SDfR protocol : Service Discovery for Robots

Stefan-Gabriel Chitic, Julien Ponge, Olivier Simonin

SDfR protocol : Service Discovery for Robots

Stefan-Gabriel CHITIC
Université de Lyon,
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne, France
stefan.chitic@insa-lyon.fr

Julien PONGE
Université de Lyon,
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne, France
julien.ponge@insa-lyon.fr

Olivier SIMONIN
Université de Lyon,
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne, France
olivier.simonin@insa-lyon.fr

ABSTRACT

Nowadays robotic applications tend towards fleets of robots, being capable of sharing information between multiple hosts as well as performing one or multiple tasks together. In order to facilitate interaction and cooperation robots should advertise about their functionalities as services. In this paper, we propose a protocol for highly dynamic robotic applications that will allow robots to discover their neighbors services and their capabilities in any wireless infrastructure. The protocol is an adaptation of the Service Discovery Protocol (SDP) used in Universal Plug and Play(UPnP). The paper also includes a series of benchmarking across multiple scenarios that allow us to evaluate our protocol. Our experiments looked up the various impacts that our proposal has on a multi-robot system, like request time of a publisher and a subscriber, the quantity of CPU and memory used and quantity of send and received bytes over the network.

Keywords

Multi-robot, Middleware, Robotic cloud, Service Discovery, Network discovery

1. INTRODUCTION

Robots will progressively populate our everyday life, in order to assist humans in many tasks or to replace them in dangerous missions. They are becoming a standard equipment in modern manufacturing sites, providing fast and reliable routine operations. They will be used also in hazardous environments, requiring robots to be more autonomous and able of cooperation with humans. Another important domain is Military missions which will employ robots as unmanned combat vehicles in order to reduce human casualties. Finally, robots are entering our domestic environment as service robots, being used in simple but unwanted jobs, such as vacuum cleaning, floor washing, and lawn mowing.

Nowadays robotic applications tend to use fleets of robots, being capable of sharing information as well as performing one or multiple tasks together. An autonomous robot fleet refers to a dynamic system that is populated by heterogeneous devices, with different hardware and software components capable of sharing data and performing one or several tasks within other members. The fleet can also communicate with mobile or fix connected objects and sensors, cooperating together to achieve a common goal.

All those new applications that are operating in a multi-robot context, are generating multiple layers of complexity into the robotic development. Firstly, a robotic application should manage the complexity of the robot as a unit with all the communication and compatibility issues between its internal components. Secondly, they need to manage the complexity inside the fleet by managing the mobility, communication, information and task sharing among multiple robots. Finally, a third layer of complexity is added, because the robots can be in communication with the environment in which they have been deployed, being part of Internet of Things [2].

Recent work has shown that these layers of complexity can be managed using the appropriate middleware families ([3],[5],[13]). A robotic middleware is a software that acts as a bridge between a network layer or an operating system and applications, being an important component in the process of developing, deploying and operating a robotic application inside a fleet. In our work, we adopt such a middleware based vision.

In this paper we focus on a central need of fleets of robots: how to allow them to be aware of connected neighbors and their services. Combining component and service-oriented programming greatly simplifies the implementation of highly-adaptive, constantly-evolving applications [7]. The robots should advertise their functionalities as services in order to allow other members of the fleet to interact with them. In network based application, service-oriented programming is now a largely accepted principle [9]. The main concepts of service-oriented programming are the publications by the providers and the discovery by the consumers of the robotic services.

In the robotic context, we adapt the well-known SDP protocol in order to allow robots to discover their connected neighbors, their services and their capabilities in any wireless infrastructure. SDP protocol is highly used nowadays in most of the connected devices. To take into account the mobility of robots, we have changed a series of fields in the messages headers as well as added a memory mechanism to limit consumed bandwidth. Our proposal is validated using experimental benchmarks on multiple scenarios with a various number of Turtlebot 2.

The paper is structured as follows: Section 2 discusses the service discovery in the related domains. Section 3 present our proposal

for service discovery in a fleet context. Section 4 proposes an implementation of our protocol. Section 5 evaluates our protocol via a series of benchmarks and Section 6 concludes the paper.

2. RELATED WORK

In the robotic world, the problem of service discovery was approached mostly in centralized network infrastructures by using classical Universal Plug and Play (UPnP) protocol. Since the concept of having the robotic tasks and processes as services is not mature yet, the main focus on research on service discovery in robotics has orientated toward the integration with the environment like smart homes or smart cities. Paper [2] provides a case study of integration of service robots and smart-homes via UPnP. In these cases, the authors are not referring to a robot as part of a specific fleet, but as part of an environment, in which the robot is considered as an entity that can offer services. This point of view is slightly different in case of a robotic fleet [19], where robots are composed of multiple services that need to be discovered by the other members.

