Flexible Communication in Multi-robotic Control System Using HEAD: Hybrid Event-driven Architecture on D-Bus ⋆

Md Omar Faruque Sarker and Torbjørn S. Dahl.

Robotic Intelligence Lab, University of Wales, Newport
Allt-yr-yn Campus, Allt-yr-yn Avenue, Newport, NP205DA, UK
(e-mail: Mdomarfaruque.Sarker| Torbjorn.Dahl@newport.ac.uk)

Abstract: Direct real-time communication among various software components of a multi-robot system (MRS) is much more complicated than that in software simulations. Existing inter-process communication (IPC) mechanisms, such as pipes, shared memory etc., are very rigid and usually enforce tight coupling among software components. Thus they do not integrate well with heterogeneous multi-robot control applications of a relatively larger MRS that typically consists of tens of robots and various sensing and monitoring elements interconnected through several host PCs. In this paper, we present a modular, flexible and decentralized multi-robot control architecture, namely hybrid event-driven architecture on D-Bus (HEAD), that overcomes these issues by decoupling IPC through D-Bus. D-Bus is a relatively new IPC technology for modern Linux desktop environments. It typically uses a message bus daemon that facilitates asynchronous data sharing among multiple processes. Here, we show that by using only a single type of message, namely D-Bus *signal* type, HEAD can efficiently enable real-time interactions among heterogeneous multi-robot control applications. The design of HEAD is flexible enough to add various types of existing and new software components with minimum programming effort. As an example, we present how we achieve a decentralized peer-to-peer communication behaviours among robot controller clients by simply adding only a few lines of new code leaving the major IPC implementation intact. This paper also reports the performance of D-Bus, under both constant and variable IPC load, obtained from our MRS implementation of a manufacturing shop-floor scenario with 16 e-puck robots.

Keywords: multi-robot system, real-time control, inter-process communication, D-Bus

1. INTRODUCTION

Inter-process communications (IPC) among various desktop software components enable them to talk to each other and exchange data, messages or request of services. Technological advancements in computer and communication systems now allow robotic researchers to set-up and conduct experiments on multi-robot systems (MRS) from desktop PCs. Many compelling reasons, including open licensing model, availability of open-source tools for almost free of cost, community support etc., make Linux an ideal operating system for MRS research. However, the integration of heterogeneous software components in Linux desktop becomes a challenging issue, particularly when each robot-control software needs sensory and other data input from various other software components e.g. pose data from a pose-tracker, task information from a task-server etc.

Traditional IPC solutions in a standard Linux desktop, e.g. pipes, sockets, X atoms, shared memory, temporary files etc. (hereafter called *traditional IPCs*), are too static and rigid to meet the demand of a dynamic software system (Wittenburg, 2005). On the other hand, complex and heavy IPC like CORBA fails to integrate into development tool-chains efficiently. They also require a steep learning curve due to their complex implementations. Besides, the failure of Desktop Communication Protocol (DCOP) in system-wide integration and interoperability issues encouraged the development of the D-Bus message bus system, D-Bus for short (Pennington et al., 2010). This message bus system provides simple mechanisms for applications to talk to one another. In this paper, we have described how we have exploited the simplicity and power of D-Bus to design our *hybrid event-driven architecture on D-Bus (HEAD)* for a large MRS.

Our work has been undertaken as a part of the collaborative EPSRC (UK) funded project, "Defying the Rules: How Self-Regulatory Social Systems Work", where we have studied the behaviour of ants, humans and robots and have developed the attractive field model (AFM), a common formal model of division of labour in social systems (Arcaute et al., 2008). In our part, we have aimed to validate AFM by exploring different communication models in a large MRS with 16-40 e-puck [1] robots. In almost all existing MRS researches, including swarm robotics, IPC among a large group of robots (≥ 10) are rare. Most of the robotic

⋆ This research has been funded by the Engineering and Physical Sciences Research Council (EPSRC), UK, grant reference EP/E061915/1.

[1] www.e-puck.org

researchers either use a simulation platform or a small group of robots with no or limited direct communication (e.g. Labella, 2007). However, in our research, we have emphasized the use of real robotic systems so that we can gain practical insight about the limitation and state-of-the art in multi-robot communication techniques.

