DyKnow: A Dynamically Reconfigurable Stream Reasoning Framework as an Extension to the Robot Operating System

Daniel de Leng[1] Fredrik Heintz[1]

***Abstract*— DyKnow is a framework for stream reasoning aimed at robot applications that need to reason over a wide and varying array of sensor data for e.g. situation awareness. The framework extends the Robot Operating System (ROS). This paper presents the architecture and services behind DyKnow's run-time reconfiguration capabilities and offers an analysis of the quantitative and qualitative overhead. Run-time reconfiguration offers interesting advantages, such as fault recovery and the handling of changes to the set of computational and information resources that are available to a robot system. Reconfiguration capabilities are becoming increasingly important with the advances in areas such as the Internet of Things (IoT). We show the effectiveness of the suggested reconfiguration support by considering practical case studies alongside an empirical evaluation of the minimal overhead introduced when compared to standard ROS.**

I. INTRODUCTION

Traditional robot applications assume that their internal processing configuration is fully known at compile-time; both the sensors and the refinement of the sensor data is assumed to be fixed. However, these assumptions are becoming increasingly unreasonable. Autonomous systems are getting access to computational resources (e.g. sensing and processing capabilities) both from other autonomous systems as well as other information services. The *Internet of Things* (IoT) has the potential to greatly increase the number of external resources available to autonomous systems, and advancements in the area of the *Semantic Web* (SW) make available knowledge bases that were previously unintelligible to machines by providing semantic attachments. Clearly, we can expect future scenarios to increasingly involve computational resources that are external to our robot applications, and the availability and quality of which varies over time. For the area of stream reasoning, that is incremental reasoning over incrementally available information, this highlights the need for the capability to reason about which streaming resources to subscribe to.

This paper presents the architecture of the *DyKnow* reconfigurable stream reasoning framework. DyKnow is an extension to the *Robot Operating System* (ROS) [1], which is a popular robot middleware used frequently in both industry and academia. DyKnow is capable of reasoning about which streams to subscribe to and can reconfigure the system during run-time. The motivation for enhancing this control is grounded in the need to reconfigure a system based on the computational resources available to it. Since the standard ROS services do not allow this degree of configuration control, DyKnow efficiently complements these services with its own for a high degree of configuration control. We show that this extension to ROS allows us to dynamically reconfigure ROS applications with minimal overhead in terms of delays and developer burden.

The remainder of this paper is organized as follows. In section II, we cover previous and related work in the area of stream reasoning and reconfiguration. We then present the DyKnow architecture in section III. This is followed by an overview in section IV of the relevant services ROS offers and how DyKnow complements these services with its own. To demonstrate the usability and ease of adoption of the ROS extension, we describe two distinct case studies in section V. This is followed by a quantitative evaluation in terms of performance in section VI. Finally, we conclude the paper in section VII.

II. RELATED WORK

A lot of progress has been made in the area of stream reasoning, covering both Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) approaches. A comprehensive survey on these approaches is presented in [2]. The DyKnow stream reasoning framework has been used for research into stream reasoning for several years. We distinguish between three generations of DyKnow, where the version presented in this paper is part of the third generation. The first generation of DyKnow [3], [4] was implemented using the Common Object Request Broker Architecture (CORBA). It primarily focused on the manipulation and abstraction of streaming data. The second generation marked a change in research focus towards semantic information integration [5] [6] which coincided with the switch from CORBA to ROS. These efforts are closely related to the OWL-S [7] service ontology which introduces semantic descriptions for services, and work [8] on ontology-based access to streaming data sources. The third generation of DyKnow seeks to expand upon the previous work on DyKnow in part by considering automatic reconfiguration based on the needs of a robot system. Work in the area of the Semantic Sensor Web, such as the Semantic Sensor Network ontology (SSN) [9], focused on well-structured semantic descriptions of sensors and share commonalities with the problem of automatic configuration. The SSN ontology was based on the Sensor Web Enablement (SWE) initiative of the Open Geospatial Consortium (OGC). At around the same time, an approach to 'semantically-enabled sensor plug and play' was proposed [10], [11], using a sensor bus that allowed

[1]Department of Computer and Information Science, Linköping University, Sweden, email: {daniel.de.leng, fredrik.heintz}@liu.se

for automated matching between sensors and SWE services based on features of interest.