Another way to see a fleet of networked robots is like a service-oriented multi agent system that deals with large-scale and highly-dynamic systems that need to provide mechanisms to discover service availabilities. Such environments like Peer-to-Peer (P2P), Multi-Agent Systems(MAS) or Service-Oriented Environments(SOE) tend to approach the problem of service discovery in *centralized*, *distributed* or *decentralized* way. *Centralized* mechanism like super-peers [8], middle-agents [10] or central registries [17] are limited in number of agents in the system and in terms of number of requests. It also uses a centralized node which can have serious impact if the central point becomes unreachable. *Distributed* approaches such as Distributed Hash Tables(DHT) [12] offers more scalability and robustness by having multiple specific nodes that can manage the resources. *Decentralized* systems consider all the nodes equals. This approach provides more flexibility, but it has its downsides, since each node only has partial view of the entire system. As mentioned in [4], an interesting way to discover service inside a decentralized and self-organized multi-agent system is to use the homophily between agents.

Classical protocols and middlewares for service discovery in distributed environments like data-grids, clouds or even smart environments use same type of methods. The base service discovery is having one *centralized* registry that manages service description. It still has the same issues as in the case of multi-agent system but it provides consistency and scalability. The other approach is having a *decentralized* system (e.g UPnP, Jini [14] or SLP [16]) that can be : *a purely distributed* solution where each node stores its own service repositories or a *hybrid* solution that includes super-nodes that aggregate information from other peers.

The solutions presented above have their downsides when applied to ad-hoc multi-robot systems. Firstly, due to the mobility of the robots, the network connection is highly instable and robots can disconnect and reconnect very often. Existing protocols do not perform the same way in a highly dynamic environment and in a static one. As mentioned in [9], the challenge is to set the trade off between physical mobility and scalability. Secondly, existing protocols are not very adaptive. The discovery protocol should be ready to be used at any time and track its usage and failures. Existing protocols like UPnP, have a limited memory factor and when the connection is timed-outed, the discovery process reinitializes itself at reconnection.

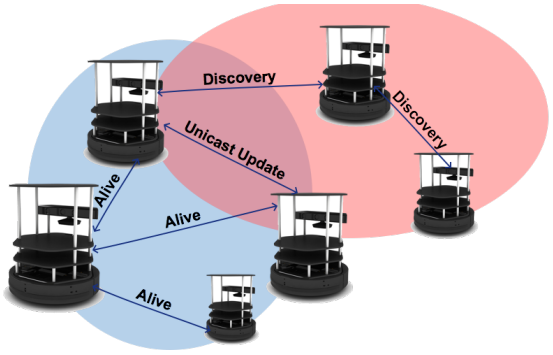

Figure 1: SDfR in ad-hoc network

Next sections present our extension of SDP used in UPnP in order to overcome the limitation of existing service discovery approaches.

3. SERVICE DISCOVERY FOR ROBOTS

Before detailing our approach, we abstract the model. Robot service discovery inside a fleet is based on a publish subscriber process. A process running on each robot looks up its neighbors and responds to new ones while advertising its own presence. Based on this, other processes on each robot can publish their services and access the neighbors services list. Figure 1 illustrates the protocol functioning on an ad-hoc segmented network.

Our protocol, called Service Discovery for Robots (SDfR), is adapted from Simple Service Discovery Protocol (SSDP) that is being used in UPnP. SSDP is used for advertisement and discovery of network services and presence information. It accomplishes this without assistance of server-based configuration mechanisms, being a purely distributed service discovery protocol. Being an evolution of SSDP, SDfR can be also used to provide service discovery with the smart-environment in which the robots are being deployed.

In order to have a suitable service discovery protocol for ad-hoc networked robots, we propose a protocol that is coping with the possibility of network disconnection. For this, we introduce a memory mechanism in which the protocol does not reinitializes itself after reconnection in order to avoid network overload.

This section is structured as follows: Subsection 3.1 discusses the limitations of SSDP and how SDfR overcomes those. Section 3.2 presents the whole protocol model. Section 3.3 continues with the validation of the model and Section 3.4 presents the headers and the messages defined.

3.1 SSDP limitations overcome

SSDP is sending all the transmissions in multicast over UDP in a request-response context. It waits for a request to be acknowledged until a timeout is reached. This way of using multicast transmission has a great impact on the bandwidth consumed by the protocol. Even-more the protocol reinitializes each time a disconnection is made non-gracefully or a timeout parameter has expired. It does not provide any memory factor. Finally, it has limited performance in ad-hoc network. SSDP is most adapted for a centralized infrastructure where mobility is managed within network coverage.