Traditional IPCs lack the important requirements of IPC among several heterogeneous software components of a large MRS. Firstly, real-time support in IPC is crucial for connecting time-critical control applications. For example, a multi-robot tracking system (MRTS) can share robot pose information with a robot-controller client (RCC) though shared memory (SHM). This pose information can be used to help a robot to navigate in real-time. However if MRTS crashes and stops writing new pose information into the SHM, RCC has no default mechanism to know that SHM data is outdated. Some form of reference counting mechanism can be used to overcome this issue, but that makes the implementation of RCC complicated and error-prone.

Secondly, IPC must be scalable so that adding more software components (thus more robots, sensors, etc.) in the information sharing game does not affect the overall system performance. But clearly this can not be achieved through traditional IPCs, e.g. SHM or temporary files, as the access to computer memory and disk space is costly and time consuming. Thirdly, IPC should be flexible and compatible enough to allow existing software components to join with newly developed components in the information sharing process without much difficulties. Again traditional IPCs are too static and rigid to be integrated with multiple software components. Besides, incompatibility often arises among different applications written in different programming languages with different IPC semantics. Fourthly, IPC should be robust, fault-tolerant and loosely coupled so that if one ceases to work others can still continue to work without strange runtime exceptions. Finally, IPC should be implemented simply and efficiently in any modern high level programming languages, e.g., C/C++, Java, Python. Practically, this is very important since IPC will be required in many places of code and application programmers have little time to look inside the detail implementation of their IPC technology.

Here we present a scalable and distributed multi-robot control architecture built upon D-Bus IPC that works asynchronously in real-time. Our observation says that D-Bus gets an insignificant IPC performance penalty, even if we add tens of new software components. In this paper, we have presented that, by using only the D-Bus signal type message, SwisTrack (Lochmatter et al., 2008), an open-source MRTS, can be integrated with our multi-robot control framework. All software components are loosely coupled and unlike traditional IPCs, one does not depend on another for setting up and shutting down IPC infrastructure. For example, in case of SHM, one software component explicitly needs to set-up and clean-up SHM spaces. In case of D-Bus, any software component can join and leave in the information sharing process at any time. Each component implements its own fall-back strategy if desired information from another component is unavailable at a particular moment. Based on a thin C API, D-Bus also provides many binding in common programming languages. In this work, we use *dbus-python*, a Python binding for D-Bus, that provide us a very clean and efficient IPC implementation.

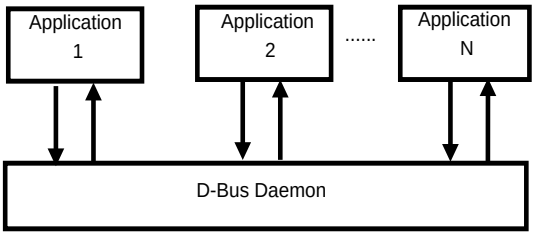


Fig. 1. A typical view of D-Bus message bus system.

Rest of this paper is organized as follows. Section 2 presents an overview of D-Bus IPC technology. Section 3 describes our the design of our multi-robot control architecture, HEAD, with its D-Bus interfaces and flexible communication schemes. Section 4 reports an implementation of HEAD. It also includes our experimental results of D-Bus communication performance. Section 5 draws conclusions.

2. D-BUS COMMUNICATION MODEL

2.1 D-Bus Overview

D-Bus was designed from scratch to replace CORBA and DCOP to fulfil the needs of a modern Linux system. D-Bus can perform basic application IPC as well as it can facilitate sending events, or signals, through the system, allowing different components in the system to communicate. D-BUS is unique from other IPCs in several ways: e.g. 1) the basic unit of IPC in D-BUS is a message, not a byte stream, 2) D-BUS is bus-based and 3) It has separate system-wide and user or session-wide bus (Love, 2005). The simplest form of D-Bus communication is process to process. However, it provides a daemon, known as the *message bus daemon*, that routes messages among processes on a specific bus. In this fashion, a bus topology is formed (Fig. 1). Applications can send to or listen for various events on the bus.

D-Bus specification (Pennington et al., 2010) provides full details of D-Bus message protocols, message and data types, implementation guidelines etc. Here a few basic D-Bus terminologies have been introduced.

D-Bus Connection: *DBusConnection* is the structure that a program first uses to initiate talking to the D-Bus daemon. Programs can either use DBUS_BUS_SYSTEM or DBUS_BUS_SESSION to talk to the system-bus or session-bus daemon respectively.

