Journal of Software Engineering for Robotics 5(1), May 2014, 80-88
ISSN: 2035-3928

Data Flow Port Monitoring and Arbitration

Ali Paikan[1,*] Paul Fitzpatrick[1] Giorgio Metta[1] Lorenzo Natale[1]
[1] iCub Facility, Istituto Italiano di Tecnologia, Via Morego, 30, 16163, Genoa, Italy

Abstract—Developing reusable software is hard; systematically developing high quality reusable software components is even harder. Separating computational components from application–dependent functionalities is a key principle for building reusable robotic systems. This work introduces an approach where coordinating logic can be transparently inserted into a reusable component, along with data transforms unanticipated by the component author. Firstly, it proposes the Port Monitor Object which extends a component port's functionality with monitoring and event generation using runtime scripting languages. Secondly, the Port Arbitrator and its application to robotics is presented which enhances a port's capability to arbitrate input data from multiple sources. Lastly, it represents some applications of these approaches to further improve the reusability and robustness of robotics system.

Index Terms—Coordination, Monitoring, Component-based system design, Humanoid robots

1 INTRODUCTION

Writing a reusable software component requires careful design. Two important choices to be made are how the component expresses its output, and what it expects of its input. With reusability in mind, there is pressure to be as generic as possible: to offer everything useful the component "knows" on the output, and to accept all sorts of variants on the input side. However, for any particular application, this generality is decidedly suboptimal. It can result in slow development and higher bandwidth requirements. The opposite approach is to let specific applications drive the development of the component; this may lead to faster development but can seriously limit component reusability. Indeed the "5C" paradigm [1], which is gaining popularity in robotics, dictates separation of concerns between Computation, Communication, Coordination, Configuration and Composition. Following this paradigm we introduce an approach where coordinating logic can be transparently inserted into a reusable component, along with data transforms unanticipated by the component author. With this approach, a robot's coordination system is no longer limited to passively receiving reports from far-flung components in their chosen formats and on their chosen schedule. It can now actively change how data is summarized and how it is communicated. This is achieved by using the component's middleware as a hook to load arbitrary coordination logic into the external-facing interface of components.

Short paper – *Manuscript received November 11, 2013; revised April 30, 2014.*

- *The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 ICT) under grant agreement No. 270273 (Xperience).*

1.1 Motivating Example

In order to clarify the main concern and contribution of our work, we draw a similar example taken from [2]. Figure 1 shows a typical object tracking and reaching scenario, in which the robot is programmed to detect a moving object, to follow that object with its gaze, and to reach for it with its hand. The image data from a pair of stereo cameras are given to two instances of Object Detector each computing the 2D position of the object in the camera frames. These modules feed this information to the 3D Position Estimator module, which performs the required geometric computations to calculate the position of the object in the robot frame, and finally sends those coordinates to the Head Control and Arm Control modules. The latter control the robot's head and arm respectively to look at the object and reach for it.

The overall behavior of the system can be fairly robust if every subsystem behaves as intended. However, some failures or uncertainties in the object detection or 3D position estimation can cause nondeterministic behavior of the robot. Klotzbucher et al. [2] characterize this as a typical coordination problem and propose having a lightweight coordination system using a state machine. The coordinator reacts to explicit events (e.g., events generated by the 3D Position Estimator if the object is not visible to the robot) and changes the state of the system so that an appropriate decision can be made, such as stopping the Arm Control module.

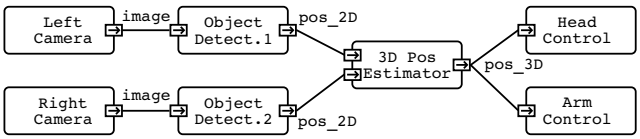


Fig. 1. Data-flow architecture of an object tracking application

To increase the robustness of the coordinator and reusability of the subsystems (i.e., computational modules), Klotzbucher et al. also propose the Coordinator–Configurator pattern to separate the coordinator from the computational module. A pure coordinator requires to be informed via events about relevant changes in system state. Required events for the coordinator can be generated in different ways. One way is to extend the functionality of computational modules and configure them to raise the proper events. For example, 3D Position Estimator can generate different events when the object is not visible or the certainty of detected object drops below a configurable threshold. Klotzbucher et al. argue that this approach can be favorable if the events are computational dependent. On the other hand, if the constraint is application domain specific (e.g., a latency in communication between Object Detector and 3D Position Estimator), this approach severely limits the reusability of the computational module. Moreover, to reuse components with different coordination systems (e.g., event processing in BDI [3]), the events required by the coordinator should match those generated by the computational modules. If this is not the case, the subsystem from one side should be modified.