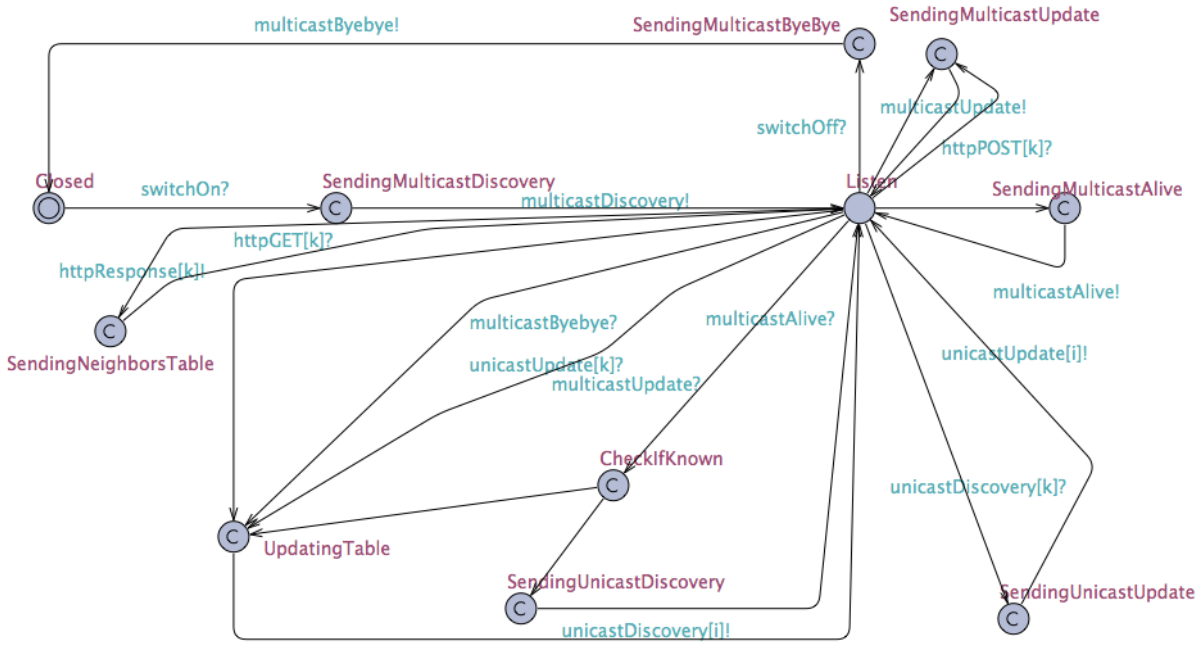

Figure 2: SDfR state automaton

SSDP and SDfR are similar because of:

Multicast transmissions As in SSDP, SDfR is sending most of the internal messages in multicast, avoiding the overhead generated by unicast transmission in order to propagate the same message.

HTTP messages The messages that are being sent are an evolution of HTTP messages.

The main differences between SSDP and SDfR are:

Limited multicast transmissions SDfR does not have a request-response model. In order to avoid failure in case of a disconnection due to the movement of the robots outside the coverage area, all the communications are done using UDP (User Datagram Protocol)

Unicast transmissions To limit the network flooding when the protocol needs information from just one robot, a second transmission is enable in unicast mode.

History map SDfR does not need to reinitialize the entire discovery protocol when the connection is lost, because it disposes of a history map of all the already seen robots and their services. In order to avoid services that are out of reach (e.g service of robots that are present in the history map but are not present in the covered communication area), a connection indicator is computed for each robot.

3.2 Protocol Description

We have designed a model based on timed automaton [1] for SDfR protocol. Figure 2 presents the timed automaton of our protocol. We have chosen to model in this way because SDfR has a finite number of states extended with a finite set of real-value clocks.

SDfR, defines two methods that indicate the desired action to be performed: *M-SEARCH* and *NOTIFY*. The *M-SEARCH* method type is used when a new robot requests the discovery of new fleet members and their services. The only message type associated with this method is *Discovery*. The other method, *NOTIFY* is used to respond to a *Discovery* request or to inform the other about changes in the current state of the robot. The message types associated with this method are: *Update*, *Alive* and *Byebye*.

- The *Update* message is being sent as a response to a *Discovery* request or when the current services or capacities of the robot change.
- The *Alive* message is sent recurrently, as a beacon, in order to inform the others about the presence of the robot.
- The *Byebye* message is being send when the robot stop gracefully, in order to inform the others about its disappearance.

When the protocol initializes a discovery muticast message is sent, and then the protocol changes state into *listen* on a multicast as well as on an unicast socket. When the other robots receive a discovery message, they will respond with an update message.