DBus Message: It is simply a message between two process. All the DBus intercommunication are done using *DBusMessage*. This message can have one of the following four types: *method call*, *method return*, *signal*, and *error*. The DBusMessage structure can carry data payload, by appending boolean integers, real numbers, string etc. to the message body.

D-Bus Interface: This is the interface on which a given Object is available to talk with, e.g. org.freedesktop.DBus.

D-Bus Path: This is the path of a remote *Object* (capitalized to avoid ambiguity) of a target process, e.g. /org/freedesktop/DBus.

D-Bus Signal: This is a type of DBus message to make a signal emission. Signal messages must have three header

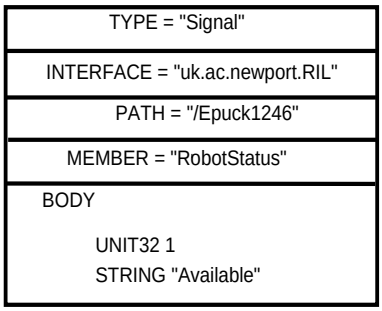


Fig. 2. A typical structure of a D-Bus signal message.

fields: D-Bus path and interface and *memeber* (exact signal name) giving the fully-qualified name of the signal. Fig. 2 shows the design of our robot-status signal that can be emitted over a specified interface and path with a data payload of an integer and a string containing the current status of a robot.

2.2 Strategies for Application Integration

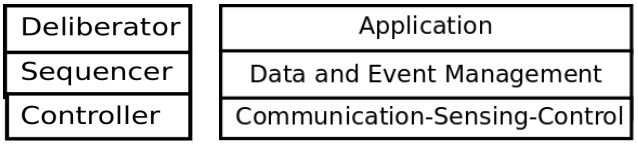
Using D-Bus IPC, an application can use one of the two basic mechanisms for interaction with other applications: 1) by calling a remote Object of a target application and 2) by emitting a signal for one or more target applications. To perform a method call on a D-Bus Object, a method call message must be sent to that Object. It will do some processing and return either a method return message or an error message. D-Bus signals are different as they cannot return anything. Under D-Bus IPC, everything can be done asynchronously without the need of polling. D-Bus provides several language bindings for integrating D-Bus to any native application. The core D-BUS API, written in C, is rather low-level and large. Bindings integrate D-Bus IPC with programming languages and environments, e.g. Glib, Python, Qt and Mono. The preferred use of D-BUS is definitely using language and environment-specific binding which offer ease of use and improved functionality (Love, 2005).

3. HYBRID EVENT-DRIVEN ARCHITECTURE ON D-BUS (HEAD)

3.1 Overview of Multi-robot Control

Controlling a robot in a well-organized manner involves following a control architecture or a set of guiding principles and constraints. A control architecture organizes the structure of a robot's control software by defining the way in which sensing, reasoning and actions are represented, organized and interconnected (Bekey, 2005). Since last few decades, robot control architectures have been evolving from deliberative to reactive and hybrid (combination of deliberative and reactive), behaviour-based and to some other forms. Hybrid control often brings together the best aspects of both reactive and deliberative control by combining the real-time low-level device control and high-level deliberative action control. Only reactive (or deliberative) control approach is not enough for enabling robots to do complex tasks in dynamic environments (Gat et al., 1997).

As shown in Fig. 3(a), hybrid control is usually achieved by a three-layer architecture composed of deliberator, sequencer and controller . *Controller* usually works under



(a) Three layers (b) Abstract model of HEAD

Fig. 3. (a) Classical three-layer hybrid robot control architecture after Gat et al. (1997) (b) Our abstract multi-robot control architecture adopted from hybrid architecture.

real-time reactive feedback control loops to do simple tasks by producing primitive robot behaviours, e.g. obstacle avoidance, wall following etc. *Deliberator* performs time-consuming computations, e.g. running exponential searches or processing computer vision algorithms. In order to achieve specific task goals, the middle component, *sequencer*, typically integrates both deliberator and controller maintaining consistent, robust and timely robot behaviours.

