Resilient Middleware for a Multi-Robot Team

Eric Vial, Mário Calha

FC/UL [**]
evial@lasige.di.fc.ul.pt
mjc@di.fc.ul.pt

Abstract. This paper addresses a resilient cooperative engine for robot teams within the context of the surveillance of physical areas. Due to the unreliability in the wireless communication between robots, a middleware must offer some resilience to the control application and guarantee that the robots never collide. We present an architecture for the robots to share a common view and to handle new events in a safe and resilient way. The system relies on two control sub-modules, the first one, the payload, could be complex and has access to information shared among robots, the second one, the wormhole is reliable but only uses local information. The system is evaluated by means of simulation tools and aims to be ported to hardware platforms composed by real mobile robots.

Resumo: Este documento aborda um motor cooperativo e resiliente para equipas de robôs no contexto da vigilância de áreas físicas. Devido à falta de fiabilidade na comunicação sem fios entre robôs, um middleware deve oferecer alguma resiliência à aplicação de controlo e garantir que os robôs nunca colidem. Apresentamos uma arquitectura que permite aos robôs partilharem uma vista comum e lidar com novos eventos de uma forma fiável e resiliente. O sistema apoia-se em dois sub-módulos de controlo, o primeiro, payload, pode ser complexo e acede à informação partilhada pelos robôs, o segundo, wormhole é confiável mas apenas utiliza informação local. O sistema é avaliado através de ferramentas de simulação e tem como objectivo ser implementado em plataformas de hardware compostas por robôs reais.

Keywords: group communication, middleware, mobile robots

1 Introduction

Mobile robot teams have the potential to reduce the need of human presence for complex or repetitive tasks. For most of them, the use of cooperation between robots can enhance the overall performance of the team. Achieving an efficient cooperation requires the use of complex algorithms implemented in each

[**] Faculdade de Ciências da Universidade de Lisboa. Bloco C6, Campo Grande, 1749-016 Lisboa, Portugal. Navigators Home Page: http://www.navigators.di.fc.ul.pt. This work was partially supported by the FCT through the LASIGE Multiannual Funding Program.

robot. This internal complexity in addition to the interaction issue with the environment makes the robot control system more sensitive to failures. Nowadays, building resilient control systems for mobile robots is a real challenge.

In this paper, we will present a middleware architecture for robots in charge of monitoring a physical area. In particular, we will focus on three architecture features which improve the system resilience: First, a control layer which relies on an hybrid approach, both synchronous and asynchronous. This layer involves two sub-systems the *payload* and the *wormhole* and guarantees a timely execution of critical tasks. The second architecture feature is the tree-oriented event structure used in the *payload* and based on small computation modules, This structure offers more stability to the system by avoiding cycles of events. Finally, as last feature, we will describe a group of modules in charge of managing a common world view for all robots and in particular a synchronization algorithm used during group merging or splitting phases. The algorithm is designed to be tolerant to communication failures.

The project context is a cooperative surveillance application of a given area. The covered zone is a campus, a plant or any well-defined area, each robot has a prior map of this environment. The purpose is to detect an accident or an intrusion and to build a common strategy to handle properly the detected event (e.g. blocking the intruder). All robots run the same version of the middleware and are equipped with local sensors, positioning and wireless devices.

This paper is organized as follows: The next section addresses the related work. In section 3, we provide an architecture overview. Section 4 gives details on the *wormhole* and *payload* model, the event-based architecture, and the world view synchronization algorithm. This latter part includes a short analysis with pros and cons. Finally, section 5 describes pending and future applications of the work and concludes the paper.

2 Related work

In the last twenty years, there has been a considerable amount of work to study mobile robot localization. Researches have been carried out focussing on two problems: computing absolute location using a priori map [6] or building incrementally this map while exploring the environment [5]. Both approaches most often rely on complex and math-oriented algorithms based on Kalman filters and maximum likelihood estimation [7]. The present paper does not address this kind of problem and we assume that the robot is equipped with a location device based on GPS or RSS technology [1]. In the same way, the way a robot team performs the surveillance of a physical area could obey many rules in order to maximize the probability of locating an intruder [8]. We wilfully chose not to optimize this part, the robot just wanders around the world, making random decisions to turn left or right at every crossing.