Another approach is to introduce a separate component which remains between the computational module and the coordinator. The component can act as a monitor which communicates with the computational module and generates events for the coordinator. Alternatively, it can be used to translate existing events into the format which is required by the coordinator. That, in fact, requires implementing an application specific (likely not reusable) module and introduces additional communication and deployment overhead to the system which may not be acceptable in some distributed applications.

To overcome these shortcomings, we propose an approach which is a pragmatic compromise between reusability and performance. In our approach, components' data flow ports [4] are extended with scripting programming language capability. Using the scripting language, a monitor entity can be embedded in the output or input ports of components to monitor data and generate proper events for the coordinator. The approach allows for computational dependent and application specific event generation at the same time, and has the following definite advantages. First, it does not pollute the computational module with application specific details. Depending on the coordination mechanism being used (e.g., state machine, BDI-based system [3]), the required events can be freely generated in the port monitor during application development time. Second, it also simplifies component implementation, since the developer does not necessarily need to be concerned with generating all possible events which can be used in different circumstances. For example, instead of parameterizing Object Detector to generate a coordination event (e.g., "`certainty_low`", "`certainty_high`", "`target_outside_workspace`"), the component can freely output the certainty value (using a separate port or along with 2D position data) which can be used by the port monitor to raise the proper events. Moreover, by embedding monitoring into the port the communication and deployment overhead due to a separate monitor component is no longer introduced in the system.

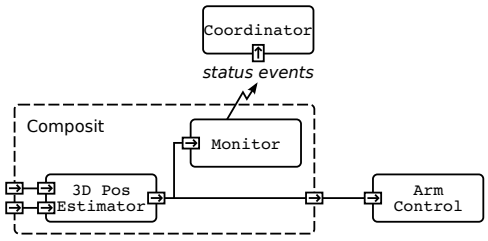
1.2 Contribution and outline

This paper is based on the work presented in [2] and proposes some approaches and guidelines to further improve the reusability and robustness of robotic systems. More specifically, it alleviates the problem of coordination and reusable component development by embedding data monitoring and arbitration into components' data flow ports. The contribution of this work can be divided into three parts. Firstly, it proposes the *Port Monitor Object* which extends a component port's functionality with monitoring and event generation using a runtime scripting language. Secondly, the *Port Arbitrator* and its application to robotics is presented which extends a port's capability to arbitrate input data from multiple sources. Lastly, it represents some guidelines and further applications of these approaches to improve the reusability a of computational component and simplify its implementation.

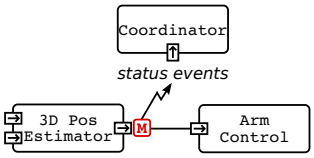
The rest of this work is structured as follows. Section 2 describes the Port Monitor Object and its implementation using the YARP [5] framework. The concept of port arbitration, the architecture and its implementation is described in Section 3. Further potential applications of the proposed approach are explained in Section 4. Section 5 presents some of the related works and provides a short discussion of the proposed approach. In Section 6 we present the conclusion.

2 The Port Monitor Object

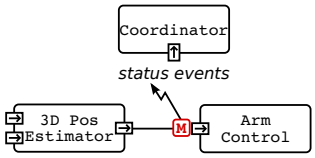
One way to inform a coordinator about state changes of the system is to employ a separate monitor module and configure it with a set of constraints to generate proper events. This is shown in Figure 2(a) for coordinating components of the object tracking example from Section 1.1. The Monitor component receives data from 3D Position Estimator and generates status events for Coordinator. To increase reusability of the composite subsystem in different architectures or with an alternative coordinator, the Monitor module should offer a generic way to be configured with the required constraints for generating events. Although this can be made to work, it can lead to a nonviable software module. Moreover, the overhead



(a) Using separate monitor component



(b) Using Port Monitor in output port



(c) Using Port Monitor in input port

Fig. 2. Different ways to provide required events for coordinator. The boxes indicated with 'M' represent Port Monitor Objects attached to the component interfaces.

of communication and deployment should also be considered in distributed architectures.

Scripting languages have been used for decades to extend the functionality offered by software components without needing to rebuild or even tweak the base system. These extensions are dynamically loaded and plugged into the component at runtime. Using a plug–in system, an alternative approach is to attach a runtime monitor object to the ports of a module. This Port Monitor Object is implemented using a scripting programming language and can be loaded and executed by ports at runtime.

