object model based on ports for input and output, and a standardised overall behaviour described by a general purpose statechart.

2.1 Design goals

A key requirement in promoting software reuse in robotic systems is to loosen the coupling between software modules. Therefore, an ideal framework is designed to support distributed software development, which automatically makes components more loosely coupled [Szyperski *et al.*, 2002]. A framework will contain distributed software components that are designed specifically for supporting robotic application developments such as device drivers, development tools, algorithms, control systems and the human-robot interaction component.

The main purpose of a robot framework is to reduce the time and complexity of the development task. However, there are many other goals for such a framework, including (see also [Kuo and MacDonald, 2005]):

Platform independence. Users should be able to freely choose an appropriate platform for running the applications.

Enhanced Scalability. Support constructs should enhance the scalability of software solutions. The framework should be flexible and extensible.

Development Process Simplification. High-level concepts, useful facilities, and simplified programming interfaces should support robot program developers. Key parts of this involve abstract programming interfaces; the framework should define key object types and the syntax and semantics of interface-specific messages that can pass between them. For example player defines:

- the notion of a "driver" as a module that can produce and consume messages
- the notion of a "device," which is a "driver" bound to an "interface"

Real-time Performance. The framework should be useful for robotic applications with soft or hard real-time requirements.

Integration with Existing Infrastructure. Applications should be able to integrate existing functionality provided by underlying distributed middleware.

Promotion of Software Reuse. The framework should support object-oriented concepts and provide predefined object interfaces so that modules built on top of the framework have a common ground for interaction. In particular the framework should provide an API for the development of drivers, which are meant to be reusable and composable.

Programming Language Independence. Application developers should be able to choose an appropriate programming language.

Transport Independence. Developers should be free to choose an appropriate transport layer for the application and implementation environment.

As a step toward the goal of building a useful robot framework, we have developed Player version 2.0. Our overall focus is to restructure Player to meet the goals of a robot framework — with the exception of hard real-time performance — while providing compatibility with the Player 1.x code base developed by the existing community of Player users.

Given the rapid pace of development and change in the robotics community any system that is to remain relevant for a significant period must be flexible. Given the rapid change of the pool of researchers, existing work must also be easy to maintain. So from a community oriented viewpoint the two key features we need to aim for are flexibility and ease of maintenance, the latter through simplicity and transparency of the developer APIs.

Player 2.0 represents a significant rework of the internal structure of Player. These changes work towards the principles of the distributed robot framework described above, with a focus on the development community's needs. In particular, the changes have focused on allowing more flexible usage than the simple client-server model of the original Player, and a rework of the internal structure of Player to provide a simpler message processing system.

We now give an overview of the original Player architecture, and then present the motivations behind the changes in Player 2.0. The new features and APIs that are now available are then described. Finally a case study of developing a plug-in driver is presented.

3 Original Player Architecture

Prior to 2.0, Player was a network oriented device server [Gerkey *et al.*, 2001]. The Player server is a distributed device repository server that provides clients with network-oriented programming interfaces to access actuators and sensors of a robot. It includes a collection of device drivers for many popular robot hardware devices. Client programs use proxy objects defined in a Player client library to write and read data to and from the desired device. Player 1.x employs a one-to-many client/server style architecture where one server serves all clients of a robot's devices. Accompanying Player are the robot simulators Stage and Gazebo, each of which allows programmers to control virtual robots navigating inside a virtual environment [Gerkey *et al.*, 2003].

Player relies on a TCP-based protocol to handle communication between client and server. The Player protocol is an open, documented standard so clients can be implemented in any language with a TCP/IP library. Indeed, client-side libraries exist in C, C++, Tcl, Java and Python, among other languages. Unlike many proprietary software development kits shipped with robots, Player does not bind to a particular robot system. It has a modular design; programmers can add support for new hardware devices. Player is developed primarily under Linux; however, it also runs on other UNIX variants such as Solaris and FreeBSD that support TCP socket mechanisms and under Windows with Cygwin.