The SAMSON Wireless Sensor Networks (WSNs) middleware [12] is similar to DyKnow in considering a dynamic environment in which a network can be reconfigured to deal with changes, albeit at a lower level. In the case of SAMSON, these changes include faults, but also disconnection and power concerns. A survey of other recent work towards WSN middlewares is presented in [13].

The work by Tang and Parker [14] on ASyMTRe is an example of a system geared towards the automatic self-configuration of robot resources in order to execute a certain task. Similar work was performed by Lundh, Karlsson and Saffiotti [15] related to the Ecology of Physically Embedded Intelligent Systems [16], also called the PEIS-ecology.

The main differences between these approaches and DyKnow is that: 1) DyKnow focuses on system reconfiguration towards information acquisition rather than actions in the physical world; 2) DyKnow extends ROS which is the de facto standard middleware for robot applications making it easily accessible; and 3) DyKnow raises the abstraction level to handle tagged streams of information and focuses on applications that subscribe to information by its semantics rather than its (fixed) resources.

III. ARCHITECTURE

DyKnow is tasked with the configuration and maintenance of *streaming resources* in a robot system. In this context, streaming resources can be sources of incrementally-available information or transformations over such information streams. Concretely, this means DyKnow will reconfigure a system during run-time to generate and maintain streams of interest to its clients. These clients can be human operators, other autonomous systems, or computational resources that run within DyKnow's own environment. Consider for example a scenario in which a robot is tasked with tracking a ball in a lab. It can initially do so using its own camera and ball detection capabilities. However, once the robot runs low on power it needs to leave the area to recharge. In the meantime, it can instead request video footage of the lab from a ceiling camera, while applying its own tracking software. This means that the tracking task can be fulfilled even if some of the original streaming resources are no longer available, which is a powerful ability when operating in a dynamic environment.

This DyKnow environment consists mainly of two kinds of entities; the DyKnow manager and any number of available computational resources that consume and produce streams of information. Figure 1 shows a high-level overview of these components. The DyKnow manager keeps track of the state of the environment in terms of which computational resources exist and how they are connected, and has the ability to change these connections. Developers provide specific instances of these computational resources, e.g. detectors and trackers. Clients, be it human operators or other devices, can communicate with the DyKnow manager to request changes to the configuration, for example to acquire a particular kind of information. By changing the connections between resources and streams during run-time, DyKnow can automatically adapt to changing circumstances.

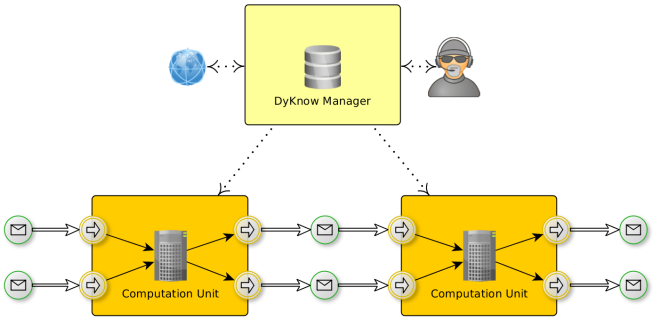

Fig. 1. Conceptual overview of computational resources (computation units) and the DyKnow manager. The parallel thick arrows represent information flows between computation units. The encircled arrows represent ports that connect the internal processing of computation units to streams for receiving or publishing. The dashed lines represent service-based interactions.

A. DyKnow Requirements

Stream reasoning systems should be able to at least deal with large volumes of high-velocity data. A number of additional requirements were taken into consideration during the development of DyKnow:

- **Usability:** In order to support more control over a system's configuration, some extra work is needed by a developer. We want to try and minimize this extra burden.
- **Adoptability:** Given an existing implementation in ROS, it should be easy to convert it to work with DyKnow. 'Easy' in this context means the changes a developer would have to make to the existing software before doing the extra work described above.
- **Performance:** There is an overhead associated with augmenting the control services that exist in ROS. This overhead should be relative to the degree of control a developer wants to support. Further, effects to message throughput should be kept low.
- **Reconfigurability:** This is the key feature that DyKnow seeks to support and which is currently lacking for ROS. We therefore focus on the reconfiguration of how components are connected to each other, allowing us to change the flow of information in the system.