Figures 2(b) and 2(c) elaborate the concept of the Port Monitor Object. As shown in Figure 2(b), the port monitor entity (drawn as a box marked M) is attached to the source side of connection between 3D Position Estimator and Arm Control. Using scripting language, users can develop light–weight code to access and monitor the data which is streamed out through the port. Computationally relevant events are now freely generated in the Port Monitor Object in any required format for the coordinator. Alternatively, one can move the monitor object to the other side of connection and attach it to the input port of Arm Control module (Figure 2(c)). In this way, the connection between 3D Position Estimator and Arm Control can also be monitored and events can be generated in case of delay or failure in communication.

2.1 Reference Implementation

To illustrate the applicability of the described approach, we present an implementation of Port Monitor Objects using the YARP [5] framework. In YARP, programs communicate via units called ports. Messages can be sent between ports,

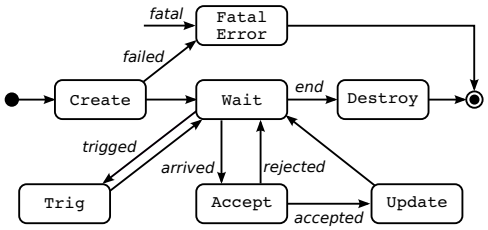


Fig. 3. The life cycle of the Port Monitor Object.

using the connections between them. Connections are not constrained to use the same protocol. The detailed implementation of individual protocols are encapsulated in plug–ins called "carriers". Until recently carriers in YARP have been seen as essentially passive elements, transmitting data in various forms but not actively modifying it. But in fact carriers can be used as hooks that give intimate access to the consumers and producers of data in a network, inserting arbitrary action that is local to a component rather than remote from it. This is the opportunity the Port Monitor Object is building on. A port monitor is implemented[1] as a carrier plug–in which can be attached to one side of a connection and configured to load a user's script. For the time being, only the Lua [6] scripting language is supported but this approach can be easily extended to other languages.

Figure 3 illustrates the life cycle of a Port Monitor Object. A callback function is assigned to each state of the monitor's life cycle (except Waiting) which can have a corresponding implementation in the user's script. Using these callbacks, users have full control over the port's data and can access it, modify it and decide whether to accept the data or discard it. Listing 1 represents the callback functions corresponding to the port monitors' states in Lua.

```
PortMonitor.create = function() return true end
PortMonitor.accept = function(rd) return true end
PortMonitor.update = function(rd) return rd end
PortMonitor.trig = function() return end
PortMonitor.destroy = function() end
```

Listing 1. Port monitor callback functions in Lua

Monitor's life cycle starts with the Create state where the `PortMonitor.create` callback is called. The initialization of user's code can be done here. Returning a true value means that user's initialization was successful and the monitor object can start watching data from the port. When data arrives to the monitor, `PortMonitor.accept` is called. Using a data reader handler passed to the function, the user can access (for reading only) the data, check it and generate events. The return value of this function indicates whether data should be delivered (accepted) or discarded. If data is accepted, `PortMonitor.update` is called, at which point the user has access to *modify* the data.

1. The source code and relevant examples can be found at https://github.com/robotology/yarp/tree/master/src/carriers/portmonitor_carrier

A port monitor will usually act as a passive object [7] in which `PortMonitor.accept` and `PortMonitor.update` callbacks are executed in sequence as data arrives while buffering guarantees that no messages are lost during execution. However, one may need to periodically monitor a connection (within a specific time interval) and, for example, generates proper events in case of delay in the communication. For this purpose, a Port Monitor Object can be configured to call `PortMonitor.trig` within desired time intervals. In Section 4.1 we demonstrate how `PortMonitor.trig` can be used to monitor delays in the communication. Finally, `PortMonitor.destroy` is called when a Port Monitor Object is detached from the port on disconnection.

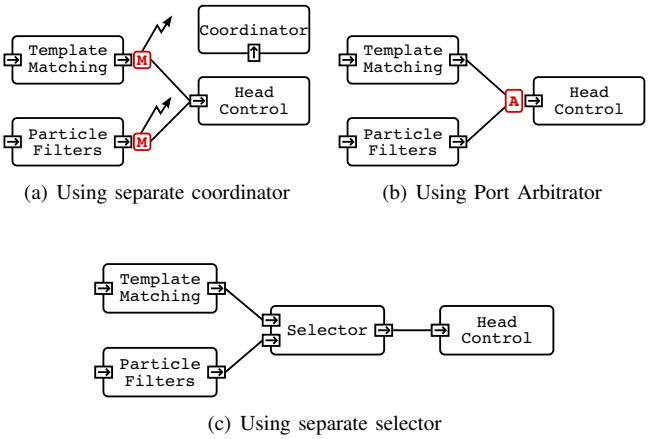
```
1  PortMonitor.create = function()
2     dispacher = yarp.Port()
3     return dispacher:open("/estimator:event")
4  end
5
6  PortMonitor.accept = function(incoming_data)
7     -- read object_pos from 'incoming_data'
8     if object_pos.certainty < 0.8 then
9        dispacher:write(event("e_certainty_low"))
10    end
11    return true
12 end
13
14 PortMonitor.destroy = function()
15    dispacher:close()
16 end
```

Listing 2. An example of monitoring data and dispatching events.

