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 as the most preferable 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 (hereafter called robot-controller client or RCC for short) needs sensory and other data input from various other software components (e.g. pose data from a tracker server, 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 heterogeneous to meet the demand of a dynamic software system **?**. On the other hand, complex and heavy IPC like CORBA fails to integrate into a development tool-chain 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 encouraged the development of the D-Bus message bus system (D-Bus for short).

In MRS research context we have found that traditional IPCs fail to support the important requirements of IPC among several heterogeneous software components. Firstly, real-time support in IPC is critical 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 navigating a robot 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 in the information sharing game does not affect the overall system performance. But clearly this

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

can not be achieved through 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 without much difficulties. Again existing IPC mechanisms, e.g. SHM, pipes etc., are too static and rigid to be integrated with existing software components. Besides, incompatibility often arises among different applications written in different programming languages with different semantics of IPC. 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 can be implemented simply and efficiently in any modern high level programming languages, e.g., C/C++, Java, Python etc. 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 any IPC.
Here we present a scalable and distributed multi-robot control architecture built upon D-Bus IPC. D-Bus IPC works asynchronously in real-time. It has virtually no limit how many software components participate in information sharing. In fact, to the best of our knowledge, the performance of D-Bus daemon does not vary if the number of participating software components varies. In this paper we have shown that by using only the signalling interfaces, SwisTrack **?**, an open-source multi-robot tracking tool 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 IPC is unavailable at any time. Based on a thin C API, D-Bus also provides many binding in common programming languages. In this work we use Python-DBus, Python binding for D-Bus, that provide us a very clean and efficient IPC mechanism.

2. D-BUS COMMUNICATION MODEL

2.1 D-Bus Overview

D-BUS was designed from scratch to replace CORBA and DCOP and to fulfil the needs of a modern Linux system. D-BUS can perform basic application IPC, allowing one process to shuttle data to another. D-BUS can facilitate sending events, or signals, through the system, allowing different components in the system to communicate and ultimately to integrate better. D-BUS is unique from other IPC mechanisms 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/session-wide bus **?** . The simplest form of communication is process to process. D-BUS, however, provides a daemon, known as the message bus daemon, that routes messages between processes on a specific bus. In this fashion, a bus topology is formed, allowing processes to speak to one or more applications at the same time. Applications

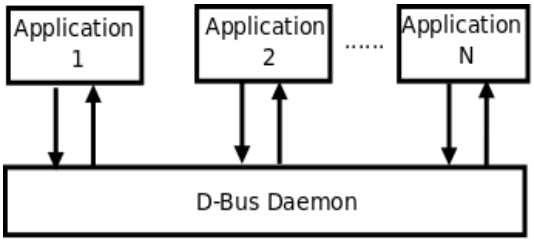

Fig. 1. A typical view of D-Bus message bus system

can send to or listen for various events on the bus.
D-Bus specification [1] provides full details of D-Bus message protocols, message and data types, implementation guidelines and so forth. Here we discuss some relevant part of this specification. As we have already mentioned D-Bus provides a system-wide system bus and user-level session bus. Fig. **??** an example of this bus structure. In this paper we have limited our discussion to the latter one and skipped some advance topics like D-Bus security and so on. Here a few basic D-Bus terminologies have been introduced from D-Bus literature.
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 respective daemons.
DBus Message: It is simply a message between two process. All the DBus intercommunication are done using *DBusMessage*. These messages can have the following four types: method calls, method returns, signals, and errors. The DBusMessage structure can carry data payload, by appending boolean integers, real numbers, string, arrays, etc. to the message body.
D-Bus Path: This is the path of a remote *Object* (hereafter, this is captitalized to avoid ambiguity) of target process, e.g. orgfreedesktopDBus.
D-Bus Interface: This is the interface on a given Object to talk with, e.g. org.freedesktop.DBus.
D-Bus Method Call: This is a type of DBus message that used to invoke a method on a remote Object.
D-Bus Signal: This is a type of DBus message to make a signal emission. As stated in D-Bus specification, Signal messages must have three header fields: PATH giving the object the signal was emitted from, plus INTERFACE and MEMBER giving the fully-qualified name of the signal. The INTERFACE header is required for signals, though it is optional for method calls. The structure of a signal is shown Fig. **??** and it shows the design of robot-status signal that emits over specified interfaces and paths with a data payload of an integer and a string containing robot-status message.
D-Bus Error: This is the structure that holds the error code which occurs by calling a DBus method.