When the protocol receives an update message, it will pass into an atomic state, *Updating neighbors table*. When the protocol receives an alive message, it will pass into *Check if known* state, that will determine if the unicast IP of the sender is already known. If so, it will pass into *Updating neighbors table*, otherwise it will send a unicast discovery message. The sender of the alive message responds by sending an unicast update message.

If the protocol traps a gracefully shutdown, a byebye message is sent and the other robots will update their neighbor table.

3.3 Protocol validation

We have modeled our system with UPPAAL [11], a well-known formalism for the behavior of systems. All the state change depend on the message exchange timeout. In our protocol, all the actions

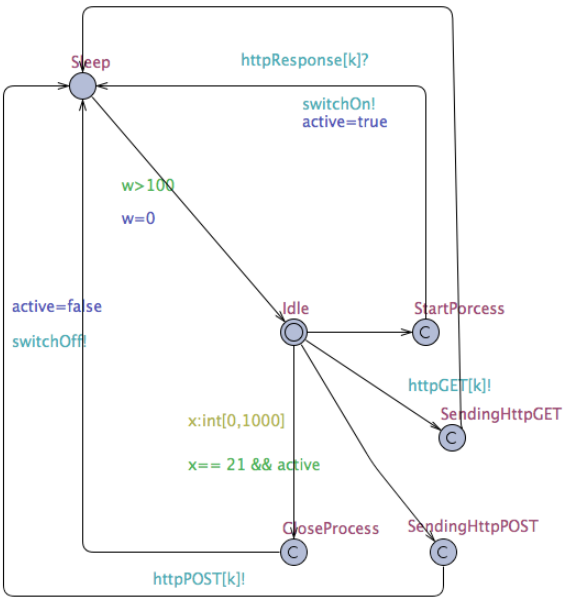

Figure 3: SDfR client automaton

that change the states depend on a specific time interval and on the sequence of the previous states. Thus, the time in our automaton is at the same time quantitative and qualitative. In order to simulate and check our model we have used a composition of the SDfR automaton and a client automaton (see fig. 3) that simulates the behavior of a random publisher or subscriber.

In order to verify our model, we have used linear temporal logic (LTL) [1]. In LTL, for a given formula f, *A[]* means that f holds for all paths and *E<>* means that there exits a path where f hold. We have verified our model for deadlock and reachability properties.

A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. *UPPAAL* provides a special keyword, as shown in eq. (1), for verifying if a composition of automaton has a deadlock.

$$A[]\ not\ deadlock \tag{1}$$

The evaluation succeed, thus protocol has no deadlocks.

Reachability properties are the simplest form of properties. They ask whether a given state formula can be satisfied by any reachable state. In order to prove that our protocol satisfies reachability properties, we have used eq. (2) and eq. (3).

$$\begin{aligned} Clients = \{\forall c | c\ follows\ the\ behavior\ of\ a\ SDfR\ client\} \\ States = \{Closed, SendingDiscovery, \\ Listen, UpdatingTable, \\ SendingAlive, SendingUpdate\} \end{aligned} \tag{2}$$

$$\forall c \in Clients, \forall s \in States, \exists E <> c.s == True \tag{3}$$

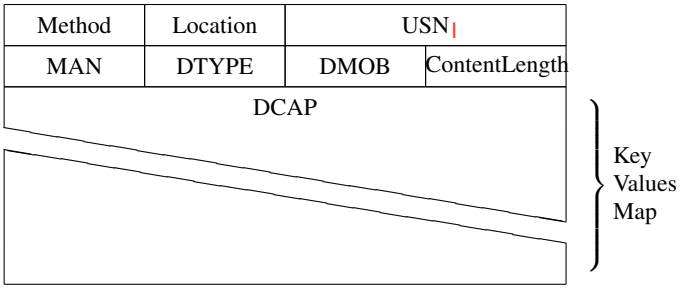

Figure 4: SDfR common header

We have conduced the model checking of our automaton in UPPAAL. The model validation is independent to the number of clients used. After the evaluation, all the states from all clients are reachable.

3.4 Messages and headers

In order to better understand the dynamic of the protocol, the following subsection focuses on the message and their headers and how they differ from UPnP in order to be adapted for multi-robot systems.

SDfR is compatible with UPnP because it uses the structures of SSDP, the service discovery protocol used in UPnP. This gives SDfR the advantage of being interoperable with any smart environment. In fig. 4, the fields inside a SDfR message header are displayed.

A full description of the SDfR header can be find below (fields with a $^+$ are new):

Location - Location of the device. This field was present in UPnP and was kept for retrocompatibility with this protocol.

USN - Unique Service Name. The field was present already in UPnP and reused by SDfR.

