Exploiting data flow ports for coordination of concurrent components

Ali Paikan, Giorgio Metta and Lorenzo Natale

Abstract—Software engineering and best practices in robotics promote modularity and composability in the attempt to reduce development time and improve maintainability of software. This, however, leads to an increased complexity of the system and in the effort required to properly coordinate the interaction between components. This short article proposes a mechanism for coordination of components in distributed architectures based on port arbitration and exploiting the same connections already used to transfer data. Our approach i) intrinsically reduces the number of links required for coordination and ii) it relies on standard data flow port offered by the middleware. Thus, make it suitable to be adopted by different robotic frameworks.

I. INTRODUCTION

Different coordination models and languages such as data–driven, event–driven, centralized and decentralized are studied in [1], [2] and [3]. Robotic middlewares and frameworks usually use architecture–dependent mechanism (e.g. BIP in GenoM [4], [5] ) or adopt some standard approaches such as finite state machines (e.g. rFSM in Orocos [6], SMACH in ROS [7]) and petri net (e.g. RoboGraph [8], RTM [9]) to properly orchestrate the interaction of components. This, however, requires that some specific features are implemented in the underlying component to be properly employed by the coordination system and therefore, it places burden on adopting approaches from other robotic frameworks. This is probably one of the reason that caused the development of different component models (which is also one of the issues addressed by BRICS [10]).

A quick review of components model offered by different robotic middlewares (e.g. OROCOS, OPROS, OpenRTM, Rt-Component, YARP, ROS) shows that using data flow ports [11] (or decoupled communication port [2]) to stream data asynchronously, has become a standard in most of the robotic frameworks. This, in fact, encouraged us to investigate the possibility of exploiting the connections among components for their own coordination.

We focus on the typical scenario of a software architecture developed using one of the robotic middleware commonly used in the literature [12]: The output of a component can be connected to one or more input ports of other components. It is also possible to connected multiple outputs to an input of a component. We also make the following assumption:

- Data is streamed out through outputs if and only if the computed data is valid. For example, an object detector sends object position information through its output port only if the object has been detected.

In the example from Figure 1, Face Detector and Object Detector can both send 3D position information to Gaze Control which controls the robot’s head to gaze accordingly. Components are running in parallel. In term of implementation, they can be distributed over different machines which communicate through network interfaces. Since there is no synchronization among behaviors, data can be delivered to an input port at any time, potentially causing race conditions. For example, in a simple scenario where a person keeps an object in front of the robot, Face Detector conflicts with Object Detector by spontaneously sending information to Gaze Control.

II. COORDINATION MECHANISM

In our approach, every connection in an input port, has a stimulation level which is accumulated using a specific type of leaky integrator whenever data arrives to the port through this connection (similar to the artificial neuron model). When the stimulation level reaches its threshold, the connection is in active state and can deliver (fire) its data (see Figure 1). To arbitrate between simultaneously activated connections, each connection has a set of weighted links (similar to synaptic weights in neural networks) to the other connections. The stimulation level which is accumulated using a specific type of leaky integrator whenever data arrives to the port through this connection (similar to the artificial neuron model). When the stimulation level reaches its threshold, the connection is in active state and can deliver (fire) its data (see Figure 1). To arbitrate between simultaneously activated connections, each connection has a set of weighted links (similar to synaptic weights in neural networks) to the other connections.

Fig. 1. Avoiding race conditions of competitive connections to an input port using port-based arbitration mechanism.
to zero. An active connection $C_i$ can inhibit or excite other connections using its bias $\beta_i$ and weights $\alpha_{ij}$.

Upon receiving data from connection $C_i$ (at time $t$) to the input port, the latter should decide whether to accept the message or discard it. First, it updates the stimulation levels of all $m$ connections ($x_1[t] \ldots x_m[t]$) to calculate the active inputs ($x_1 \ldots x_m$) using Equation 1.

\[
x_i = \begin{cases} 
1 & \text{if } C_i \text{ is active}, \\
0 & \text{otherwise}.
\end{cases}
\] (1)

\[
y_i = x_i \sum_{j=1}^{m} (\alpha_{ji} y_j) + \beta_i, \quad (i \neq j)
\] (2)

Next, outputs ($y_1 \ldots y_m$) are calculated using Equation 2, as well as the maximum output $y_k$. If $y_k > 0$ (i.e. it is not inhibited) and $i = k$ (i.e. connection $C_i$ has the maximum output value), $C_i$ is chosen by the selector and its message will be delivered to the behavior; otherwise it will be rejected.

Imagine in the example from Figure 1, we want the robot to gaze at a face if there is no object in the scene. In other words, if $C_2$ is active, $C_1$ should not be selected. This can be done by self-biasing both connections (e.g. $\beta_1 = \beta_2 = 1$) and let $C_2$ (when it is active) to inhibit $C_1$ ($\alpha_{12} = 0, \alpha_{21} = -1$).

A connection can also be configured as an auxiliary connection. As can be inferred from its name, data from an auxiliary connection will never be delivered to the component by the input port thus cannot conflict with other connections. As such it is only used to regulate other connections. For example, imagine in the Figure 1, we want the robot to gaze at an object if there is also a person in the scene. That is $C_2$ is not initially biased ($\beta_2 > 0$) and it should be selected if $C_1$ is also active. Notice that in this example, we are not interested in tracking the face. Therefore, $C_1$ should be set as an auxiliary connection to excite $C_2$ by choosing $\alpha_{12} = 1, \alpha_{21} = 0$.

We implemented our approach using the YARP middleware [12] and tested it by developing a complex behavior on the iCub humanoid robot [13] in which the robot is programmed to search for an object, grasp it and then look for a person and finally return the object to him (Figure 2). This behavior uses 10 components which are coordinated using our approach. We show that our approach allowed us to implement the behavior completely out of simple existing components and without the need to develop a special purpose component responsible for coordination. We also demonstrate that the final behavior is intrinsically robust to unexpected events (e.g. if the object is dropped from the hand, the robot interrupts the ongoing action and attempt to re-grasp the object). More importantly we developed our behavior incrementally.

III. CONCLUSION

This article has briefly introduced a coordination mechanism in distributed architectures based on port arbitration. One of the advantages of our approach is that

\[ The details and video of the experiment can be provided on demand.\]