2.2 Strategies for Application Integration

Under D-Bus, there are two basic mechanisms for applications to interact with each other: by calling a remote Object of target application and by emitting a signal for interested 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

[1] http://dbus.freedesktop.org/doc/dbus-specification.html

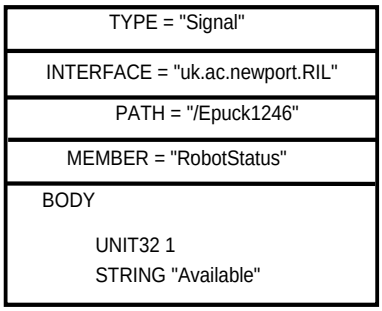

Fig. 2. A typical structure of a D-Bus signal message

a method return message or an error message. Signals are different in that they cannot return anything: there is neither a "signal return" message, nor any other type of error message see this [2] for some example use-cases. Thus on D-Bus 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. On top of this API, bindings integrate with programming languages and environments, including Glib, Python, Qt and Mono. On top of providing language wrappers, the bindings provide environment-specific features. For example, the Glib bindings treat D-BUS connections as GObjects and allow messaging to integrate into the Glib mainloop. The preferred use of D-BUS is definitely using language and environment-specific bindings, both for ease of use and improved functionality **?**.

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. As defined by **?**, a robot control architecture is a set of guiding principles and constraints for organizing a robot's control system. Since last few decades, robot control architectures has been evolving from deliberative to reactive and hybrid (combination of deliberative and reactive), behaviour-based and to some other forms. It has been well established that hybrid control can bring 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 a dynamic environments **?**. As shown in Fig. **??**, this is usually achieved by a three-layer architecture composed of deliberator, sequencer and controller . Controller usually works under 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 search or computer vision processing 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.

[2] http://www.ibm.com/developerworks/linux/library/l-dbus.html

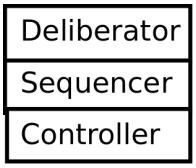

Fig. 3. Classical three layer robot control architecture **?**

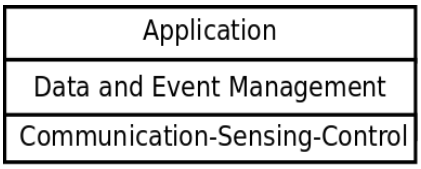

Fig. 4. Our abstract multi-robot control architecture

3.2 Characteristics of HEAD

Although our multi-robot control architecture, HEAD, has been designed by adopting the principles of hybrid architecture it has many distinct features that are absent in a classical hybrid architecture. Firstly, with respect to 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 **?** and many other places. 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, at the bottom layer of HEAD is the communication layer where all external communication takes place over any suitable medium. 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, the apparent tight coupling with sensors to actuators has been reduced by introducing a data and event management (DEM) layer. DEM 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. DEM simply creates new event channels and interested components subscribe to this event for reading or writing. If one components updates an event DEM updates subscribed components about this event. Controller and deliberator components synchronize their tasks based on this event signals. DEM efficiently serves newly arrived data to controller and deliberator components by this event 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 an application layer that runs real-application code based on high-level user algorithms as well as low-level sensor data and device states. In classic hybrid architecture the role of this layer has been described mainly in two folds: 1) producing task plan and sending it to sequencer and 2) answering queries made by sequencer. Application layer of HEAD follows the former one by generating plan