MAN - Message Type. The field was present already in UPnP and reused by SDfR.

DTYPE$^+$ - Device Type. This represents the type of hardware platform. (e.g Turtlebot2, PR2, etc).

DMOB$^+$ - Device Mobility. This new field was added in order to characterize the mobility of a robot. (e.g Mobile, Temporary Mobile, Static, etc).

ContentLength - The length of the message content without the header. For transmissions without any payload, the field is set to 0.

DCAP$^+$ - Device Capacities. It represents a dictionary on keys and values that characterizes the state of the robot. It can include statical information like the CPU frequency or the memory capacity, as well as dynamically information like the percentage of battery, the CPU usage ratio, etc. This data was included in every header of the SDfR because the information sent is highly-dynamic.

The main differences between SDfR and SSDP are the addition of 3 new fields: *DTYPE*, *DMOB* and *DCAP*. Another difference is the location field that is always marked by a '*'. In SSDP, the location was used to physically pin point the device like 'kitchen-fridge',

but in a fleet context it is hardly the case to have a fix physical location. Furthermore, the USN from SSDP, which represents the unique name of the service, is changed with the unicast IP address of the robot. All the SDfR messages are being sent in multicast, but the robot needs the unicast address in order to use the information given by the protocol.

All the messages share the same header information, only the payload of the message differs form one message type to another. There are even messages without payload like *Byebye* or *Discovery*. The fields, like in HTTP messages, do not have a fixed size, being very useful to customize the header for future use.

Bellow are two examples of SDfR message. First we present a discovery message.

```
M-SEARCH
Location:*
USN:10.1.124.134
UUID:847568B4-CF41-4530-940C
MAN:ssdp:discovery
DTYPE:Turtlebot2
DCAP:CPU=2.0Ghz|RAM=4Gb|BAT=59%
DMOB:Mobile,
ContentLength:0
```

The following message is the update message send in response to the previous message by another robot.

```
NOTIFY
Location:*
USN:10.1.101.94
UUID:D1FED88A-659B-4EC1-A01E
MAN:ssdp:update
DTYPE:Turtlebot2
DCAP:CPU=2.0Ghz|RAM=4Gb|BAT=98%
DMOB:Mobile,
ContentLength:247

{"Services":
  [
    {
      "Name":"P2P Monitoring",
      "URL":"10.1.101.94:8042/auto_description",
      "Uuid":"3FA2F711-E142-4572-9AF0"
      "Metadata":{
          "status" : "ok",
          "alerts" : "0",
      }
    }
  ]
}
```

4. IMPLEMENTATION

Our solution is coded in the Go programming language [15]. We wanted an easy to build programming language that allows us to develop simple, reliable, and efficient robotic service. Go provides concurrent abstractions and safe memory management, something lacking in C/C++ and to a certain degree from Python. Even more, with 'Go', we can build all-in-one package, that does not have any dependencies since all the auxiliary libraries are built-in in the service executable. Even with all the dependences, the executable is still lite. Furthermore, we are able to build cross-platform executables which is an important aspect in deploying SDfR service across a heterogeneous platform of robots.

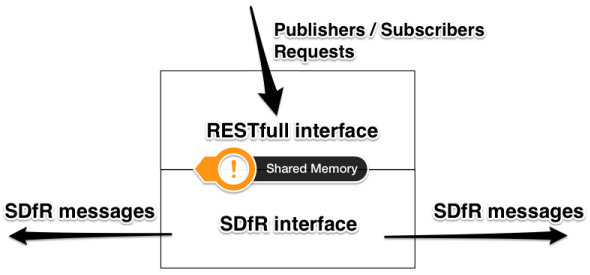

Figure 5: SDfR service architecture

To achieve the service discovery goal, we needed a generic way to describe the services in the Update messages. Our service is not interested in consuming the registered services, its purpose is to distribute the information about these services across the network fleet. A service is described as follows:

Name - The name of the service. Same service on multiple robots may have the same name.

URL - The auto-description URL of the service. This is being used by other services / processes that want to consume the service to configure themselves.

Uuid - A universal unique identifier. This field is unique across all the fleet. Even if a service is found on multiple robots, they will have different Uuids.

Metadata - A key value dictionary that is used to filter and to describe the services.

We have implemented SDfR as a service itself. The main advantage is that it can run separately of the other processes on the robot and all the messages are being consumed by instances of the service on multiple robots. If it fails, it would not affect the other services running on the robot. It is able to recover after crash and restart the process independently. This sand-boxing[1] also ensures that the information sent by the protocol is not corrupted by any other third-parties.

SDfR service is composed of two key components that share the same memory as shown in fig. 5. The external interface used to communicate with other services and components is represented as a RESTfull[2] interface. The second component is the internal communication interface used by SDfR participants in order to advertise their presence and services across the neighbors network.