We have organized our multi-robot control architecture, HEAD into three layers as shown in Fig. 3(b). Although HEAD has been designed by adopting the principles of hybrid architecture it has many distinct features that are absent or overlooked in a classical hybrid architecture. Firstly, with respect to the controller layer, HEAD broadly views sensing and control as communication with external entities. Communication as sensing is not new, e.g. it has been reported in multi-agent learning (Mataric, 1998). When robots' on-board computing resources are limited communication can effectively make up their required sensing capabilities. On the other hand, low-level device control is also a series of communication act where actuator commands are typically transmitted over a radio or physical link. Thus, all external communication takes place at the *communication layer*. Components sitting in this layer either act as sensors that can receive environmental state, task information, self pose data etc. via suitable communication link or do the real-time control of devices by sending actuator commands over a target communication channel.

Secondly, under HEAD, the apparent tight coupling with sensors to actuators has been reduced by introducing a *data and event management (DEM) layer*. This layer acts as a short-term storage of sensed data and various events posted by both controller and deliberator components. Task sequencing has been simplified by automated event triggering mechanism. Components of DEM layer simply creates new event channels and components of other layers can subscribe to interested event channels for reading or writing events. If one components updates an event, DEM layer component notifies subscribed components about this event. Controller and deliberator components synchronize their tasks based on this event signals. DEM layer efficiently serves newly arrived data to the controller and deliberator components by this event sharing mechanism. Thus neither specialized languages are needed to program a sequencer nor cumbersome if/else checks are present in this layer.

Finally, deliberator layer of HEAD has been described as the *application layer* that runs real-application code based on high-level user algorithms relying upon low-level sensor data and device states. In classic hybrid architecture the

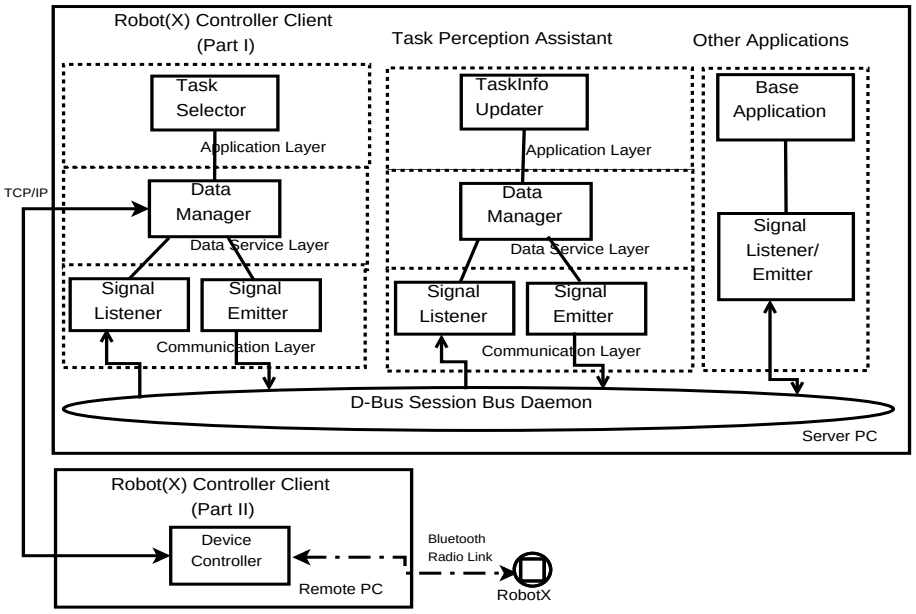


Fig. 4. General outline of *HEAD*. A RCC application has been split into two parts: one runs locally in server PC and another runs remotely, e.g., in embedded PC.

role of this layer has been described mainly in two folds: 1) producing task plan and sending it to the sequencer and 2) answering queries made by the sequencer. Application layer of HEAD follows the former one by generating plan and queuing it to a DEM layer component, but it does not support the latter one. A DEM layer component never makes a query to an application since it acts only as a passive information gateway. Thus this reduced coupling between DEM and application layer has enhanced HEAD with additional robustness and scalability. New applications can be benefited from DEM layer's existing or new event interfaces. Any malfunction or failure in application layer or even in communication layer can be isolated without affecting others.