B. ROS Preliminaries and Shortcomings

ROS is a popular middleware for robot applications that allows developers to write implementations as ROS *nodes*, which can communicate with each other by using *services* and *topics*. These nodes are combined into *packages*, of which many have been made publicly available. Topics can be used to connect nodes to establish a flow of information, which makes them the implementation counterpart to the concept of information streams. Topics are advertised by *publishers* and can be subscribed to by other nodes using *subscribers*, such that a single topic can have multiple publishers

and subscribers. Services allow nodes to advertise functionality to other nodes, which can then be requested by these nodes. Services (optionally) take a number of arguments and can (optionally) return a result to the service caller. ROS uses *Node Handles* to expose its API to developers of nodes, which packages such as *Image Transport* augmentation to support efficient image transportation.

Where standard nodes correspond to individual processes when run, *nodelets* are run on threads within the process of a *Nodelet Manager* node. Communication between nodelets is generally more efficient than between nodes. Further, nodes are instantiated either manually or through a launch file, whereas nodelets can also be instantiated using the Nodelet Manager's services. This makes it possible for programs to instantiate other programs at will. The disadvantage of using a Nodelet Manager however is that the thread pool is shared among all nodelets, and if a single nodelet crashes it takes down the entire Nodelet Manager.

Nevertheless, the flexibility offered by the Nodelet Manager makes it an excellent tool for dynamically reconfiguring a ROS system, given that nodelets are used. Unfortunately the services offered by a Nodelet Manager are limited to the loading and unloading of nodelets. DyKnow therefore complements these services with the help of persistent *nodelet proxies* that augment the ROS Node Handle.

C. Nodelet Proxy

The persistent *Nodelet Proxy* is the key component that allows DyKnow to exert a greater control over augmented (DyKnow-enabled) nodelets. A developer establishes a nodelet proxy by creating a DyKnow variant of the Nodelet Handle. Recall that the ROS Nodelet Handle serves as an API that can be used to call ROS functionality, such as creating publishers and subscribers. The DyKnow Node Handle instead delegates these calls to the Nodelet Proxy, which either delegates to the ROS Node Handle or to custom DyKnow variants. Specifically, DyKnow provides its own publishers and subscribers that can be used as ordinary ROS publishers and subscribers. The key difference lies in the distinction between ports and topics.

ROS publishers and subscribers connect directly to topics; a subscriber can name a topic and a callback method, whereas a publisher can name a topic and a message to be sent. The DyKnow variants replace topics with ports. The Nodelet Proxy maintains a mapping between ports and topics, and allows for this mapping to change as the result of services that are offered by the proxy. This way, ports can be associated with different topics over time, which allows for run-time reconfiguration to occur.

A schematic of the Nodelet Proxy and its relation to a host nodelet is shown in Figure 2. The nodelet implementation by a developer is indicated by NodeletImpl, which extends ros::Nodelet. For the DyKnow integration to work, the developer needs to create a dyknow::NodeHandle, which takes a ros::NodeHandle as an argument. From that point on, any calls previously done to the ROS Node Handle instead get sent to the DyKnow Node Handle. When a

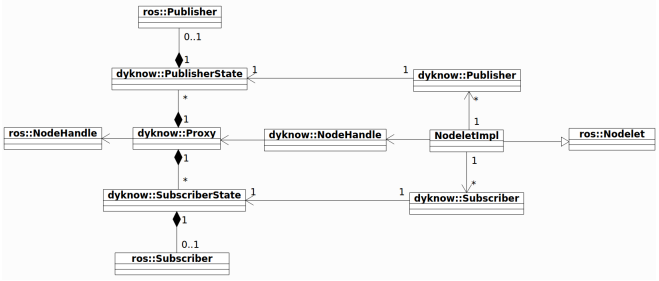

Fig. 2. UML diagram showing the DyKnow nodelet implementation and its relation to standard ROS components, where NodeletImpl indicates a developer-provided stream reasoning implementation.

developer creates subscriptions or publishers, DyKnow provides dyknow::Subscriber and dyknow::Publisher handles. These handles delegate calls to state objects that are maintained by the proxy. Every state object corresponds to a port, and can be assigned a topic by (re)assigning a ROS subscriber or publisher.