A *neighbor robot* is defined as follows in SDfR list of services:

IPAddr - The unicast IP address of the neighbor.

DeviceType - The device type of the neighbor (e.g Turltebot2, PR2).

DeviceCapabilities - A key value dictionary of the robot capacities.

[1]Component isolation or sand-boxing is a security mechanism for separating running process. The code and data spaces are also separated for each process.

[2]RESTfull defines a set of architectural principles by which it can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages.

DeviceServices - A list of services. The list is composed of the service type presented above.

DeviceMobility - The mobility of the robot.

PingRatio - The quality of transmission between the requester and the desired robot.

URL	*Method*	*Description*
`/me/services/`	GET	Returns the registered service list
`/me/services/`	POST	Registers or updates a service in the local service list
`/me/services/`	DELETE	Deletes a local service from the list
`/me/capacities/`	GET	Returns the capacities of the robot
`/me/capacities/`	POST	Adds or update a capacity
`/me/capacities/`	DELETE	Deletes a capacity
`/neighbors/`	GET	Returns a full list of neighbors and full description of their services
`/search/capacities/ ?<value>=[><]<key>`	GET	Filters the list of neighbors for capacities in the URL query. > < can be used for comparison and | for regular expression
`/search/services/ <name>/ ?<value>=[><]<key>`	GET	Filters the list of neighbors for services with the specified name and metadata filters from the URL query. > < can be used for comparison and | for regular expression

Table 1: API description

A full description of each web-service provide by SDfR can be found in table 1. SDfR service is represented as Representational State Transfer [6] (RESTfull) web-service. It is based on normal HTTP requests which enables to infer the type of request being made and is completely stateless. All the responses are JSON messages.

The service that pilots SDfR protocol has some built-in additional features. Firstly, based on a simple configuration file, it is able to automatically connect to an ad-hoc network. The SSID network is composed using the fleet id and is secured using a WAP2 Personal password from the configuration file. This mechanism allows to have multiple fleets of robots in the same networked space. Even more, the robots can auto assign IP addresses. The standard network space is *10.<fleet id>.<x>.<y>*, where x and y are computed by each robot from their internal MAC address in order to avoid IP conflict [18]. Furthermore, if an IP conflict happens, the service has a mechanism to trigger an IP change on the robots.

5. BENCHMARKING

To evaluate our protocol we looked up the various impacts that it will have on the system composed by the robots. Firstly we were interested by the request time of a publisher that advertise its service and a subscriber that request information about the services on nearby neighbors. Secondly, we have analyzed the impact on the machine on which the SDfR runs, especially the CPU used. Finally, keeping in mind that the protocol should not use a great bandwidth, we analyzed the quantity of send and received bytes.

We have divided our benchmarks into 5 scenarios by using various number of Tuterbot 2 robots equipped with an Intel Core 2 Duo, 2.1 GHz CPU, 4Gb of Ram PC running on Ubuntu 13.04. We have used a number of 2, 4, 6, 8 and 10 robots. For each scenario, we have run 4 different ratios of publishers /subscribers: 20%/80%, 40%/60%, 60%40%, 80%/20%.

Nb Robots	*Pub/Sub ratio*	*Nb pub*	*Nb sub*
2	20%	4	16
2	40%	8	12
2	60%	12	8
2	80%	16	4
4	20%	8	32
4	40%	16	24
4	60%	24	16
4	80%	32	8
6	20%	12	48
6	40%	24	36
6	60%	36	24
6	80%	48	12
8	20%	16	64
8	40%	32	48
8	60%	48	32
8	80%	64	16
10	20%	20	80
10	40%	40	60
10	60%	60	40
10	80%	80	20

Table 2: Test-cases summary

In our test-runs we have used simulated services that want to register/subscribe into SDfR. We have a combined number of 10 publisher/subscribers on each robot. We have simulated three type of actions: publish, unpublish and subscribe.

- We have simulated new services that want to *publish* via a POST to `/me/services/` with a delay time between 1 and 10 seconds. In order to simulate publishers, we have used an Apache server on each robots that responds to the auto-discovery URL of each publisher.
- Each of the already published services could be *unpublished* with a random delay between 1 and 5 seconds via a DELETE to `/me/services/`.
- We have generated separated threads for each *subscriber* that perform GET requests on `/neighbors/` . Each thread will constantly request the table of neighbors from SDfR, in order to stress at maximum our protocol.

Each test-run was given 5 minutes to collect the data for our benchmarks. In table 2 is summarized our test cases.

