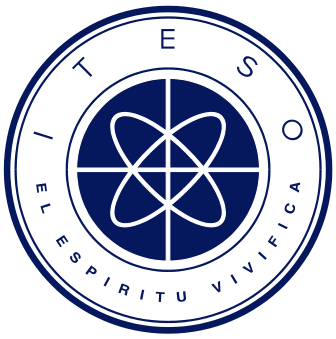
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS



SISTEMA EMBEBIDO BASADO EN IOT Y TECNOLOGÍA MULTI-AGENTE PARA EL CONTROL DE TRÁFICO VEHICULAR

Trabajo recepcional que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Rolando Santoyo Rincón

Asesor: Raúl Campos Rodríguez

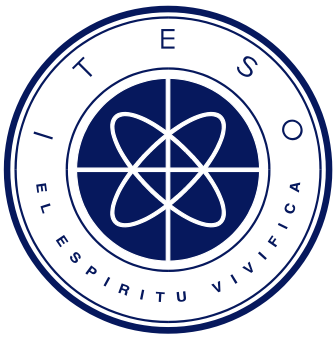
Tlaquepaque, Jalisco. Mayo de 2016.

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS



SISTEMA EMBEBIDO BASADO EN IOT Y TECNOLOGÍA MULTI-AGENTE PARA EL CONTROL DE TRÁFICO VEHICULAR

Trabajo recepcional que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Rolando Santoyo Rincón
Becario CONACYT No. 168906

Asesor: Raúl Campos Rodríguez

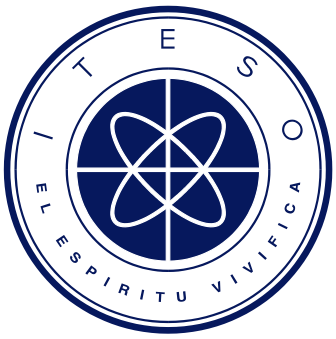
Tlaquepaque, Jalisco. Mayo de 2016.

TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS



EMBEDDED SYSTEM BASED ON IOT AND MULTI-AGENT TECHNOLOGY FOR THE CONTROL OF VEHICULAR TRAFFIC

Receptional document to obtain the diploma of

EMBEDDED SYSTEMS SPECIALIST

Author: Rolando Santoyo Rincón
CONACYT Scholarship No. 168906

Advisor: Raúl Campos Rodríguez

Tlaquepaque, Jalisco. May, 2016.

Sistema Embebido Basado en IoT y Tecnología Multi-Agente para el Control de Tráfico Vehicular

Resumen

La transportación de personas es quizá uno de los principales problemas que la población en las grandes ciudades tienen que enfrentar día con día, debido a que los sistemas de transporte no pueden satisfacer las necesidades actuales que requiere la movilidad urbana, afectando con esto la productividad y la calidad de vida de los habitantes.

Estos problemas de transportación se pueden clasificar en las siguientes áreas:

- Grandes distancias a recorrer
- Transporte público de poca calidad e inadecuado.
- Dificultades para vehículos no motorizados
- Congestión vehicular
- Dificultades de estacionamiento
- Perdidas de espacios públicos
- Alto costo de mantenimiento
- Impacto al ambiente y alto consumo de energía
- Accidentes y seguridad
- Distribución de mercancías

De entre esta lista de áreas, la congestión vehicular es uno de los problemas que más prevalecen en áreas urbanas con altos niveles de población (arriba de 1 millón de habitantes). Ese problema está altamente relacionado con el incremento de vehículos motorizados (ante falta de un transporte público eficiente) lo cual trae como consecuencia que se requiera una mayor infraestructura de transporte. Esta infraestructura no ha sido capaz de crecer al mismo ritmo con que lo hace el parque vehicular, por lo que esto deriva en problemas de congestionamiento vehicular. A su vez, el congestionamiento en si se traduce en problemas tales como largos periodos de tiempo de viaje, pérdida de productividad, contaminación en el aire, estrés en los conductores de vehículos, exceso de gasto el combustible, etc.

Existen diversas medidas que se han intentado para resolver el problema de congestionamiento, tales como:

- Control de acceso a avenidas o carreteras congestionadas.
- Sincronización de señales de tráfico, las cuales se pueden ajustar por horarios definidos por patrones de tráfico. Esto incluye también el uso de sistemas inteligentes que reciben información de sensores para detectar la presencia de carros o peatones que intentan cruzar la calle.
- Restricción por día del uso de vehículos.
- Carpooling en el cual carros con 2 o más pasajeros pueden usar un carril preferencial.
- Transporte publico tales como el metro, tren ligero o autobuses.

Estas medidas pueden solucionar de manera parcial el problema de congestionamiento, pero no resuelven el problema de manera definitiva [1].

Precisamente, la motivación de este trabajo se enfoca en tratar de reducir el problema de congestión vehicular aplicando nuevas tecnologías para extender la solución basada en la sincronización de la señales de tráfico. Como se sabe, la carga de tráfico vehicular no es constante, es decir que varía continuamente durante el día dependiendo de diferentes factores tales como las horas pico en donde las personas tiene que llegar a sus lugares de trabajo o estudio, si hay algún evento especial con gran acumulación de personas, si existe algún accidente cercano o simplemente mantenimiento de una vialidad.