3.2 D-Bus Signal Interfaces and Software Integrations

D-Bus IPC allows us to completely decouple the interaction of different parts of software components of HEAD. Here we use the term *software component* or *application* to denote the logical groupings of several sub-processes or threads that works under a mother process or main thread (the term *thread* and *process* is used interchangeably). Software components that follows our three-layer architecture for grouping their processes are called *native components* whereas other software applications are called *external components*. In order to integrate both native and external components with HEAD, we have designed two separate communication process: a D-Bus signal reception process, *SignalListener*, and a D-Bus signal emission process, *SignalEmitter*. Inside a native component both of this process can communicate with data and event management process, *DataManager*, by using any suitable mechanisms, such as, multi-threading, multi-processing, TCP etc.

As shown in Fig. 4, a RCC and task-information provider, task perception assistant (TPA) can be treated as the native software components of HEAD. Any external component that intend to act as a sensing (or actuating) element of HEAD needs to implement a SignalEmitter (or SignalListener). For example, we have extended SwisTrack with our D-Bus signal emitting code so that it can emit robot pose messages to individual robots D-Bus path under a common interface. This emitted signal is then caught by SignalListener of individual robot's corresponding RCC. Thus the tight-coupling between SwisTrack and RCC has been removed. During run-time SwisTrack can flexibly track variable number of robots and can broadcast their corresponding pose messages without any re-compilation of code. Moreover, the crashing of SwisTrack or RCC does not affect any other component at run-time.

Expanding SignalEmitter and SignalListener for more D-Bus signals does not require to make any change in our IPC code. D-Bus signal emission process involves the following steps.

Step 1: Connect to a D-Bus daemon.
Step 2: Optionally reserve a D-Bus path or service name (this is not required if the same path is not used by any other process).
Step 3: Send signal to a specified path.

In order to add more signals we just need to repeat step 3 as many times as we need. On the other hand, signal listening can be done by setting up a suitable event loop under any supported language bindings.

3.3 Flexible Communication Schemes

HEAD provides flexibility in designing IPC among multiple components. By using a proper combination of D-Bus signal interface and path, different communication patterns can be achieved. For example, one component can broadcast a signal S in an interface A and a path P and interested components can listen to this signal by matching to that particular interface, path and signal. On the other hand for peer-to-peer (P2P) decentralized communication, we give each application its own interface and different interacting nodes a separate path. For example, if a MRTS intend to send individual robot pose to different RCCs,

it can do that by using path P1, P2, etc. under same S and A. Finally in case of each RCC intending to send signals to its neighbour RCCs, we design our D-Bus signal emitter in a such a way that each RCC can listen signals coming to its own D-Bus path, but can emit signal to other possible D-Bus paths of the RCC group. Thus, using this types of strategies, we can avoid bouncing a signal back to the application itself. It also simplifies the overall testing and debugging process of D-Bus-based applications in real-time.

4. IMPLEMENTATION OF A MULTI-ROBOT MANUFACTURING SCENARIO

4.1 Multi-robot Task Allocation (MRTA)

We have performed a series of MRTA experiments in a manufacturing shop-floor scenario where N number of mobile robots are required to attend to M number of shop tasks spread over a fixed area A. Let these tasks be represented by a set of small rectangular boxes resembling to manufacturing machines. In order to complete a shop task j, a robot R_i needs to reach within a fixed boundary D_j . Each task j has an associated task-urgency ϕ_j that indicates its relative importance over time. If a robot attends to a task j in x^{th} time-step, value of ϕ_j will decreases by a small amount $\delta_{\phi 1}$ in $(x+1)^{th}$ time-step. On the other hand, if a task has not been served by any robot in x^{th} time-step, ϕ_j will increase by another small amount, $\delta_{\phi 2}$, in $(x+1)^{th}$ time-step. According to AFM, all robots will establish attractive fields to all tasks due to the presence of a system-wide continuous flow of information. The strength of these attractive fields, called *stimulus*, will vary according to the distances between robots and tasks, task-urgencies and corresponding sensitizations of robots etc. The detail mathematical model of each robot's task selection mechanism based on AFM and its robotic implementation can be found in Arcaute et al. (2008) and Sarker and Dahl (2010) respectively.