We have analyzed the request time of a service that wants to publish itself in SDfR because it is an important metric for the use of the protocol. A service needs to have a fast response when it advertises its service in order to avoid blocking states. As displayed in fig. 6, the response time is between 106.2 and 107.2 ms which is satisfying. When the number of robots increases the request time remains limited, having a maximum increase of 0.5 ms per 10 robots. The publish request time also includes the auto-description URL check, where SDfR performs a request on the Apache Server.

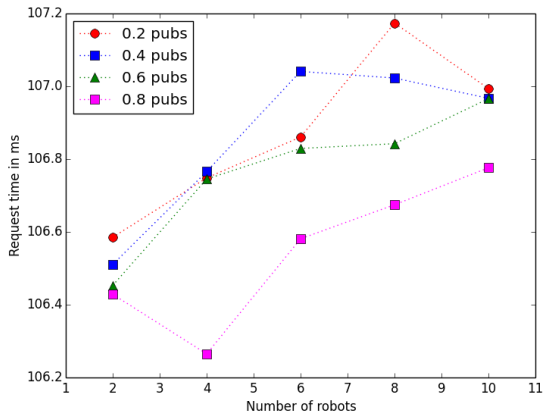

Figure 6:

In figure 7, we looked the request time of a service subscriber. Based on the number of subscribers per robot, we have generated parallel threads that requested constantly the neighbors table. We have observed that the request time is very low, between 1.7 and 2.9 ms even if the SDfR service was stressed at maximum. The trend for this graphic appears to be linear in the number of robots.

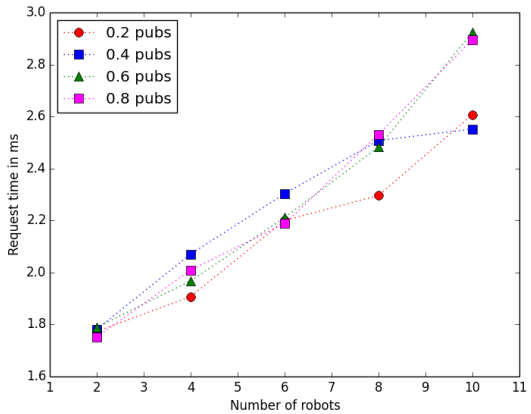

Figure 7: Service subscribe average time

In order to measures the weight of the service on the host machine, we have analyzed the CPU usage and the memory occupation. The CPU average usage was between 0.2% and 0.8% (see fig. 8). The average value is computed using all the peak measurements. The peak occupations represented 0.1% of the total test-run. If we compute over all the test-run, the average value is almost 0%.

The memory occupation was at almost 0.2% of the total memory of the PC regardless of the robots or publishers numbers (no graphics given).

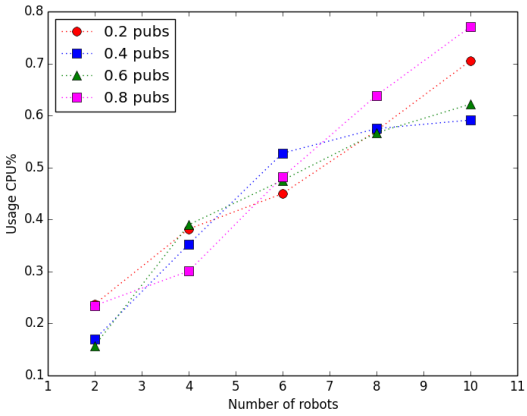

Figure 8: CPU % usage

Another import metric that we have analyzed during our test-runs is the quantity of bytes sent and received by each robot. Figure 9 and 10 present the evolution of the quantity of kilobytes transmitted and received by all the robots that took part in each scenario. The receptions are between 60 and 380 kilobytes during the 5 minutes run per robot, while the transmissions are between 50 and 800 kilobytes per robot. Those values are very satisfying because a robot is sending on average only 160 kilobytes/minute. The bandwidth occupied by SDfR is very low.

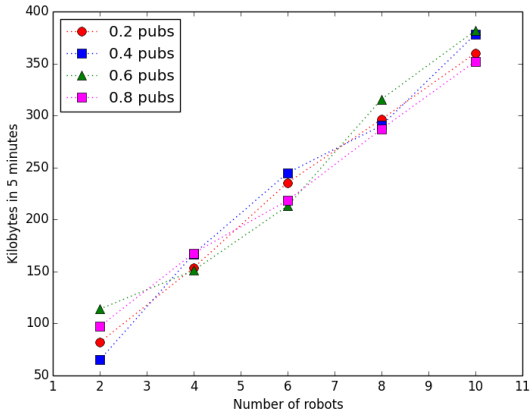

Figure 9: SDfR Transmission bytes

6. CONCLUSION

In this paper we presented the challenges to make an adapted service discovery protocol for multi-robot systems. We have discussed the limited applicability of existing service discovery protocols in the context of robot fleets. Then, we have proposed a new protocol, called SDfR, suitable for service discovery inside an ad-hoc networked fleet and we have presented our implementation as a service with a RESTfull interface in 'Go' language. Furthermore, we have evaluated our new protocol in terms of resource consumption on the robots.