Finally, the proxy augments the standard ROS API with additional features that are made available through the DyKnow Node Handle. A DyKnow nodelet is able to set a callback for whenever the nodelet is reconfigured. This can be useful when a reconfiguration requires actions to be taken by a nodelet, for example to notify some part of the system of its new subscriptions and publishers. Further, statistics such as the number of reconfigurations are maintained and made available.

D. Transformation Specifications

Nodelets often require some form of configuration using parameters. For example, a camera nodelet may require a path to a lens model. Since nodelets in DyKnow relinquished control over their publisher and subscribers, the initial configuration and subsequent reconfigurations require knowledge about the ports used by the developer, as well as which topics they should be (re)connected to. Nodelets of the same type can thus differ by configuration.

To deal with these differences, DyKnow makes use of *transformation specifications* to describe different usages of nodelets based on different configurations. The DyKnow manager keeps track of transformation specifications, in addition to instantiated nodelets and their connections. A *transformation* can be seen as a blueprint for a nodelet instance. Its specification contains a reference to the nodelet's source, which is required to instantiate that nodelet. Additionally, a configuration and label are added.

An example of a transformation specification is shown in listing 1. The *label* of a transformation is used by the DyKnow manager to find a particular transformation specification. The *source* refers to the path used by the Nodelet Manager to dynamically load a nodelet. The *parameters* are passed to the nodelet during instantiation. A listing of *ports* is provided so DyKnow can connect them to topics of its choosing. Finally, a series of optional *tags* can be provided to describe ports, which can be useful during configuration

Listing 1. Transformation specification example

```
<transformation type="nodelet">
  <label>undistort(${Cam})</label>
  <source>package/Undistort</source>
  <args>
    <arg label="Cam">{cam1; cam2; cam3}</arg>
  </args>
  <params>
    <param name="config-path" type="string">
      /path/to/configuration/${Cam}/
    </param>
  </params>
  <ports>
    <port type="out">undist</port>
    <port type="in">rawcamera</port>
  </ports>
  <tags>
    <tag port="undist">Undistorted(${Cam})</tag>
    <tag port="raw-camera">RawRGB(${Cam})</tag>
  </tags>
</transformation>
```

planning. These tags can for example make use of Semantic Web concepts to provide a service description.

In some cases, closely-related transformations can be simplified using *transformation templates*. Expanding on the example from listing 1, we may have multiple cameras that use the same software but require different lens models. If we have many such cameras, it would be tedious to write a separate specification for each of them. Instead, we can determine what the set of cameras is, e.g. provided by a developer or acquired from an ontology. The provided argument Cam refers to a set of substitution terms. In the example the set is made explicit, but a reference to an ontological concept is also possible. In such a situation a URI is provided, and a list of individuals is obtained for which the label can be used to create a set of substitution terms. The substitions are done in-place for occurrences of ${Cam}, yielding transformations with labels undistort(cam1), undistort(cam2), and undistort(cam3).

IV. SERVICE OVERVIEW

Nodes can make use of advertised services to interact with each other. DyKnow expands upon the existing services for nodelets in order to provide more control over system configurations. In this section, we look at the pre-existing services ROS provides, and the new services DyKnow adds.

A. Pre-Existing Services for Nodelets

The Nodelet Manager is the sole provider of nodelet services in ROS and offers three services:

- **NodeletLoad:** Given a nodelet name and type, the Nodelet Manager instantiates a nodelet of that type, where the type is a reference to the nodelet's source.
- **NodeletUnload:** Given a nodelet name, the Nodelet Manager destroys that nodelet. A nodelet cannot unload itself.
- **NodeletList:** Returns an array of nodelet names.

B. DyKnow Proxy

The proxy adds additional services to control the connections between topics and ports. It can also list for a given nodelet what topics are connected to which ports at the time of the service call.

- **GetConfig:** Returns a list of ports and associated topics for the nodelet the proxy is associated with.
- **SetConfig:** Takes a list of ports and topics to be connected for the nodelet the proxy is associated with.
- **GetStatistics:** Returns nodelet statistics in terms of uptime, the number of reconfigurations performed, and the number of messages sent or received for each port.