Based on the object tracking example from Figure 2(b), we show how these callbacks can be used to generate events for the coordinator when the certainty of 3D Position Estimator drops below a desired threshold. Listings 4 shows a Lua script which is loaded by the Port Monitor Object attached to the output port of the estimator module. In `PortMonitor.create` a YARP port is created to dispatch events. This allows other modules (e.g., Coordinator) to receive these events by subscribing to this port. Monitoring data and event generation are done in `PortMonitor.accept`. The port's data is read and the condition for generating the event is checked. If the certainty is below the threshold (e.g., 0.8), `e_certainty_low` is generated and published using the dispatcher. Finally, in `PortMonitor.destroy`, the dispatcher port is closed.

3 The Port Arbitrator Object

In robotic applications there are cases where making an immediate decision upon state changes of the system becomes crucial to the overall behavior of the complex system [8]. In terms of coordination, it can be much simpler and more efficient to have a reactive decision made quickly rather than introducing delay in the control loop by making every minor (and sometimes inessential) state change of the system explicitly visible to the coordinator.



(a) Using separate coordinator (b) Using Port Arbitrator

(c) Using separate selector

Fig. 4. Different ways to select desired data from multiple sources. The box indicated with 'A' represents a Port Arbitrator Object.

Figure 4(a) shows a simplified architecture of an object tracking application. Template Matching and Particle Filter modules are configured to recognize a desired object based on different object detection algorithms. The reason for using two different modules for the same purpose is that depending on the environmental condition and how object appears in the scene, one algorithm performs better than the other. Each module sends 3D position of the detected object along with its certainty to the Head Control component. The latter should receive data from the module which is more confident about its result. This is again a problem of coordination. Yet, how to coordinate these competitive modules? It should be clear that delegating this responsibility to the Head Control is not the right choice since it strictly limits reusability of the module.

One solution to this problem is to implement a specific selector which receives data from both detector modules, chooses the one with higher certainty value and sends it to Head Control (Figure 4(c)). The drawback of this approach is that it introduces communication and execution overhead [9] in executing the Selector and transferring data to and from it. Finally, the Selector module is hard to be reused in different applications; unless one makes the selector more generic with the cost of lower performance.

Another solution is to monitor the output values of each module and make any changes in the certainty level visible to a separate Coordinator using proper events (e.g., `e_certainty_low`, `e_certainty_ok`). The Coordinator is then responsible for contacting all the involved modules to inquire each to block or deliver their output to Head Control. This requires extending modules that perform computation and introduces specific logic to enable or suspend sending output data. In case modules are allowed to talk to multiple receivers, this logic should also become aware of the current network topology (for good reasons this information is usually hidden

by the middleware). Moreover, proper coordination requires that a certain amount of messages and acknowledgments are exchanged between the modules involved in the arbitration and the Coordinator. This "bureaucracy" introduces latencies, bandwidth overhead and adds complexity to the application.

For this family of coordination problem, we propose another approach based on arbitrating data from multiple sources in the input port of a component. We call this a *Port Arbitrator*. The approach can be used in the design of any robotic system where immediate reaction to changes in the system's state is required and these minor changes are not necessarily needed to be reasoned about by a third-party component. For the object tracking example, minimizing delays is functionally more important than making every change of the system's state explicitly visible to the coordinator via events (notice that, although these events are not used by a separate component, they can be made available to higher level decision makers or monitors, if required).

Figure 4(b) shows how a Port Arbitrator Object is used in the object tracking example. The port arbitrator (drawn as a box marked A) is attached to the input port of Head Control. The arbitrator is configured with a set of constraints to properly arbitrate between data arriving from the Template Matching and Particle Filter modules. As for port monitors, the arbitrator object can be dynamically loaded and plugged into an input port. Thus the communication and deployment overhead due to separate selector or coordinator components are no longer introduced in the system.

3.1 Internal Architecture of Port Arbitrator

Figure 5 represents the internal architecture of the Port Arbitrator Object. The aim of using a port arbitrator is to allow data from, at most, one connection at a time to be delivered to an input port. A port arbitrator consists of a set of selection constraints, an event container and a selector block.