In order to provide a uniform abstraction for a variety of devices, we chose to follow the UNIX model of treating devices as files. Thus the familiar file semantics hold for Player devices. For example, to begin receiving sensor readings, the client opens the appropriate device with `read` access; likewise, before controlling an actuator, the client must open the appropriate device with `write` access. Each device has a single outgoing data buffer and a single incoming command buffer. Both buffers are overwritten by newer values. In addition to the asynchronous data and command streams, there is a request/reply mechanism, akin to `ioctl()`, that clients can use to get and set configuration information for Player devices.

4 Motivation for Changes

The functions that Player must fulfill for the robot community have grown since its conception, as has the range of hardware that is supported. In particular there are now many 'virtual' drivers available for Player, such as navigation algorithms, that were not the core purpose of the original Player architecture.

In general the motivation for the changes in Player 2.0 stems from the needs of a diverse and changing developer community: flexibility of the system and simplicity and transparency of the APIs. Flexibility can be enhanced by moving toward a more general robot framework as described in 2.1, and significant reworking of the driver APIs and internal structure of Player will enhance the simplicity of the development process.

Specifically there are a number of issues that the development community have found with the existing system. Some of these have been temporarily solved with work arounds built into Player 1.x, while several other issues that remain unsolved. The following list contains specific key issues that have motivated the design of Player 2.0:

Client-Server model too restrictive: With the growing number of virtual drivers, and with a move toward large distributed systems of robots and sensors, there is a need for more general arrangements of devices on the network.

Wire data transformations not robust or flexible: In Player 1.x, the driver writer had to deal directly with network layer data marshaling issues, which led to code that is difficult to debug and maintain. In addition to this the restriction to integer formats in the Player wire structures also necessitated the use of inconvenient units.

Complicated driver API: The original API, while it allows for flexible lightweight drivers to be written, was difficult for people new to Player to understand and was another source of bugs. The new driver API provides a minimal set of methods that a driver needs to implement in order to process Player messages.

Single data and command types: A single data and command structure for each interface was found to be too restrictive, and many interfaces soon exhibited workarounds, such as including all possible data (even when some was not needed), or using a discriminated union with a byte describing which data type was actually being transmitted. The new message namespace allows for interfaces to support multiple data types, with the subtype being specified in the Player message header.

Desire for alternative transport protocols: While TCP/IP is suitable for a large number of applications it is not ideal in every case. Other transport layers, such as JINI and CORBA, have been discussed for some time and the new Player structure allows for this to be used in place of TCP. Additionally a monolithic Player is now possible using only internal message passing (no network layer).

5 Player 2.0 library division

Player is now divided into two halves, the core and the transport layer. The Player core is divided into the

core library, `libplayercore`, and the built-in driver library, `libplayerdrivers`. Currently a TCP/IP transport layer has been implemented, and this is provided by the combination of two more libraries: `libplayertcp` and `libplayerxdr`. These libraries will be discussed in more detail below.

The separation of the Player core from the transport layer is one of the key changes in Player 2.0, as it allows for more flexible use of the Player system. For example, Player can now be tightly integrated with a JINI or CORBA transport layer, or alternatively can be run as a standalone monolithic system.

In terms of the driver API the split of the transport layer also offers an enhancement as data marshaling is now the responsibility of the transport layer, not the individual drivers. Data marshaling was consistently a source of bugs in Player 1.x and particularly was a barrier for developers unfamiliar with Player.

5.1 Player core

The Player core library provides the core API and functionality for the Player system. This includes the device and driver classes, the dynamic library loading code, configuration file parsing and the driver registry. In Player 2.0 the core system is a queue-based message passing system. Each driver has a single incoming message queue and can *publish* messages to the incoming queue of other drivers, and to specific clients in response to requests. A driver can also broadcast data to all subscribed client queues. The core library is responsible for coordinating the passing of these messages and defining message syntax. The Player interface specification defines the message semantics.

The change to a queue-based message passing model also vastly simplifies the driver API. There is now a single method a driver must implement in order to communicate with the server. This is much simpler than the multiple heterogeneous queues in Player 1.x.