C. DyKnow Manager

The DyKnow manager provides services for the management of transformation specifications and nodelets. Additionally, it keeps a model of the computation graph.

- **AddTransformation:** Given a transformation specification, store the specification under the associated label. If a template is sent, the manager adds a transformation specification for every substitution. Specifications can be overridden.
- **RemoveTransformation:** Given a label, remove the transformation specification with that label, if any.
- **Spawn:** Given a transformation label and name, instantiate a nodelet of that transformation type with the supplied name. Nodelets can be protected from unloading. Uses NodeletLoad.
- **Destroy:** Given a name, destroy the nodelet with that name if it exists and if it is not protected. An unprotected nodelet can destroy itself this way. Uses NodeletUnload.
- **GetModel:** Returns a listing of all running DyKnow nodelets and their port-topic connections. Also returns all stored transformation specifications.

V. CASE STUDIES

We focus on two case studies in which DyKnow's dynamic reconfiguration support has proven to be useful. The first case study deals with the creation of interactive visualization tools, which facilitate human interaction. The second case study is focused on autonomous systems, and deals with a configuration planner for semantic reconfiguration.

A. Interactive Visualization Tools

ROS provides a wide array of visualization tools using a Qt-based framework. For the visualization of nodes and topics, rqt_graph provides a graphical user interface that communicates with the ROS master and produces a DOT graph. While this approach works great for nodes, it fails to detect nodelets as they are threads within the Nodelet Manager node. We therefore forked rqt_graph and replaced the communication with the ROS master to instead query the DyKnow manager for its computation graph. Since ROS does not allow for dynamic reconfiguration, we also had to

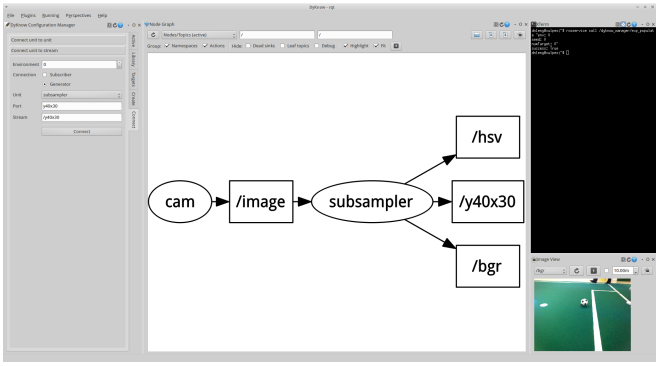

Fig. 3. Screenshot of the interactive visualization tool.

switch from the manual refresh in rqt_graph to a frequency-based refresh. This was combined with a control widget to allow a user to interact with the DyKnow manager.

A screenshot of the tool at work is shown in Figure 3, where the bottom left camera view was produced by the rqt_image_view widget. The graph shown in the center panel was created using the control panel on the left. Ovals in the center graph correspond to nodelets, and rectangles correspond to topics. The bottom-right image view shows the color video stream from a NAO camera after having been processed by a the subsampler. Since no changes were made to the rqt_graph interface itself, this representation is natural to ROS developers.

The control panel on the left supports a number of features based on the services provided by DyKnow. The *active* tab lists the currently active nodelets together with their associated transformations, the number of input ports, and the number of output ports. A *library* tab offers a listing of transformation specifications by label, and allows a user to import or delete transformations. Nodelets can be instantiated through the *create* panel; the user provides a name for the nodelet to be created and a type in terms of transformation specifications. The panel shown is the *connect* panel, where either a combination of two nodelets and ports are selected to be connected with a topic decided by the tool, or a single port and topic can be connected where the user gets to specify the topic name manually.

The visualization tool makes use of all of the DyKnow Manager services, offering an interface that can be managed by human operators: The tool acquires the known computation graph from the DyKnow manager to draw the current configuration, and exposes the services for adding and removing nodelets and transformations through the control panel. It serves as a proof of concept for reconfigurability while providing a useful visualization tool for developer introspection.