4.2 Experiments

AFM can be implemented as a complete distributed task-allocation system where each robot's RCC can independently select its own task using their own on-board hardware. However, we are particularly interested to evaluate different communication strategies for MRTA without dealing with the large amount of complexities involved in a real robotic set-up. Thus, instead of distributing RCCs among multiple host PCs, we have conducted our experiments using a single host-PC. The limitations of our e-puck robot's hardware e.g. lack of on-board CPU, self-localization module, have restricted us to emulate decentralized communication among RCCs within the D-Bus message bus of a single host-PC.
In Sarker and Dahl (2010) we reported a set of experiments that were designed to validate AFM by testing the occurrence of effective MRTA. As shown in Fig. 5 (right), our software system consists of the SwisTrack MRTS, a TPA and several RCCs (only two of them are shown here). They are developed in Python with its state of the art *Multiprocessing* [2] module. This python module satisfies our need to

[2] http://docs.python.org/library/multiprocessing.html

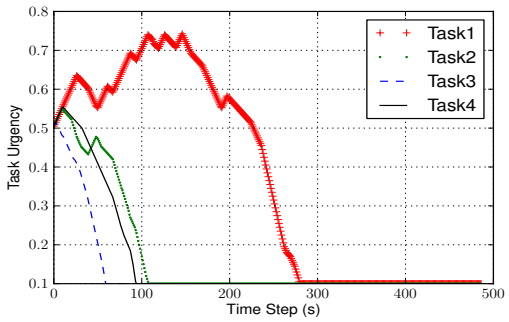


Fig. 6. Dynamic changes in task urgencies.

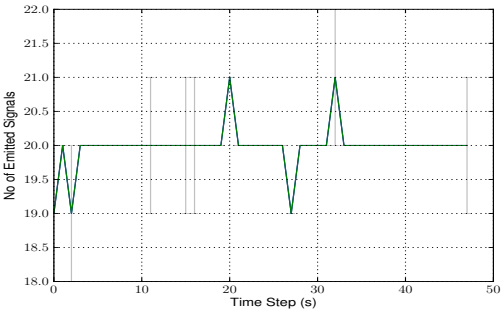


Fig. 7. D-Bus task information signalling frequency of TPA in centralized communication mode.

manage data sharing and synchronization among different sub-processes. As shown in Fig. 5, a RCC consists of five sub-processes: *SignalListener* and *SignalEmitter*, interface with both SwisTrack and TPA. *DataManager* handles data storage and event management issues. *TaskSelector* implements AFM algorithms for task selection. *DeviceController* moves a robot to a target task. Bluetooth radio link is used to establish communication link between a RCC and the corresponding e-puck robot.

4.3 Results

In our experiments, we recorded the D-Bus communication load on our communication system with varying level of task urgencies (see Fig. 6) in both centralized and emulated decentralized communication mode. In centralized communication mode, communication load was almost constant over time (Fig. 7). However in decentralized communication mode, since the emission of robot P2P signals happened asynchronously, the overall communication load on the system varied over time (Fig. 8). In Fig. 9 we can see that, at a specific time in a relatively dense group of robots, a robot was able to listen upto 9 P2P signals within a 50s time-window.

5. CONCLUSION

In this paper, we have presented a scalable hybrid multi-robot control architecture, HEAD, that relies on D-Bus IPC technology. HEAD has several distinct differences from a classical hybrid architecture, e.g. being event-driven that makes sequencing task automated and robust, use of communication as sensing etc. From the architectural

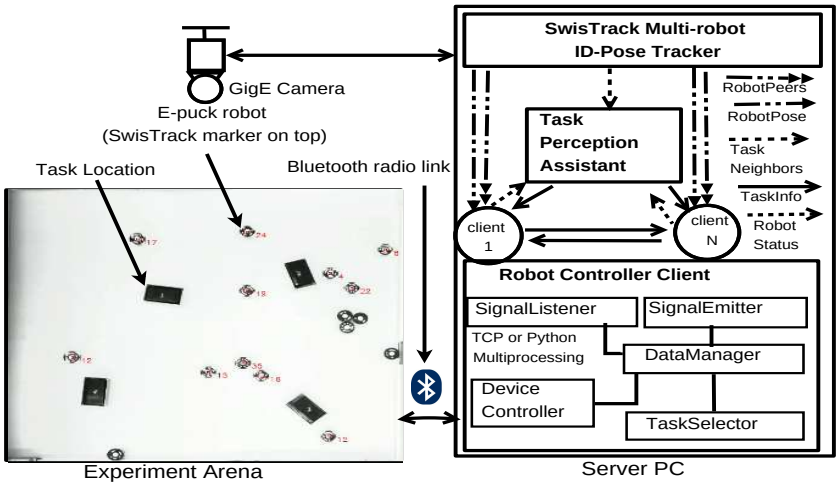


Fig. 5. An example implementation of *HEAD*. Various D-Bus signals are active in an automated loop in decentralized communication mode where components can be distributed among several hosts in the network (not shown here).