A Port Monitor Object can be attached to each connection (C_i) going through the port arbitrator. The port monitor can inspect the connection and inserts the corresponding events into a shared container. It can also remove an event (if previously inserted by itself) from the container[2]. Normally events remain valid in the container until they are explicitly removed by the monitor object. An event can also have a specific lifetime: i.e. it will be automatically removed from the container when its lifetime is over. For each connection C_i, there is a selection constraint written in first order logic as a boolean combination of symbolic events. Upon the arrival of data from a connection, the selector evaluates the corresponding constraint and if it is satisfied, it allows the data to be delivered to the input port; otherwise the data will be discarded. Clearly a consistency check on the boolean rules must be performed to guarantee that only a single connection C_i can deliver data at any given time.

2. This is similar to the Event–Mask mechanism used in user interface programming or in operating systems.

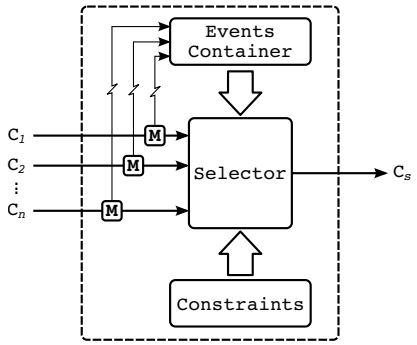


Fig. 5. The architecture of Port Arbitrator object. Straight lines show the data flow and zigzag lines represent event flow.

3.2 Representation and Evaluation of Constraints

We refer to the object tracking example from Figure 4(b) to demonstrate how selection constraints are represented and how they can be evaluated based on events from a container. As we have mentioned before, Head Control should receive data from the detector module which is more confident about its result. The confidence level is indicated by the certainty value sent out from the module to Head Control. A monitor object is attached to each connection. The monitor reads the certainty value associated with the detected object and inserts an event into the container when the certainty is above a desired threshold. The event is removed from the container if the certainty value drops below the threshold. In our example connections are named C_1 and C_2 for Template Matching and Particle Filter, respectively, whereas the corresponding events are `e_template_ok` and `e_particle_ok`.

To allow data from Template Matching (C_1) to be delivered to Head Control when `e_template_ok` exists in the event container we add this rule:

```
C1 if e_template_ok
```

A similar constraint should also be set to receive data from Particle Filter. Suppose now we want to give preference to data from Particle Filter if both trackers are confident about their results. This can be achieved by modifying the selection constraint for C_1 (Template Matching) as follow:

```
C1 if e_template_ok and not e_particle_ok
```

As we have described for the object tracking example, constraints can be expressed as boolean combinations of symbolic events. To evaluate the expression, every symbolic event is substituted with a boolean value. If the event exists in the container, it represents a true value in the expression; otherwise it is evaluated as false.

3.3 Reference Implementation

Port arbitrator extends the port monitor's scripting API for setting constraints and altering events in the container. In fact, when a Port Monitor Object is attached to an input port, the user's script can access the extended API for arbitration. To illustrate this, we show how this extended functionality can be used for arbitrating connections in the object tracking example of Figure 4(b). A monitor object is attached to each connection of the input port of Head Control. Each monitor object loads a script in which we set the constraints described in section 3.2 and it alters the corresponding events by parsing the incoming data and evaluating the associated certainty value. Listing 5 shows the script for setting selection constraint and monitoring data from Template Matching. The selection constraint (i.e., `e_template_ok and not e_particle_ok`) is set in the `create` callback using `PortMonitor.setConstraint`. The certainty value of the detected object is monitored in the `accept` callback. If the certainty is above the desired threshold, `e_template_ok` will be added to the container using `PortMonitor.setEvent`. Similarly, it will be removed from the container using `PortMonitor.unsetEvent` whenever the certainty value drops below the threshold.

```
PortMonitor.create = function()
   PortMonitor.setConstraint("e_template_ok and
                              not e_particle_ok")
   return true
end

PortMonitor.accept = function(incoming_data)
   -- read object_pos from 'incoming_data'
   if object_pos.certainty > 0.8 then
      PortMonitor.setEvent("e_template_ok")
   else
      PortMonitor.unsetEvent("e_template_ok")
   end
   return true
end
```

Listing 5. Monitoring data from Template Matching and setting constraint for arbitration.

Setting the selection constraint of Particle Filter and monitoring its data is done in the same way. Notice that each monitor object can set its own selection constraint and only alters its own events in the shared container.