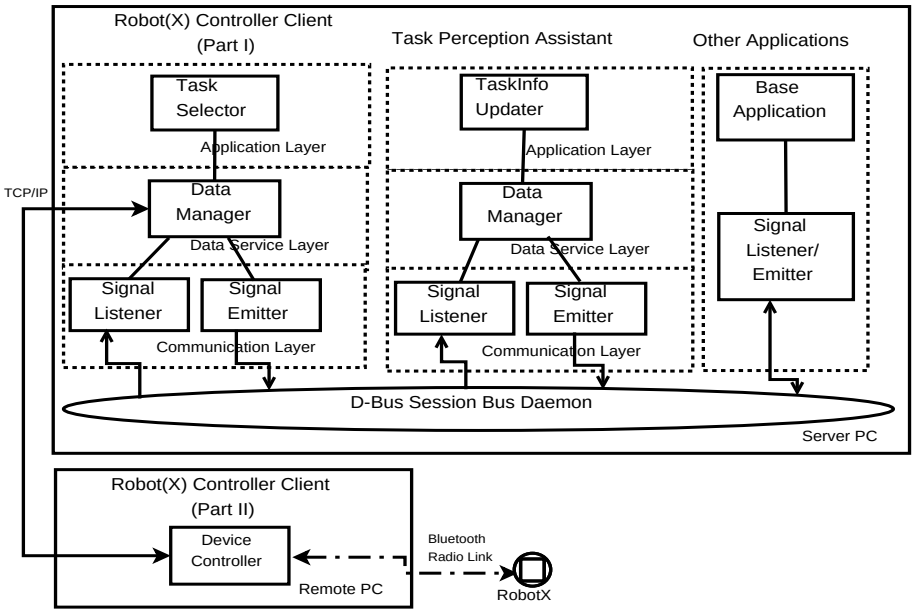

Fig. 5. General outline of *HEAD*. Robot-Controller-Client application has been splitted into two parts: one runs locally in server PC and another runs remotely, e.g., in embedded PC

and queuing it to DEM layer, but it does not support the latter one. DEM layer 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. Additional applications can be added with 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.

3.3 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 (here we use the term thread and process interchangeably). Software components that follows our three-layer architecture for grouping its processes are called native component whereas existing software applications are called external component. As shown in Fig. **??**, RCC and TPA are native software components of HEAD whereas SwisTrack a multi-robot tracking tool **?** used with HEAD is called an external component.

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 (offered in Python multiprocessing), TCP or any other networking protocol. Any external component that intend to act as a sensing (actuating) element of HEAD need to implement a SignalEmitter (SignalListener). For example, we extend SwisTrack with D-Bus signal emitting code (aka SignalEmitter) so that it can emit robot pose messages to individual robots D-Bus path under a common interface (uk.ac.newport.SwisTrack). This emitted signal is then caught by SignalListener of individual robot's RCC. Thus the tight-coupling between SwisTrack and RCC has been removed. During run-time SwisTrack can flexibly track variable number of robots and broadcast their corresponding pose messages without any re-compilation of code. Moreover, in worse cases, if SwisTrack or RCC crashes it 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 the IPC implementation code. Let us first look at how to setup a signal emission process.

Steps for setting up signal emission:

Step 1: Connect to a D-Bus daemon

```
DBusError error;
DBusConnection *conn;
dbus_error_init (&error);
conn = dbus_bus_get (DBUS_BUS_SESSION, &error);
```

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

```
DBusMessage *message;
message = dbus_message_new_signal (
``/target/dbus/path",``target.dbus.interface",
``Config");
/* Send the signal */
dbus_connection_send (connection, message, NULL);
dbus_message_unref (message);
```

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

2

basic implementation of both of these processes in Python language can be found in this tutorial [3].

3.4 Flexible Communication Schemes

HEAD provides flexibility in designing communication among multiple components. By using a proper combination of signal interface and path different communication patterns can be achieved. For example, one component can broadcast signal S in interface A and path P and interested components can listen to this signal by matching to 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 multi-robot tracker 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 signal emitter in a such a way that each RCC can listen signals coming to its own signal path but can emit signal to other possible paths of the group. Thus, this types of strategies make it possible to avoid bouncing a signal to self and to simplify the overall testing and debugging of D-Bus communication of different applications in real-time. Moreover, D-Bus library comes with a handy tool, dbus-monitor, that can be used to listen to messages and debug the behaviours of applications.

4. IMPLEMENTATION OF A MULTI-ROBOT MANUFACTURING SCENARIO

4.1 Multi-robot Task Allocation

We have performed a multi-robot task allocation 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 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 δ_ϕ 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 in $(x+1)^{th}$ time-step. According 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' task selection mechanism based on AFM and its robotic implementation can be found in **??**.

4.2 Experiments

We have developed a system where up to 40 E-puck robots **?** can operate together according to the generic rules of the AFM. In **?**we reported a set of experiments that

[3] http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html

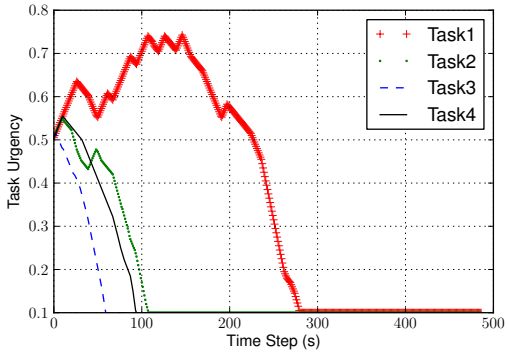

Fig. 7. Dynamic changes in task urgencies

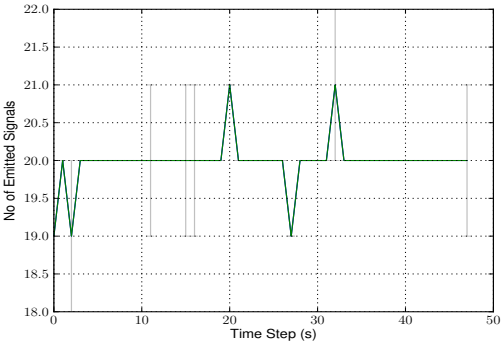

Fig. 8. D-Bus task information signalling frequency of TPA in centralized communication mode