In the control architecture, the payload relies on a flexible and modular tree-oriented architecture. The idea is to break down the control layer into a chain

[1] Global Positioning System and Received Signal Strength

of small modules. Each one is triggered with an incoming event and is able to generate outgoing events up to others modules. This concept is a simplified application of the subsumption theory developed by Rodney A. Brooks [9] at the beginning of the 80's. Unlike many implementation projects based on this theory, we do not allow information cycles between module layers, modules are top-down triggered which minimizes the risk of an out-of-control diffusion of events. Finally, we took up the idea of suppressing some input signal to inhibit a group of modules which is similar to the Brooks suppressor concept.

In the vehicular domain, designing safety-critical application is essential, the work in [1], [2] or [3] presents a hybrid (synchronous and asynchronous) control model used in real-time applications. This model is based on two sub-systems, the *payload* in charge of running complex algorithms to figure out the best behaviour of each car to avoid collisions while the second sub-system, the *wormhole* is running synchronous and robust algorithms aimed to check whether the payload timely sends corrections to the car actuators. In case a timely timing failure is detected, the wormhole can temporally take control of the car. As this technique is applicable to any domain where a safety-critical control is mandatory, we used a payload-and-wormhole-based architecture for the robot control layer.

Robot soccer game is an entertaining and well-known application of robot team cooperation. Actually, it shows many common points with our project like the need for all robots to real-time maintain a common view of the world. In the paper [4], the authors present an approach of view model which was successfully implemented during the 2002 RoboCup Sony competition. Due to some high latency in wireless communication, the robot team does not perform any view synchronization. In order to track a dynamic object like the ball, each robot combines local information from vision sensors with shared information sent by team-mates. The robot maintains timestamps and uncertainty values for each view object, uncertainty is updated when receiving new information and grows with time. Unlike in [4], our world model is based on a view synchronization but we use certainty flags associated with timestamps to give more or less weight to an object position in the world view.

3 Architecture overview

In this section, we give an architecture overview through the description of three key features of the middleware, depicted in figure 1. The left side shows the middleware layer division. The *wormhole* and *payload* are the middleware basic components. The *wormhole* is placed in cut-through configuration between the *payload* and the sensor/actuator layer and does not have access to the network device. The *payload* runs all complex tasks in charge of the robot control. All tasks are triggered by events broadcasted through a module tree. An example of module tree used in the *payload* is shown in the right side of the figure. Each group of modules is dedicated to a specific task: Position update, navigation or world view management. Here, we will focus on the view management and especially the synchronization mechanism.

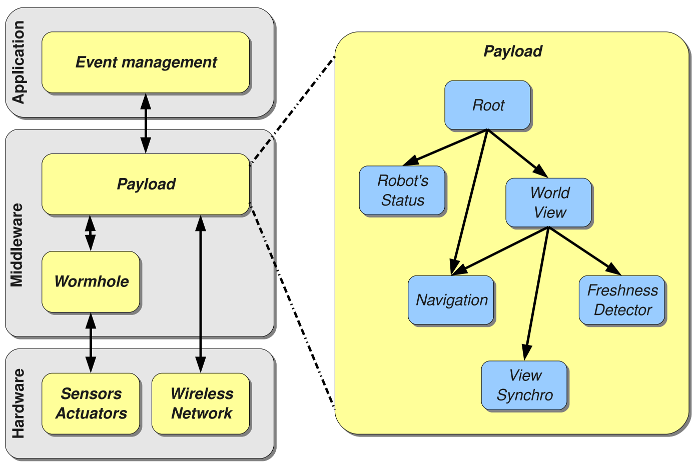


Fig. 1: Middleware layer division and module tree

3.1 Wormhole and payload model

The *payload*, may be asynchronous and runs complex and intelligent algorithms. These algorithms might not be deterministic and the computation time may vary especially if they require network communication. The wormhole is synchronous and runs simple and more robust algorithms. On the robot, some critical tasks like the collision avoidance require for the *payload* to timely send information to the hardware layer. In our case, this information will be new speed and heading commands.