B. Semantic Reconfiguration

One of DyKnow's key features pertains to automated reconfiguration based on semantic descriptions. Transformation specifications have room for tags (or metadata) for ports associated with that transformation. We can use these tags to semantically describe transformations by the types of information they require and produce. Using this information, DyKnow can appropriately connect outputs to inputs. Given a tag or semantic description, it is then possible to automatically generate and maintain a computation graph that produces this information. This provides DyKnow with the capability to reason about which streams to subscribe to. The architecture and services presented in this paper make it possible for DyKnow to also perform reconfigurations necessary to maintain a desired stream.

As an example, consider the scenario of a DyKnow-supported NAO robot on a soccer field. We are interested in the position of the ball on the field, which the NAO robot is able to obtain. The DyKnow Manager knows of the following transformations:

- A *bottom camera*, which produces a YUV video stream;
- An *image subsampler*, which takes a YUV video stream and can produce BGR and HSV image streams, as well as downsampled YUV channels;
- A *ball detector*, which takes different resolutions of YUV channels and the NAO's joint states to detect ball objects in its relative frame of reference;
- A *localization* node that estimates the NAO's position in the field's coordinate system, and transforms ball detections to that coordinate system.

The localization node can be described in terms of taking world objects in the relative coordinate system of a NAO and converting these to the field coordinate system. The ball detector produces relative ball objects, which are a kind of world objects. It takes a subsampled video streams from the NAO's own cameras. The image subsampler produces those streams from the raw YUV streams from the NAO's cameras, one of which is provided by the bottom camera. The DyKnow configuration planner can then determine that it needs to produce a computation graph in the above order. If the bottom camera already has a running nodelet, the DyKnow configuration planner will know this from the DyKnow manager and simply note the relevant outgoing topics. It can then set up the remaining nodes and connect them, before returning the name of the resulting topic coming from the localization node. When a battery recharge is needed, the NAO can find the ceiling camera as a suitable alternative to its own camera based on the tags.

VI. PERFORMANCE EVALUATION

The proxy introduced by DyKnow potentially introduces an overhead in throughput. Measuring the overhead gives insights into the cost of adopting DyKnow.

A. Experiment Setup

Topic-based communication between nodelets is assumed to be faster than between nodes because nodelets are part of the same process and nodes are not. In this experiment, we use both as benchmarks for comparison. The computation graph is a linear sequence of connected node(let)s such that each intermediate node(let) receives from a predecessor node(let) and immediately publishes to a successor

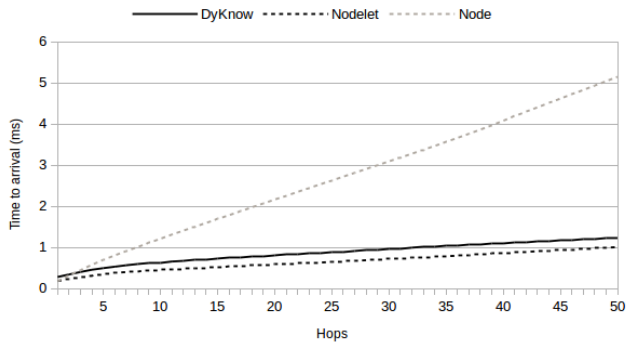

Fig. 4. Performance graph showing the different time-to-arrivals for messages relative to the number of hops for a linear chain.

node(let). The source produces messages containing current time-stamps at a fixed frequency f. Every (intermediate) receiver checks that time-stamp against the arrival time and reports the time difference. The number of node(let)s n then corresponds to the number of message hops.

B. Results

The performance results are shown in Figure 4, where the performance graph contrasts the number of hops to the average time-to-arrival for messages sent along the node(let) chain. The source produced 1,000 time-stamped messages at a frequency of $f = 5$Hz, which every receiver compared to the local time upon arrival prior to forwarding the message. The graph illustrates the time results for DyKnow nodelets and ROS nodelets, as well as ROS nodes. As expected, nodes are much slower than nodelets because they have to communicate between processes. The results for nodes put into perspective the overhead we can see for DyKnow nodelets when compared against ROS nodelets, which grows slowly to about 0.2ms after $n = 50$ hops. We therefore conclude that the overhead induced by DyKnow is negligible.