were designed to validate AFM by testing the occurrence of convergent MRTA. As shown in Fig. **??** (right), our software system consists of a multi-robot tracking system, a task perception assistant (TPA) and robot controller clients (RCC).

4.3 Results and Discussions

In our experiments, we measured the communication load on our communication system by logging all D-Bus signals. In centralized communication mode, communication load was almost constant over time. However in decentralized communication mode, since the emission of robot P2P signals happened asynchronously we found that the overall communication load on the system varied over time.

5. CONCLUSION

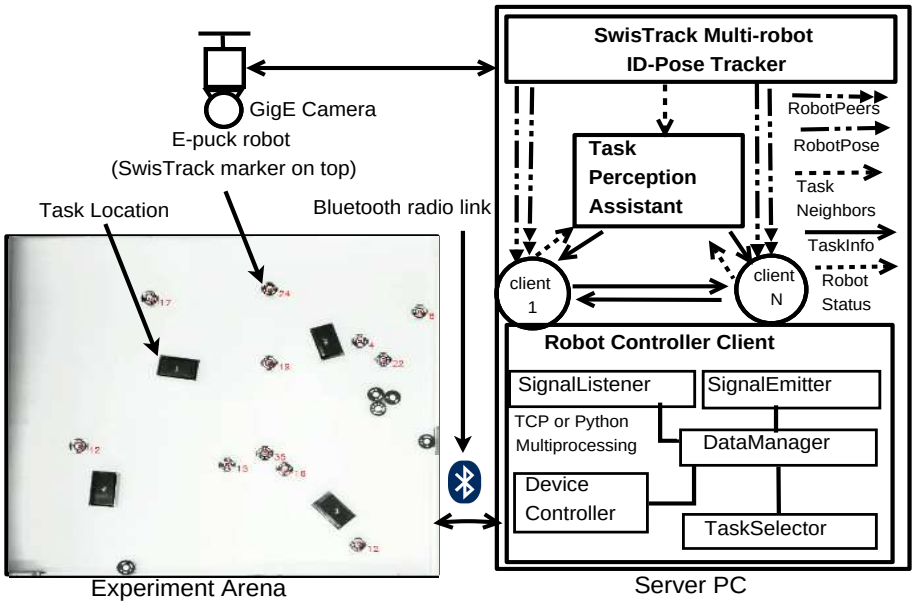

Fig. 6. An example implementation of *HEAD*. Robot-Controller-Clients can talk to each other and exchange task information through D-Bus signals

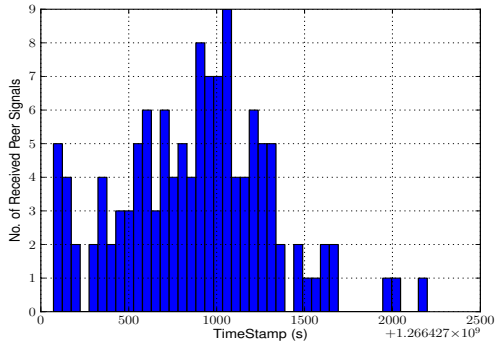

Fig. 9. Robot12's simultaneous reception of task information signals from peers

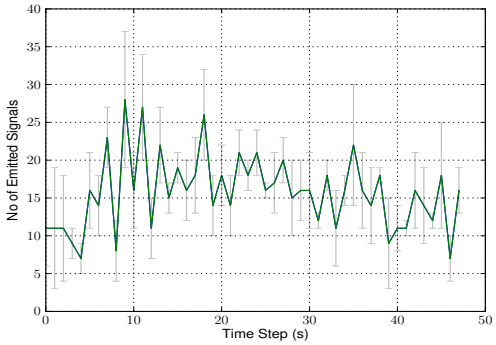

Fig. 10. Local P2P task information signalling frequency of all robots in decentralized communication mode