The *wormhole*'s job is to check whether these commands are timely sent by the *payload*. If so, the *wormhole* will just forward them to the hardware layer. Otherwise, it will assume the robot's control by calculating and sending itself these commands. The *wormhole* uses a low-level navigation algorithm which only involves local information. The *wormhole* will keep the control until the *payload* starts again to send commands on time. If the *payload* does not regain stability or even no longer sends any command (the *payload* may have crashed), the *wormhole* can restart the whole *payload* process.

The *payload* sends the new commands in a structure called the *promise*. Each *promise* includes a deadline which enables the *wormhole* to control the *payload*'s timeliness. For each promise, the *payload* is expected to send the next command before the current deadline is exceeded.

3.2 Module definition

The event-based architecture is well adapted to robot management. It allows to design a flexible and modular architecture. Indeed, we can create an event type for any robot feature and dedicate a part of the tree to handle this event. Moreover, robot hardware is composed by sensors, actuators and communication devices, each one of them can generate a special event or be triggered by this event. The *obstacle* event is an example of event which contains the distance values read from the local sensor.

The *payload* is composed by modules and groups of modules, all gathered in a tree. Each of them is in charge of managing a robot feature. A module can be seen as a process, with a short computation time, which is started by a single incoming event and can produce one or more outgoing events.

We will detail later the different types of events but basically, an event is broadcasted from a given location in the tree until the leaves. Each module which can consume this event, is started. That way, various modules can run at the same time.

3.3 Team management and view synchronization

Given that robots can move away from each other and go beyond their wireless range, a group can be split in various sub-groups. The unreliability of the wireless network can also lead to isolate a single robot if it temporarily loses the Wi-Fi signal. When two groups merge together, a synchronization mechanism is necessary to consolidate the information of each group view. The *payload* includes such a dynamic mechanism which ensures that all robot views are the same inside the group. The synchronization task as all other *payload*'s tasks is triggered by events. The *synchro* event will be detailed in chapter 4.3.

4 Design and implementation

We will now detail the payload and wormhole architecture, the implementation of the module structure and finally the world view synchronization algorithm.

4.1 Wormhole and Payload implementation

The *wormhole* relies on three modules as shown in figure 2: The *Timely Timing Failure Detection* (TTFD) monitors the timeliness of the asynchronous *payload* process and can activate the *Safety task* to assume control. The *Control task* receives the *promise* which includes the new speed and heading values and decides whether these values can be forwarded to the actuator layer. The *TTFD* sends as well control updates to the *payload* to inform it won back or lost the control. Ideally, the *payload* should use these control updates to improve its performance. In particular, it could try to real-time adjust the priority of some internal processes.

Logical flowcharts of the *TTFD* and *Control* tasks are given in figure 3. The *payload* runs in three modes: *"active"* when it has the control, *"disable"* when it loses the control after the latest deadline is exceeded and finally in *"test"*, when the *wormhole* receives a timely promise while the *payload* is disabled. The *test* mode is a transition period, the *wormhole* keeps the control and waits for the *payload* to meet the current deadline before giving him back control.