4 Potential Applications

We have explained how the Port Monitor Object can be used for monitoring data and generating events. We have also shown how this object is extended to instantaneously arbitrate data from multiple connections. In previous work we demonstrated that this arbitration mechanism can be effectively used to implement complex tasks without resorting to centralized coordinators [10]. The Port Monitor Object has also been used within the Xperience project [11] to simplify the design and implementation of a complex application on the iCub [12] humanoid robot. In this work the iCub was programmed to remove objects from a table using a set of functionalities (grasping, pulling objects with a tool and asking for human assistance). Notably these functionalities were not designed to be integrated for this specific application. The Port Monitor Object was therefore a crucial tool to introduce the transformations required for proper integration and to add application specific functionalities (i.e. coordination and data filtering). In the remainder of this paper we show other applications that illustrate how our approach can further improve the performance of robotics system and increase component reusability.

4.1 Monitoring communication for Quality of Service

To achieve robust behavior of a robotic application, the behavior of subsystems and communication among them should be properly monitored. A Port Monitor Object can be attached to an input port to monitor the connection between the components and raise proper events in case of latency or failure in the communication. The events can be used by a coordinator to control urgent critical situations or monitor the Quality of Service (QoS) over longer periods. The events can also be used by an arbitrator to select the component which instantaneously provides data with least latency. In the object tracking example from Figure 4(b), choosing between data from Particle Filter and Template Matching can be done not only based on confidence level, but also by checking which one is producing data with lower latency, higher or just more reliable frequency (i.e. lower jitter). This QoS can vary due to current bandwidth usage in the network connection or computational load of the node in which the module is deployed.

To show how the Port Monitor Object can be used for monitoring communication frequency, we refer to the example from Figure 2(c) and report the pseudo-code of the script to raise an event when data received by Arm Control is delayed for a specific time.

```
PortMonitor.create = function()
   PortMonitor.setTrigInterval(0.2)
   return true
end

PortMonitor.accept = function(incoming_data)
   received = true
   return true
end

PortMonitor.trig = function()
   if received == false then
      --- raise 'e_qos_not_ok' event
   else
      received = false
   end
end
```

Listing 6. An example of monitoring communication for QoS.

As shown in Listing 6, first we setup a trigger to call `PortMonitor.trig` every 200 ms. Whenever data arrives to the monitor object, a flag (`received`) is set. On every call to `PortMonitor.trig`, the flag is checked and if it has not been set, the `e_qos_not_ok` event is generated (for the sake of brevity, the required code for dispatching events is omitted from the listing). The flag is also reset in the `trig` callback for the next check. In this example, the script only raises an event regarding delay in the communication. Noticed that the check performed in this case is overly simplified but this example can be easily extended in a real application. An interesting extension is monitoring failure in the communication and raising proper events using the `PortMonitor.destroy` callback.

4.2 Data Guarding and Filtering

Developing reusable software is hard; systematically developing high quality reusable software components is even harder [13]. With reusability in mind, there is a risk of *over generalization* and increased complexity. In other words, to build a reusable component, the developer tries to foresee any future needs and add them as reconfigurable functionalities to the software. Such a commitment may lead to more complex computational components which are polluted with application–dependent functionalities. Imagine that, for the object tracking example from Figure 1, we want to limit the operational workspace of the robot's arm to reach for the object only in a specific region. One way to achieve this is to configure the 3D Position Estimator module to send object's position if it is within the desired region. The problem is that the output of the estimator module is also used by Head Control, therefore this solution also limits the operational space of the robot's head. Another approach is to delegate this responsibility to the Arm Control module by configuring it to accept the position data if it is within the desired limits. However, if the Arm Control is concurrently used by other modules (which need to control the arm in different workspaces) it should be reconfigured every time it receives data from a different module.

A more flexible approach is to use a Port Monitor Object (e.g., in the output port of 3D Position Estimator) and constrain it to filter the data. This can be done by monitoring data packets in the `PortMonitor.accept` callback and rejecting those that do not satisfy the constraints of the application. In this way, developers are not forced to include any application–dependent functionality into their components. Therefore components can be reconfigured only with the parameters which purely affect their computational functionalities.

4.3 Data Transformation

Components exchange data through their ports. To establish a meaningful communication, components should agree on the type of information they exchange. Brugali et al. [14] classify communication as strongly–typed and loosely–typed and discuss the pros and cons of each category. Strongly–typed communication is more efficient and easier to debug but at the same time it limits reusability of the components. In contrast, loosely-typed communication is more flexible but it requires more manual programing because interpretation of messages should be implemented in the components.

Scripting languages due to their text–processing capabilities have been known to be well suited to the task of data transformation and munging [15]. The port monitor approach allows for data modification (`PortMonitor.update`) using scripting programming languages. For this reason it is potentially an ideal place for basic data conversion. One can attach a Port Monitor Object to an input port of a component and implement a simple script to take the data and convert it into the format which is required by the component. Moreover, using port monitor for loosely–typed data mapping, we simplify the implementation of the components since the latter do not need to bear the responsibility of interpreting information.