The Player message structure has also been altered slightly. The addressing system has been expanded to allow for inter-server subscriptions and now consists of four values: host, robot, interface, index. This 4-part address is intended to be useful in a variety of different transports. The message namespace has been been expanded to two layers, with a type and subtype, which formalises a common workaround in Player 1.x of adding a subtype field to the message body. Given the 2-layer message namespace, a device can consume multiple types of commands and can produce multiple types of data. For example, configuration changes, such as a change in sensor pose relative to the robot can be pushed out by the device for consumption by any interested parties.

The built-in Player drivers have been separated into `libplayerdrivers`. The separation of this library is largely from the point of view of sanity, but it also means that if you only wish to use external plug-in drivers you do not need the size overhead of the built-in drivers. Additionally from a distribution point of view updates to the drivers can be distributed separately from the Player core.

Both `libplayercore` and `libplayerdrivers` can be called from Java via automatically generated bindings, which will assist developers wishing to access Player driver code via Java technologies such as JINI or RMI.

5.2 Transport layer

Currently a TCP transport layer is provided, in the form of two libraries: `libplayertcp` and `libplayerxdr`. However alternatives can easily be provided such as CORBA or JINI transports. This section will describe the functionality of the TCP transport layer.

`libplayertcp` itself is relatively simple; it establishes a TCP socket and then handles the transmission of messages. The key responsibility of `libplayertcp` is to move messages between TCP sockets and Player device queues. Incoming messages on sockets are routed to the appropriate queues, while outgoing messages are popped off queues and sent along on the appropriate sockets.

Whereas earlier versions of Player used a custom encoding of messages as packed C structs that contained integers in network byte-order, Player 2.0 uses an open standard called eXternal Data Representation, or XDR [Network Working Group, 1987]. The XDR specifies an efficient, platform-independent encoding for commonly-used data types, including integers and floating point values. As data marshaling is now the responsibility of the transport layer, we have developed a C library, `libplayerxdr`, which performs the XDR data marshaling. `libplayerxdr` provides a single function for each Player message type that packs and unpacks message payloads, converting between native and XDR formats. The functions in `libplayerxdr` are automatically generated from the `player.h` header file, which defines the message structures (similar functions can be automatically generated for new interfaces by writing and processing another header file). As a result, we expect significantly fewer bugs from data marshaling.

The Player XDR library is an implementation of one possible data marshaling library. In particular Player XDR is targeted at marshaling data for the TCP/IP transport layer provided by `libplayertcp`. It is important to note that such marshaling support is not always necessary. For example, when using JINI as a transport, Java's built-in object serialization support can be used to send messages, removing the need for a user-level data marshaling system.