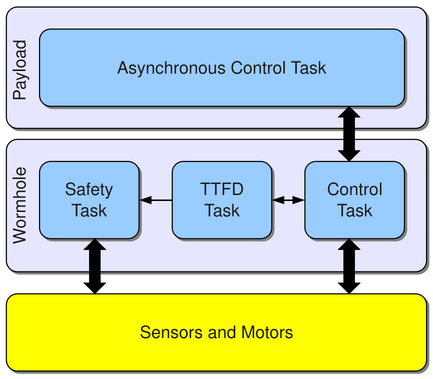


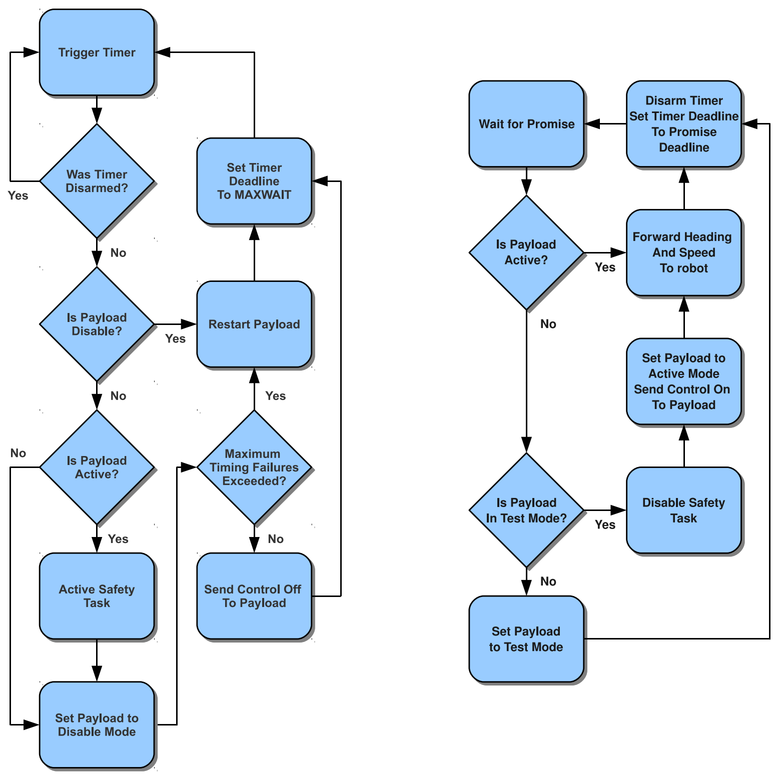
Fig. 2: Payload and Wormhole layers

There are two restart conditions for the *payload*. After setting it to *disable*, the *TTFD* task will increment a timing failure counter and will wait *MAXWAIT* milliseconds. If the failure counter is greater than a prior threshold or if no promise is received within this waiting period, the *TTFD* task will restart the *payload*.

4.2 Tree-oriented structure

The modules could be meshed as a graph and thus make event cycles possible. In order to avoid hazardous out-of-control cycles inside one robot or between several robots, we chose a tree-oriented module architecture. Each tree's branch has one or several parents. Events are top-down broadcasted until the leaves. A module computation is started if the current event can be consumed by the module. There are three types of events:

– **Hard events**: They are signals generated by the robot's hardware, e.g. the robot's clock (*beat* event) or a distance sensor measure (*obstacle* event). Such events are always broadcasted from the tree's root.
– **Local soft events**: These events are produced by a module and are broadcasted through the neighbour branches (modules with same parent) and the sub-branches. Any module can produce several soft events during the same computation.
– **Remote soft events**: Instead of being broadcasted locally, they are transmitted through the wireless network and sent to all other robots. Once delivered to a given robot, the event is broadcasted from the same branch as if it would be produced locally. This mechanism relies on two architecture properties: The module tree has the same structure for all robots which means that any path in the local tree matches the same path in a remote tree. Secondly, the path to locate the module which produced the event in the tree, is stored in the transmitted event. That way, we cannot have event


Trigger Timer
Was Timer Disarmed?
Yes
No
Is Payload Disable?
Yes
Restart Payload
Set Timer Deadline To MAXWAIT
No
Is Payload Active?
No
Yes
Active Safety Task
Set Payload to Disable Mode
Maximum Timing Failures Exceeded?
Yes
No
Send Control Off To Payload
Wait for Promise
Disarm Timer Set Timer Deadline To Promise Deadline
Is Payload Active?
Yes
Forward Heading And Speed To robot
No
Set Payload to Active Mode Send Control On To Payload
Is Payload In Test Mode?
Yes
Disable Safety Task
No
Set Payload to Test Mode

(a) TTFD task (b) Control task

Fig. 3: Wormhole logical flowchart

cycles between robots and the remote soft events meet the same constraints as the hard and local soft events.

Let's consider the module tree depicted in figure 4 with three robots in the same group. We assume that *e1* and *e2* are hard events, *e3* a local soft event and *e4* a remote soft event. Now, let's have a look on the started modules if *e1* is triggered on robot 1.

– Module 1 of Robot 1 (locally triggered by *e1*)
– Module 2 of Robot 1 (locally triggered by *e1*)
– Module 4 of Robot 1 (locally triggered by *e3*)
– Module 6 of Robot 2 (remotely triggered by *e4* with path *root.g1*.
– Module 6 of Robot 3 (remotely triggered by *e4* with path *root.g1*.