4.4 Logging and Performance Monitoring

To analyze the runtime performance of a robotic system, the behavior of components, their interactions and, in general, any critical state changes in the system should be monitored over long periods. This is analogous to the Top–Down passive monitoring in the field of application performance management [16]. Passive monitoring is usually an appliance which leverages network port mirroring. The idea of port monitoring can be applied to record the quality of service provided by a computational component over time. The only way components can communicate with the external world is via their ports. In Section 2 and Section 4.1 we have clearly shown how the Port Monitor Object can be used to generate both computational and communicational events that are locally recorded or sent to a central event logger for off–line analysis.

5 Discussion

The core concept of port arbitration has some similarities with the coordination models and languages in the context of parallel and distributed systems [17]. A coordination model provides a framework based on global communication abstractions in which the interaction of active and independent entities can be expressed using a specific language [18]. There are also some coordination tools which have been explicitly designed for component–based systems such as Reo [19] and ToolBus [20]. Our approach aims at facilitating and simplifying component development by removing not only coordination–related, but possibly all application–dependent functionalities from the component implementation.

Concerning the generality of our approach, it is fair to say that for the port monitor to be effective in all applications the

components need to expose all data through their ports. To our knowledge, however, there is no universal solution to this problem.

The current implementation of the Port Monitor Object exploits a scripting language to achieve better flexibility, easier deployment and implementation of the required functionalities. However the Port Monitor Object can be implemented using a compiled language if higher performance is required (e.g., for real–time processing).

To demonstrate our approach and its potentials we presented a reference implementation using the YARP framework. However, the Port Monitor Object can be implemented in other distributed frameworks, for example by extending existing connection ports (e.g. topics in ROS [21] or buffered ports in OROCOS [22]) with the functionalities required to load code and execute it to parse incoming or outgoing data. This extension can be easily implemented if the framework provides a callback mechanism or by actively monitoring a port using a dedicated thread.

Even though coordination based on port arbitration can cover a wide variety of robotic applications, we have experienced certain limitations in the system. First, since arbitration is usually done on the data from unidirectional connection to an input port, it cannot be easily used in a service–oriented system where interactions between modules are bidirectional and done using blocking remote procedure calls. Moreover, a robotic task might require performing a sequence of actions synchronized with the internal state of components. This can be also made using port monitoring and arbitration, nevertheless, delegating this responsibility to a dedicated, external component can be preferable in favor of simplicity and performance.

6 Conclusions

This article has introduced port data monitoring and arbitration to alleviate the problem of coordination and facilitate development of reusable components. We have illustrated the Port Monitor Object and how it extends a component port functionality with monitoring and event generation using runtime scripting languages.

We have shown that our approach allows separating the computation from application dependent code. This increases the reusability of the components and it simplifies their implementation. We have also demonstrated how the Port Monitor Object can be used to implement data filtering and transformation, quality of service as well as performance monitoring. Overall this can substantially improve the robustness of robotics application.

We have illustrated the concept of port arbitration and its application to robotics. Our approach to port arbitration can also contribute to improving the performance of a robotic system when changes to the system's state can be kept local to certain components and immediate reaction is required.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 ICT) under grant agreement No. 270273 (Xperience) and 611832 (WALK-MAN).

References

[1] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 1, pp. 100–112, 2010. 1

[2] M. Klotzbücher and H. Bruyninckx, "Coordinating Robotic Tasks and Systems with rFSM Statecharts," *Software Engineering for Robotics*, vol. 1, no. January, pp. 28–56, 2012. 1.1, 1.2

[3] M. Georgeff, B. Pell, and M. Pollack, "The belief-desire-intention model of agency," *Intelligent Agents V: Agents Theories, Architectures, and Languages*, pp. 1–10, 1999. 1.1, 1.1

[4] A. Shakhimardanov, N. Hochgeschwender, and G. K. Kraetzschmar, "Component models in robotics software," *Proceedings of the 10th Performance Metrics for Intelligent Systems Workshop on - PerMIS '10*, p. 82, 2010. 1.1

[5] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006. 1.2, 2.1

[6] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "Lua–an extensible extension language," *Software: Practice & Experience*, vol. 26, no. 6, pp. 635–652, 1996. 2.1

[7] K. Murata, R. Horspool, E. Manning, Y. Yokote, and M. Tokoro, "Unification of active and passive objects in an object-oriented operating system," *Object-Orientation in Operating Systems, 1995., Fourth International Workshop on*, pp. 68–71, 1995. 2.1