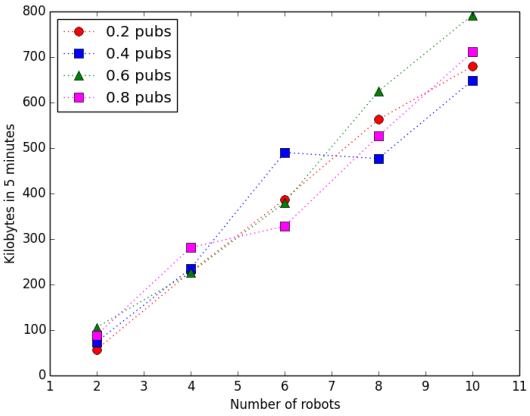

Figure 10: SDfR Reception bytes

These results show that SDfR is a light and useful brick in the service management in a robotic fleet even if a benchmarking with a larger number of robots should be made.

Our perspectives are to continue to enrich the families of middleware for robotics by adding other components that use SDfR service. Our vision is to improve the robotic fleet middleware by adding new service oriented bricks like P2P monitoring and P2P configuring.

In our approach to improve the family of middleware for multi-robot systems, we intend to use SDfR in order to build additional bricks of middleware that will simplify the process of development, but especially the process of deployment and operation of new robotic softwares. The next step is to investigate a new monitoring environment for the fleet of robots. We plan to provide a mechanism to monitor the fleet based on a peer to peer (P2P) protocol. This new service will be registered with SDfR, but will also rely on SDfR in order to obtain information about the nearby robots. The monitoring tool can be used by an end-user in order to survey the fleet, even the part of the fleet that is not reachable by the ad-hoc network.

In addition, we want to provide a simple mechanism to deploy new software and configure the robots. In our opinion, when a new software needs to be deployed, we do not need to deploy it manually on each robot. Even more, we consider that the automated deployment should be done in the environment where the robots are deployed, thus using a P2P deployment tool.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] R. Borja, J. de la Pinta, A. Álvarez, and J. Maestre. Integration of service robots in the smart home by means of UPnP: A surveillance robot case study. *Robotics and Autonomous Systems*, 61(2):153 – 160, 2013.

[3] S. Chitic, J. Ponge, and O. Simonin. Are middlewares ready for multi-robots systems? In *Simulation, Modeling, and Programming for Autonomous Robots - 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings*, pages 279–290, 2014.

[4] E. del Val, M. Rebollo, and V. Botti. Enhancing decentralized service discovery in open service-oriented multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 28(1):1–30, 2014.

[5] A. Elkady and T. Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 2012.

[6] R. Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[7] S. Frénot, F. Le Mouël, J. Ponge, and G. Salagnac. Various Extensions for the Ambient OSGi framework. In *Adamus Workshop in ICPS*, Berlin, Allemagne, July 2010.

[8] P. K. Gummadi, S. Saroiu, and S. D. Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *ACM SIGCOMM Computer Communication Review*, 32(1):82–82, 2002.

[9] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M. A. Gerosa, and A. B. Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.

[10] M. Klusch, B. Fries, and K. Sycara. Automated semantic web service discovery with owls-mx. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM, 2006.

[11] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[12] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[13] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pages 736–742. IEEE, 2008.

[14] A. Pereira, N. Costa, and C. Serôdio. Peer-to-peer Jini for truly service-oriented WSNs. *International Journal of Distributed Sensor Networks*, 2011, 2011.

[15] R. Pike. Go at google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 5–6, New York, NY, USA, 2012. ACM.

[16] D. Romero, R. Rouvoy, L. Seinturier, and P. Carton. Service discovery in ubiquitous feedback control loops. In *Distributed Applications and Interoperable Systems*, pages 112–125. Springer, 2010.

[17] P. Rompothong and T. Senivongse. A query federation of uddi registries. In *Proceedings of the 1st international symposium on Information and communication technologies*, pages 561–566. Trinity College Dublin, 2003.

[18] S. Thomson, T. Narten, and T. Jinmei. Ipv6 stateless address autoconfiguration. IETF RFC 4862, 9 2007.

[19] C. N. Ververidis and G. C. Polyzos. Service discovery for mobile ad hoc networks: a survey of issues and techniques. *Communications Surveys & Tutorials, IEEE*, 10(3):30–45, 2008.