Although module 3 can consume the *e4* event, this module is not started in robots 2 and 3 because it cannot be reached from the path *root.g1*. What's

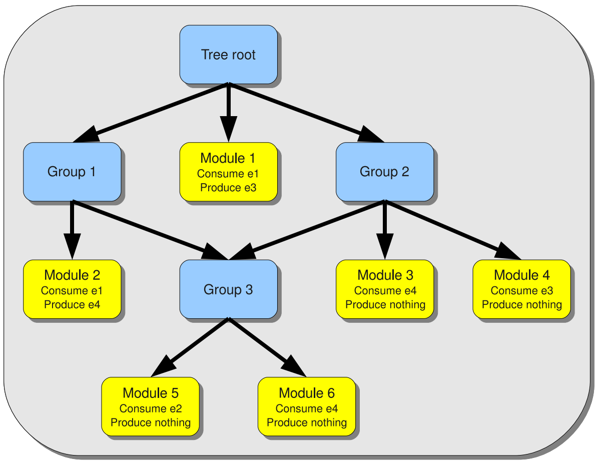


Fig. 4: Example of module tree common to robots 1, 2 and 3

more, this architecture opens the possibility of dynamically enabling or disabling a sub-branch of the module tree. When disabled, events are no longer broadcasted through this branch. The activation or disactivation can be performed by any computation module. In our project, each robot has three working modes: Wandering, Searching and Blocking (an intruder), each mode corresponds to a single branch of the navigation sub-tree. We use the branch enabling/disabling mechanism to activate the modules associated to the robot current mode.

4.3 World view synchronization algorithm

We will now describe in details the synchronization algorithm used to maintain in each robot a coherent world view when two or more groups are merging. This view is composed by all dynamic objects present in the world. The first part will deal with the algorithm principles and the second part will present some synchronization scenarios.

Algorithm overview: When two groups are merging, the synchronization is performed by exchanging a *synchro* event which contains the list of all view objects except for the robots position. This latter is already exchanged through the *hello* events so including this information in a *synchro* event would be redundant. The *synchro* event is normally sent by the group leader. The leader is the robot with the lowest id in the group. A group is identified by a single id. So all robots from a group share the same leader and group id. The *synchro* event reception is not centralized by the leader, each robot from the destination group, will handle the *synchro* event and extract the object list. The synchronization phase ends when all robots have the same leader and group id.

The basic steps below are associated to a faultless synchronization phase. By fault, we mean any event reception failure due for instance to a temporally

Wi-Fi signal loss. Different fault scenarios will be discussed in the next section. Two *synchro* events are exchanged during a faultless synchronization, the first *synchro* event is always generated by the group leader with the higher id.

– *Step 1*: Leader 1 receives a *hello* event from the group leader 2.
– *Step 2*: Leader 1 broadcasts a *synchro* event through the group 2.
– *Step 3*: Robots from group 2 receive the object list and update their view.
– *Step 4*: Leader 2 broadcasts a *synchro* event through the group 1.
– *Step 5*: Robots from group 1 receive the object list and update their view.

The algorithm 1 gives the modules involved in the synchronization phase. The *Hello Receiver* module (line 1) is triggered by an *object* event which is used by a robot to broadcast its own position, the module checks out whether this event comes from another leader. The *View Synchronizer* module (line 9) is triggered by a *synchro* event, it extracts the object list and updates the local world view (line 16). Finally, the *Freshness Detector* module (line 22) is triggered by the *beat* event and removes out-of-date objects from the robot's view. The *beat* is a hard event periodically generated by the system (see section 4.2).

In order to keep the algorithm clear, we won't detail below neither the *Hello Sender* module which is also triggered by the *beat* event and produces an *object* event, nor the *View Updater* module triggered by an *object* event and which updates the world view. The robot state parameters are as follows:

– *myId*: single robot identifier.
– *myView*: view objects including team-mate positions.
– *myLeader*: current group leader identifier.
– *myGroup*: current group identifier.