VII. CONCLUSIONS

The ability for an autonomous system to reconfigure itself during run-time is becoming increasingly important as the variety and availability of computational resources increases. This paper focuses on the popular ROS middleware, which is often used for such systems. Unfortunately, ROS does not support the kind of reconfiguration needed by these application. We presented DyKnow, a dynamically reconfigurable stream reasoning framework that augments the ROS services to support run-time reconfiguration of nodelets. To support reconfiguration, DyKnow needs to get more information about nodelets. Compact transformation specifications were introduced to provide this information. DyKnow also follows the ROS API, which makes converting nodelets to work with DyKnow an easier task. The usefulness of the extended configuration control was shown in the two presented case studies. Finally, a performance overview measuring the overhead cost of using DyKnow was provided. The case studies highlight the usability and ease of adoption of DyKnow, and the performance results show a minimal overhead when comparing DyKnow nodelets to standard ROS nodelets; the stated DyKnow requirements have thus been met.

The results are however not perfect; ROS was not designed to support this kind of reconfigurability. Despite this, a pure extension was sufficient to integrate these capabilities. ROS integration opens up these capabilities to a large audience and potential number of robot platforms.

The reconfiguration support in DyKnow is a critical tool to tackle the bigger problem of reasoning about the availability and relevance of computational resources in order to fulfil a particular information acquisition task. Future work focuses on using the reconfiguration capabilities to this end.

ACKNOWLEDGMENT

This work is partially supported by grants from the National Graduate School in Computer Science, Sweden (CUGS), the Swedish Aeronautics Research Council (NFFP6), the Swedish Foundation for Strategic Research (SSF) project CUAS, the Swedish Research Council (VR) Linnaeus Center CADICS, and the ELLIIT Excellence Center at Linköping-Lund for Information Technology.

REFERENCES

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *Proc. ICRA*, 2009.

[2] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[3] F. Heintz and P. Doherty, "DyKnow: An approach to middleware for knowledge processing," *Journal of Intelligent and Fuzzy Systems*, vol. 15, no. 1, 2004.

[4] F. Heintz, "DyKnow : A stream-based knowledge processing middleware framework," Ph.D. dissertation, Linköping University, 2009.

[5] F. Heintz and Z. Dragisic, "Semantic information integration for stream reasoning," in *Proc. FUSION*, 2012.

[6] D. de Leng and F. Heintz, "Ontology-based introspection in support of stream reasoning," in *Proc. SCAI*, 2015, pp. 78–87.

[7] D. Martin *et al.*, "OWL-S: Semantic markup for web services," *W3C member submission*, 2004.

[8] J.-P. Calbimonte, O. Corcho, and A. J. Gray, "Enabling ontology-based access to streaming data sources," in *Proc. ISWC*. Springer, 2010, pp. 96–111.

[9] M. Compton *et al.*, "The SSN ontology of the W3C semantic sensor network incubator group," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 25–32, 2012.

[10] A. Bröring, K. Janowicz, C. Stasch, and W. Kuhn, *Semantic Challenges for Sensor Plug and Play*, 2009, pp. 72–86.

[11] A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski, "Semantically-enabled sensor plug & play for the sensor web," *Sensors*, vol. 11, no. 8, pp. 7568–7605, 2011.

[12] J. M. T. Portocarrero, F. C. Delicato, P. F. Pires, T. C. Rodrigues, and T. V. Batista, "SAMSON: Self-adaptive middleware for wireless sensor networks," in *Proc. SAC*, 2016, pp. 1315–1322.

[13] F. Kerasiotis, C. Koulamas, C. Antonopoulos, and G. Papadopoulos, "Middleware approaches for wireless sensor networks based on current trends," in *Proc. MECO*, 2015, pp. 244–249.

[14] F. Tang and L. Parker, "Asymtre: Automated synthesis of multi-robot task solutions through software reconfiguration," in *Robotics and Automation*. IEEE, 2005, pp. 1501–1508.

[15] R. Lundh, L. Karlsson, and A. Saffiotti, "Autonomous functional configuration of a network robot system," *Robotics and Autonomous Systems*, vol. 56, no. 10, pp. 819–830, 2008.

[16] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. Seo, and Y.-J. Cho, "The PEIS-ecology project: vision and results," in *Proc. IROS*. IEEE, 2008.