[8] F. Mastrogiovanni, A. Paikan, and A. Sgorbissa, "Semantic-Aware Real-Time Scheduling in Robotics," *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 118–135, Feb. 2013. 3

[9] D. M. Dhamdhere, *Operating Systems: A Concept-based Approach, 2E*. Tata McGraw-Hill Education, 2006. 3

[10] A. Paikan, G. Metta, and L. Natale, "A port-arbitrated mechanism for behavior selection in humanoid robotics," *16th International Conference on Advanced Robotics (ICAR), 2013*, pp. 1–7, 2013. 4

[11] G. Metta, T. Asfour, A. Ude, and J. Piater. Deliverable 1.2.1: Blueprint for the Cognitive Architecture. [Online]. Available: http://www.xperience.org 4

[12] G. Metta, G. Sandini, and D. Vernon, "The iCub humanoid robot: an open platform for research in embodied cognition," *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pp. 50—-56, 2008. 4

[13] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, p. 120, 1995. 4.2

[14] D. Brugali and P. Scandurra, "Component-based Robotic Engineering Part I : Reusable building blocks," vol. XX, no. 4, pp. 1–12, 2009. 4.3

[15] D. Cross, *Data munging with Perl*. Manning Publications, 2001. 4.3

[16] L. Dragich. (2008) The Anatomy of APM. [Online]. Available: http://apmdigest.com/the-anatomy-of-apm-4-foundational-elements-to-a-successful-strategy 4.4

[17] D. Gelernter and N. Carriero, "Coordination languages and their significance," *ACM CACM*, vol. 35, no. 2, p. 96, 1992. 5

[18] E. Denti, A. Natali, and A. Omicini, "Programmable coordination media," *Coordination Languages and Models*, pp. 274–288, 1997. 5

[19] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, 2004. 5

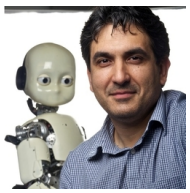
[20] J. A. Bergstra and P. Klint, "The toolbus coordination architecture," in *Coordination Languages and Models*. Springer, 1996, pp. 75–88. 5

[21] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009. 5

[22] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation*, vol. 3. IEEE, 2001, pp. 2523–2528. 5

Ali Paikan Ali Paikan is currently a Postdoctoral researcher at the Istituto Italiano di Tecnologia (IIT) in the iCub Facility department. He accomplished his Ph.D. in Robotics at the IIT in 2014 and he received the double M.S. degree from the University of Genova, in 2010, Italy and from the Ecole Centrale de Nantes, France in 2009, within the joint European Master on Advanced Robotics (EMARO). During his research period (2005 - 2007) at the Mechatronics Research Laboratory (MRL), Iran, Ali was actively involved and awarded in various RoboCup competitions. He has an extensive background in robotics and his main research interests include software architectures for robotics, software reusability and real-time systems.

Paul Fitzpatrick has a background in artificial intelligence, robotics, and engineering, with a PhD in Artificial Intelligence from MIT, and a BEng in Computer Engineering from the University of Limerick, Ireland. He has worked as a software engineer in the fields of robotics, process control and automation, and static verification. Paul grew up on a small farm in Ireland. There were goats involved. He now lives in Montclair, New Jersey. There are fewer goats involved.



Giorgio Metta is the director of the iCub Facility. He holds a MSc cum laude (1994) and PhD (2000) in electronic engineering both from the University of Genoa. From 2001 to 2002 he was postdoctoral associate at the MIT AI-Lab. He is assistant professor at the University of Genoa since 2005 and with IIT since 2006. He is Professor of Cognitive Robotics at the University of Plymouth (UK) since 2012. Giorgio Metta's research activities are in the fields of biologically motivated and humanoid robotics and, in particular, in developing humanoid robots that can adapt and learn from experience. His research developed in collaboration with leading European and international scientists from different disciplines like neuroscience, psychology, computer science and robotics. He is an author of approximately 250 publications. He has been working as principal investigator and research scientist in about a dozen international as well as national funded projects.

Lorenzo Natale Dr. Lorenzo Natale is presently Researcher at the IIT. He received his degree in Electronic Engineering (with honors) and Ph.D. in Robotics in 2004 from the University of Genoa. He worked at the LIRA-Lab at the University of Genoa and was later postdoctoral researcher at the MIT-CSAIL. In the past ten years Lorenzo Natale worked on various humanoid platforms. His research interests range from sensorimotor learning and perception to software architectures for robotics. He has been key researchers in several EU funded projects and authored about 70 peer-reviewed publications. He is associate editor of the International Journal of Humanoid Robotics and the International Journal of Advanced Robotic Systems.