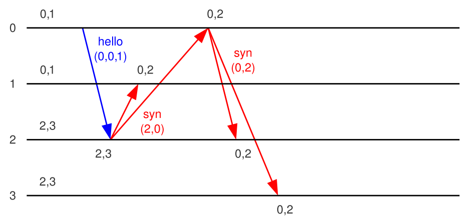
Examples of synchronization scenarios: Figures 5 and 6 show various synchronization scenarios with respectively two and three different groups merging at the same time. Each group is first composed by two robots, robots 0 and 1 for the first group, robots 2 and 3 for the second one and so on. Arrows identify events (blue for *hello* and red for *synchro* events) which are handled by robots for the synchronization phase. Other *hello* events broadcasted to periodically announce robot positions are not represented here. Each scenario is given as an example. Therefore, the number of events exchanged during a scenario could be different according to the order each event is delivered with. This statement is especially true if the number of groups merging at the same time is large.

In a robot time line, couple of black values correspond respectively to the leader and group id. The *hello* event parameters are the source robot, leader and group id. Finally, the *synchro* event parameters are the source and destination leader id (*leader1* and *leader2* in the algorithm 1).

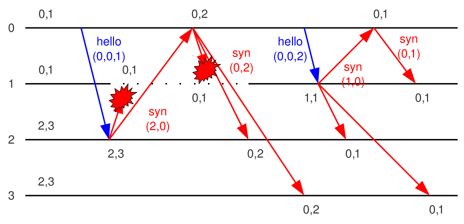
Scenarios 5b, 5c and 5d highlight temporally reception failures which lead the robot to broadcast extra events to achieve the synchronization. Such failures could be due to a temporary Wi-Fi signal loss. The extra event phase is initialized by the faulty robot which receives a *hello* packet from its leader. This mechanism is implemented at line 5 of the algorithm 1.

Algorithm 1 View synchronization algorithm

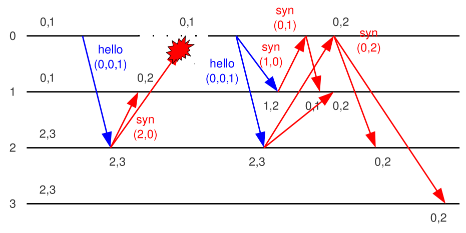
```
upon event <object | type, id, leader, group> do
    if $type = "robot" \wedge (leader \neq myLeader \vee group \neq myGroup)$ then
        if $id = leader \wedge myId = myLeader \wedge id < myId$ then
            $synchronization(myLeader, leader)$
        else if $id = myLeader$ then
            $synchronization(myId, leader)$
            $myLeader \leftarrow myId$

upon event <synchro | leader1, leader2, objList> do
    if $myLeader = leader2$ then
        if $leader1 < myLeader$ then
            $myLeader \leftarrow leader1$
        if $myId = myLeader$ then
            $synchronization(myId, leader1)$
        for all $obj \in objList$ do
            trigger <object | obj.type, obj.id, obj.leader, obj.group>
        if $leader1 > leader2$ then
            $myGroup \leftarrow leader1$
        else
            $myGroup \leftarrow leader2$

upon event <beat | > do
    $updateRequired \leftarrow false$
    for all $obj \in view$ do
        if $isUptodate(obj) = false$ then
            $view \leftarrow myView - \{obj\}$
            if $obj.type = "robot" \wedge (obj.id = myLeader \vee obj.id = myGroup)$ then
                $updateRequired \leftarrow true$
    if $updateRequired = true$ then
        $updateLeader()$

procedure SYNCHRONIZATION(leader1, leader2)
    $objList \leftarrow \{\}$
    for all $obj \in myView$ do
        if $obj.type \neq "robot"$ then
            $objList \leftarrow objList + \{obj\}$
    trigger <synchro | leader1, leader2, objList>

procedure UPDATELEADER
    $myLeader \leftarrow myId$
    $myGroup \leftarrow myId$
    for all $obj \in myView$ do
        if $obj.type = "robot" \wedge obj.id < myLeader$ then
            $myLeader \leftarrow obj.id$
        if $obj.type = "robot" \wedge obj.id > myGroup$ then
            $myGroup \leftarrow obj.id$
```