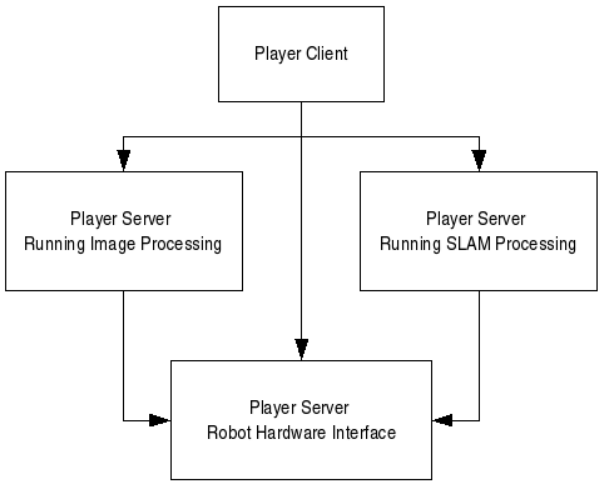


Figure 1: Example of potential Player 2.0 server connections

5.3 New usage paradigms

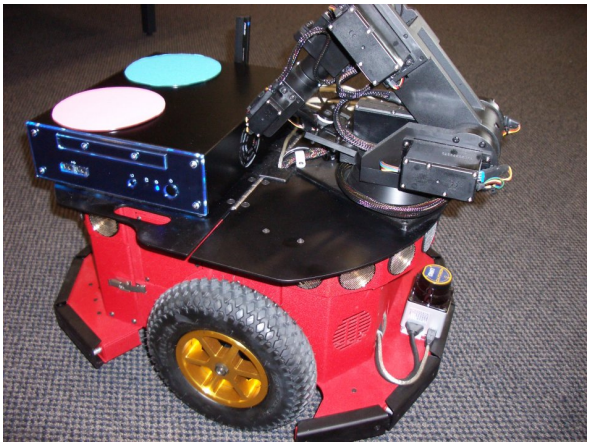
As a result of its new library structure, Player 2.0 can be used in many different ways. Whereas originally Player was, at its core, a TCP-based device server, this usage is now just one of the options open to developers. Other configurations are possible, for example a monolithic application that interacts with the devices directly through their message queues, without any network layer. This application might even be written in a language other than C++, such as Python or Java; bindings in these languages for `libplayercore` and `libplayerdrivers` are currently under development. More complex configurations are also possible, and will become more common as new transport layers are developed.

The most common use of Player 2.0 will likely remain a TCP-based client/server system, which is why we have enhanced functionality along these lines. Player now acts as a distributed framework with servers being able to subscribe to each other to meet the requirements of individual interfaces. There are still some restrictions as there cannot be circular dependencies between Player servers; all dependencies must be able to be resolved on server initialisation. Figure 1 shows an example Player server network, with the arrows indicating device subscriptions.

6 Example plug-in driver for the new API

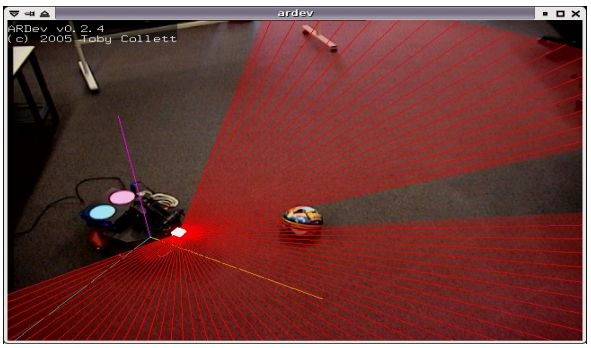
This section describes the process of writing a Player 2.0 plug-in driver. The URG laser scanner from Hokuyo is used as a case study. Figure 2 shows the laser mounted on a Pioneer 3 robot.

The `urg_laser` driver is a simple threaded driver that



(a) URG laser mounted on Pioneer

(b) Laser detail



(c) Laser scan output in Augmented Reality

Figure 2: Hokuyo URG laser scanner

communicates with the laser scanner through a standard USB ACM device (very similar to a standard UART serial port). The main thread of the driver sits in a loop which processes any waiting messages then performs a blocking read on the device to get a laser scan update.

Figure 3 shows the class declaration for the driver. The important features are:

- The driver must inherit from the player *Driver* class.
- The Constructor takes a *ConfigFile* parameter and an integer section parameter.
- The driver must implement the abstract *Setup* and *Shutdown* methods
- The driver re-implements the *ProcessMessage* method to provide support for handling requests and commands
- The driver re-implements Main, which will be called when the driver thread is started.

Figure 4 shows the code needed for `urg_laser` to function as a plug-in driver. `URGLaserDriver_Init` is a factory method that the server calls to create a driver instance and `player_driver_init` is called when the module is loaded to register the driver with the player server core.

Figure 5 shows the implementation of the `urg_laser` methods. Particularly note:

- The method of reading config file parameters in the constructor (this has not changed from Player 1.x).
- The calls to StartThread and StopThread in Setup and Shutdown.
- The ProcessMessage method which is now the single interface point for all driver communications. MatchMessage is used to compare a message definition (type, subtype and address), and Publish is used to post responses.
- The main loop, which processes any pending messages (non blocking), then updates the device data (blocking in this case) and then uses Publish to pass the data onto subscribed devices and clients.

7 Summary

Every research program requires the right tools. In this paper we presented our perspective on the idea of "robot frameworks," which are flexible, reliable, and reusable tools to support robotics research. We presented a set of design characteristics that the ideal robot framework would exhibit.