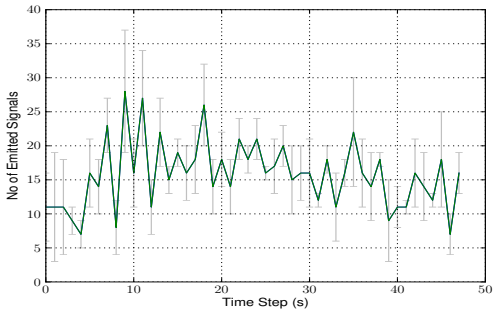


Fig. 8. D-Bus P2P task-info signalling frequency of all robots in decentralized communication mode.

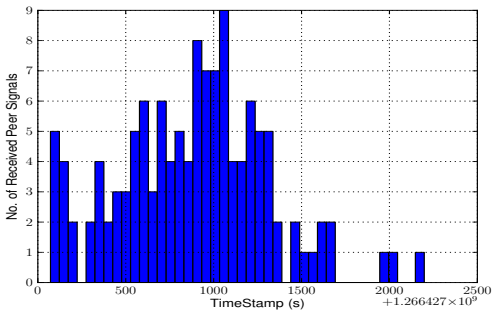


Fig. 9. Robot12's reception of local D-Bus signals.

design and practical implementation in a MRS with 16 e-puck robots, we can say that HEAD is robust and scalable in terms of communication, sensing and control of various elements of a large MRS. We have achieved this, mainly, by decoupling the communications of heterogeneous software components into two separate processes of listening and emitting D-Bus signals.

Although Magnenat and Mondada (2009) used D-Bus for managing control of different micro-controllers of a single robot, to the best of our knowledge, in MRS literature there is no other instance of MRS that uses D-Bus in this fashion to control a large group of robots. This same control strategy can be applied to many other fields, e.g. in multi-agent systems, networked control of different heterogeneous software components. In future, we look forward to deploying HEAD in a MRS having about 40 e-puck robots.

REFERENCES

Arcaute, E., Christensen, K., Sendova-Franks, A., Dahl, T., Espinosa, A., and Jensen, H.J. (2008). Division of labour in ant colonies in terms of attractive fields. In *Ecol. Complex.*

Bekey, G. (2005). *Autonomous robots: from biological inspiration to implementation and control.* The MIT Press.

Gat, E., Bonnasso, R., Murphy, R., and Press, A. (1997). On three-layer architectures. *Artificial intelligence and mobile robots.*

Labella, T. (2007). *Division of labour in groups of robots.* Ph.D. thesis, Universite Libre de Bruxelles.

Lochmatter, T., Roduit, P., Cianci, C., Correll, N., Jacot, J., and Martinoli, A. (2008). Swistrack-a flexible open source tracking software for multi-agent systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008*, 4004–4010.

Love, R. (2005). Get on the d-bus. *Linux Journal*, 2005(130), 3.

Magnenat, S. and Mondada, F. (2009). Aseba Meets D-Bus: From the Depths of a Low-Level Event-Based Architecture. In *IEEE TC-Soft Workshop on Event-based Systems in Robotics (EBS-RO).*

Mataric, M.J. (1998). Using communication to reduce locality in distributed multiagent learning. *Journal of Experimental & Theoretical Artificial Intelligence*, 10, 357–369.

Pennington, H., Carlsson, A., and Larsson, A. (2010). D-bus specification. http://dbus.freedesktop.org/doc/dbus-specification.html.

Sarker, M. and Dahl, T. (2010). A robotic validation of the attractive field model: An inter-disciplinary model of self-regulatory social systems. In *Proc. of Ants 2010: Seventh International Conference on Swarm Intelligence, to appear.*

Wittenburg, G. (2005). Desktop ipc. http://page.mi.fu-berlin.de/wittenbu/studies/desktop_ipc.pdf.