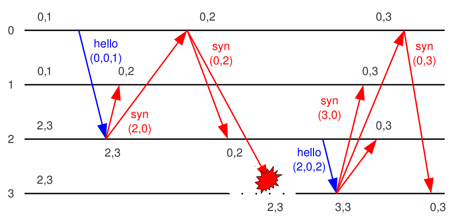
(a) Synchronization without failure



(b) Reception failure on robot 1



(c) Reception failure on robot 0



(d) Reception failure on robot 3

Fig. 5: Examples of view synchronization between two groups

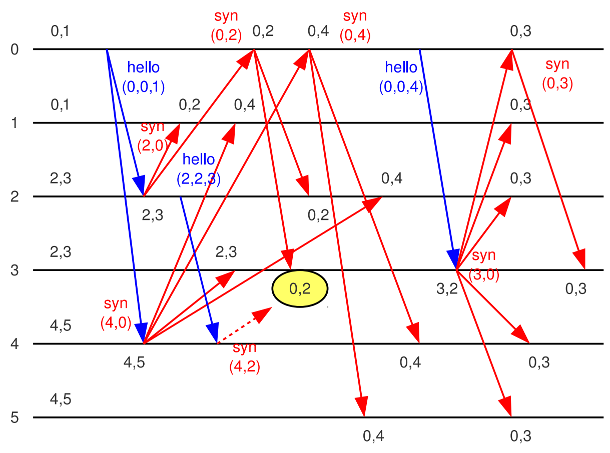


Fig. 6: Example of view synchronization between three groups

The last scenario 6, three groups merging at the same time, is unusual but shows the algorithm resilience. We can notice that at the end of the first "round", the robot 3 doesn't have the same group id than the others (yellow-circled value). The situation gets stable after the second *hello* event.

Advantages and drawbacks: This algorithm is very simple and offers resilience in signal loss situations. Nevertheless in some tricky scenarios, it could require more than one round, i.e. more than one *hello* event to stabilize itself. *Hello* events are periodically generated (according to the *beat* signal frequency), so increasing this beat signal frequency to accelerate the synchronization phase could be attractive but may on the other hand overload the wireless network and what's more, lead to some algorithm instability if this frequency is greater than half the mean round trip delay of the wireless network.

5 Conclusion and future work

We have proposed a middleware architecture aimed to offer a resilient control system for mobile robots. A middleware version was written in C and evaluated by means of robot simulation Java tools (Simbad v1.4). Most common scenarios like communication failures, robot group splitting and merging, or payload overload have been successfully tested. The next step is now to port the middleware to an embedded platform based on an ARM chip and a FPGA.

References

1. Luís Marques, António Casimiro, Mário Calha, Design and development of a proof-of-concept platooning application using the HIDENETS architecture, DSN '09: Proceedings of the International Conference on Dependable Systems and Networks, 223-228, 2009.
2. Casimiro, A. and Rufino, J. and Marques, L. and Calha, M. and Veríssimo, P., Applying architectural hybridization in networked embedded systems, The Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, November, 2009.
3. Paulo Veríssimo. Travelling through wormholes: a new look at distributed systems models. SIGACT News, 37(1):66–81, 2006.
4. Maayan Roth, Douglas Vail, and Maria Manuela Veloso, A Real-time World Model for Multi-Robot Teams with High-Latency Communication. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), October, 2003.
5. M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. IEEE Transactions on Robotics and Automation, 17(3):229–241, 2001.
6. J. J. Leonard and H. F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. IEEE Transactions on Robotics and Automation, 7(3):376–382, 1991.
7. Andrew Howard, Maja J Matarić and Gaurav S Sukhatme. Localization for Mobile Robot Teams Using Maximum Likelihood Estimation. IEEE/RSJ International Conference of Robotics and Intelligent Systems (IROS): 434-459, 2002.
8. Kristel Verbiest, Eric Colon. Securing Hostile Terrain with a Robot Team. Fourth International Workshop on Robotics for risky interventions and Environmental Surveillance-Maintenance, RISE'2010– Sheffield, UK, January 2010
9. Rodney A. Brooks, A Robust Layered Control System For a Mobile Robot, Massachusetts Institute of Technology, Cambridge, MA, 1985.