We also described our recent work on restructuring the Player robot device server into a system that more closely aligns with the idea of a robot framework. Player 2.0 is a major improvement in two basic areas, simplicity

```
#include <libplayercore/playercore.h>

class URGLaserDriver : public Driver {
public:

  // Constructor;
  URGLaserDriver(ConfigFile* cf, int section);
  // Destructor
  ~URGLaserDriver();

  // Implementations of virtual functions
  int Setup();
  int Shutdown();

  // This method will be invoked on each incoming
       message
  virtual int ProcessMessage(MessageQueue* resp_queue,
                             player_msghdr * hdr,
                             void * data);

private:
  // Main function for device thread.
  virtual void Main();

  urg_laser_readings_t * Readings;
  urg_laser Laser;

  player_laser_data_t Data;
  player_laser_geom_t Geom;
  player_laser_config_t Conf;
};
```

Figure 3: Header file for urg laser driver

```
// Factory creation function.
// This function is given as an argument when
// the driver is added to the driver table
Driver* URGLaserDriver_Init(ConfigFile* cf, int section
    )
{
  // Create and return a new instance of this driver
  return((Driver*)(new URGLaserDriver(cf, section)));
}

// Init method called by the module loader
// need the extern to avoid C++ name-mangling
extern "C"
{
  int player_driver_init(DriverTable *dt)
  {
    table->AddDriver("urg_laser", URGLaserDriver_Init);
    return 0;
  }
}
```

Figure 4: Additional code for plug-in module

```
URGLaserDriver::URGLaserDriver(ConfigFile* cf, int s)
: Driver(cf, s, false, PLAYER_MSGQUEUE_DEFAULT_MAXLEN, PLAYER_LASER_CODE)
{
  // Initialise data and process config optinos
  memset(&Data, 0, sizeof(Data));
  // ...

  // read options from config file
  Geom.pose.px = (cf->ReadTupleFloat(s,"pose",0,0));
  Geom.pose.py = (cf->ReadTupleFloat(s,"pose",1,0));
  Geom.pose.pa = (cf->ReadTupleFloat(s,"pose",2,0));
}

URGLaserDriver::~URGLaserDriver()
{ // clean up any resources }

// Set up the device.  Return 0 if things go well, and -1 otherwise.
int URGLaserDriver::Setup() {
  // Start the device thread; spawns a new thread and executes Main
  StartThread();
}

// Shutdown the device
int URGLaserDriver::Shutdown() {
  // Stop and join the driver thread
  StopThread();
}

// Process an incoming Message
int URGLaserDriver::ProcessMessage(MessageQueue* resp_queue, player_msghdr * hdr, void * data)
{
  if(Message::MatchMessage(hdr, PLAYER_MSGTYPE_REQ, PLAYER_LASER_REQ_GET_GEOM, this->device_addr))
  {
    assert(hdr->size == sizeof(player_laser_config_t));
    Publish(device_addr, resp_queue, PLAYER_MSGTYPE_RESP_ACK, PLAYER_LASER_REQ_GET_GEOM, &Geom, sizeof(Geom))
    return 0;
  }
  return -1;
}

// Main function for device thread
void URGLaserDriver::Main()
{
  // The main loop; interact with the device here
  for(;;) {
    // test if we are supposed to cancel
    pthread_testcancel();

    // Process any pending messages
    ProcessMessages();

    // update device data
    Laser.GetReadings(Readings);
    // fill in the data structure
    Data.min_angle = Conf.min_angle;
    Data.max_angle = Conf.max_angle;
    Data.resolution = DTOR(2.0*270.0/768.0);
    Data.ranges_count = (768/2);
    for (int i = 0; i < 768/2; ++i)
    {
      Data.ranges[i] = Readings->Readings[i*2] < 20 ? (4095) : (Readings->Readings[i*2]);
      Data.ranges[i]/=1000;
    }

    Publish(device_addr, NULL, PLAYER_MSGTYPE_DATA, PLAYER_LASER_DATA_SCAN, &Data, sizeof(Data));

    // you may need to sleep here if this loop consumes
    // too much processor time
  }
}
```

Figure 5: Source of urg laser driver

and flexibility. The driver API has been vastly simplified with many more parts of the communications being hidden from the user. This will lead to smaller code that is easier to maintain, and of course fewer bugs. The flexibility of the new system can be seen in the library division that allows for different transport layers (or no transport at all), and in the new device address structure that allows for drivers to maintain subscriptions across the network.

As a demonstration of the simplicity of the new driver API, we have also included an example Player 2.0 driver. We hope to motivate the reader to try the new system and to develop and contribute new drivers for reuse in the community. Player can be downloaded from `playerstage.sourceforge.net`.

Acknowledgment

Toby Collett is funded by a top achiever doctoral scholarship from the New Zealand Tertiary Education Commission.

References

[Ando *et al.*, 2005] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. RT-Middleware: distributed component middleware for rt (robot technology). In *IEEE/RSJ International conference on robots and intelligent systems*, pages 3555–60, Edmonton, August 2005.

[Becker and Pereira, 2002] L. B. Becker and C .E Pereira. SIMOO-RT - An Object-oriented Framework for the Development of Real-time Industrial Automation Systems. *IEEE Trans. on Rob. and Auto*, 18(4):421–30, Aug 2002.

[Brooks *et al.*, 2005] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Towards component-based robotics. In *IEEE/RSJ International conference on robots and intelligent systems*, pages 3567–72, Edmonton, August 2005.

[Chaimowicz *et al.*, 2003] L. Chaimowicz, A. Cowley, V. Sabella, and C.J. Taylor. Roci: a distributed framework for multi-robot perception and control. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 266–271, 2003.

[Côté *et al.*, 2004] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raïevsky, M. Lemay, and V. Tran. Code reusability tools for programming mobile robots. In *Proc IEEE Intl. Conf. on Intelligent Robots and Systems*, volume 2, pages 1820–5, Sendai, Japan, Oct 2004.

[Dominguez-Brito *et al.*, 2004] A.C. Dominguez-Brito, D. Hernandez-Sosa, J. Isern-Gonzalez, and J. Cabrera-Gamez. Integrating robotics software. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3423–8, April 26 – May 1 2004.

[Gerkey *et al.*, 2001] Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Gaurav S Sukhtame, and Maja J Matarić. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

[Gerkey *et al.*, 2003] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, July 2003.

[Jang *et al.*, 2005] Minsu Jang, Jaehong Kim, Meekyoung Lee, and Joo-Chan Sohn. Ubiquitous robot simulation framework and its applications. In *IEEE/RSJ International conference on robots and intelligent systems*, pages 3213–8, Edmonton, August 2005.

[Kuo and MacDonald, 2005] Yuan-hsin (Oscar) Kuo and Bruce MacDonald. A distributed real-time software framework for robotic applications. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'05)*, pages 1976–81, Barcelona, 18–22 April 2005.

[Montemerlo *et al.*, 2003] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) toolkit. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 2436–2441, Las Vegas, NV, October 2003.

[Nesnas *et al.*, 2003] A. Nesnas, I.A.and Wright, M. Bajracharya, R. Simmons, and T. Estlin. CLARAty and Challenges of Developing Interoperable Robotic Software. In *Proc. IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 2428–35, Nevada, Oct 2003.

[Network Working Group, 1987] Inc. Network Working Group, Sun Microsystems. RFC 1014 – XDR: External data representation standard, June 1987.

[OROCOS, 2005] OROCOS. Open Robot Control Software Open Robot Control Services. `http://www.orocos.org`, 2005.

[Schlegel, 2003] C. Schlegel. A component approach for robotics software: Communication patterns in the orocos context. In *18. Fachtagung Autonome Mobile Systeme (AMS), Informatik aktuell*, pages 253–63, Karlsruhe, December 2003. Springer.

[Simmons and Apfelbaum, 2004] R. Simmons and D. Apfelbaum. TDL: Task Decription Language. `http://www-2.cs.cmu.edu/~tdl/`, 2004.

[Szyperski *et al.*, 2002] C. Szyperski, D. Gruntz, and S. Murer. *Component Software*. Addison Wesley, 2nd edition, November 2002.

[Utz *et al.*, 2002] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4):493–7, August 2002.

[Vaughan *et al.*, 2003] R. T. Vaughan, B. P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proc. IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 2421–7, October 2003.

[Woo *et al.*, 2003] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infrastructure. In *Proc. International Conference on Intelligent Robots and Systems*, pages 1475–80, Las Vegas, October 2003.