Basado en esto, la solución propuesta en este trabajo pretende crear un sistema de control de tráfico vehicular (semáforos) que basado en la detección en tiempo real de la carga o densidad vehicular en las calles donde se desea controlar dicho tráfico, el sistema pueda determinar la prioridad con la que los semáforos en la intersección pueda permitir el flujo de tráfico controlando de manera inteligente el tiempo en verde asignado a cada una de las calles reduciendo así la congestión vial en tal intersección.

Para lograr esto, el sistema se manejara como un sistema multiagente, en donde cada uno de los semáforos en la intersección (o nodo) sería un agente, los cuales puede establecer comunicación entre si para compartir información entre ellos. Dicha comunicación entre agentes se puede lograr de manera sencilla y eficiente usando plataformas enfocadas a IoT (Internet de las cosas en inglés) de las cuales existen diversas opciones en el mercado.

Las características del sistema propuesto en este trabajo se describen a continuación:

- El Sistema es implementado utilizando la tarjeta de desarrollo Galileo (una tarjeta por semáforo), la cual está enfocada para sistemas embebidos y que proporciona soporte para comunicación basada Ethernet. Esta trajeta cuenta con un sistema operativo de libre distribución basado en Linux llamado Yocto, el cual permite instalar una máquina virtual de Java para poder ejecutar código de Java y soportar la librería o middleware JADE, la cual proporciona funciones específicas para sistemas multi-agente.
- El software desarrollado para el sistema de control de tráfico es implementada en Java usando una metodología multi-capas, la cual incluye capa de configuración, drivers y aplicación.
- Es un sistema multi-agente en el cual cada semáforo es un agente. Cada agente puede comunicarse con el resto de los agentes en el sistema siendo capaz de recibir y enviar mensajes para compartir información, hacer alguna petición o notificación.
- Cada uno de los agentes en el sistema se convierte en el agente leader por cierto periodo de tiempo, en el cual este controla el estado de las lámparas del resto de los agentes.
- El sistema propuesto puede ser fácilmente adaptado para cualquier topología, ya que el estado de las lámparas y transiciones está definido en una tabla de transición. De la misma manera el sistema puede ser adaptado para soportar diferente número de lámparas en los agentes.
- El sistemas también esta parametrizado, de manera que es posible adaptarse de manera sencilla a las necesidades de un nuevo nodo o intersección.
- El sistema puede operar en 2 diferentes modos para seleccionar al agente leader: de forma predefinida siguiendo el orden establecido en la tabla de transición o basado en la densidad de tráfico que cada uno de los agentes reportar. En ambos modos el agente líder es seleccionado de manera coordinada con el resto de los agentes por medio de mensajes intercambiados entre los agentes.
- El sistema es capaz de manejar sobre carga de tráfico, para lo cual el agente que la reporte tendrá una prioridad aún más alta para permitir el flujo de vehículos.
- El sistema es capaz de recuperarse por sí mismo de alguna posible falla de comunicación.

- El sistema soporta el manejo de peticiones de emergencia de un agente externo (como pudiera ser una ambulancia o una central de control) los cuales permiten poner al sistema en un estado específico por cierto tiempo (esto especificado dentro del mensaje).
- The system supports attending emergency request from external agents (such as ambulance or main control office) which ask the complete system to transit to a specific state during certain time.

En el marco teórico de este documento se provee de información adicional acerca del problema de congestión y sistemas modernos de control de tráfico. También se explica en detalle los que son sistemas inteligentes y sistemas multiagentes. Así mismo se explica lo que es el fenómeno de Internet de las Cosas (IoT).

En el documento se explica a detalle la propuesta propia para resolver el problema de congestionamiento vehicular, así como una explicación de las herramientas utilizadas para esto (la tarjeta Galileo, la librería Jade, etc).

El documento continua dando detalles de implementación del sistema propuesto, mencionando la topología seleccionada para el sistema de control de tráfico. Se explica a detalle la especificación del software desarrollado, proveyendo la descripción de cada una de las capas que lo conforma y explicando los algoritmos, métodos, máquinas de estados y diversos flujos que se implementaron.

También se provee de varios ejemplos o escenarios en donde el sistema fue probado, mostrando resultados obtenidos de ellos con lo cual se puede verificar que el funcionamiento del sistema es el esperado.

Finalmente se provee de algunas conclusiones que se obtuvieron de la implementación de este sistema y de algunas mejoras a este que pudieran ayudar a mejorar su desempeño o funcionalidad.

EMBEDDED SYSTEM BASED ON IOT AND MULTI-AGENT TECHNOLOGY FOR THE CONTROL OF VEHICULAR TRAFFIC

Abstract

Vehicle congestion is perhaps the biggest problem related to transportation that the inhabitants of big metropolis must face day by day, generating problems such as long commute periods, waste of productive time, air pollution, stress in drivers, overspending on fuel for vehicles, etc. Several solutions has been attempted to mitigate the problem, but so far this effort has not been enough. The work presented here pretends to attack this problem, going further with one of these solutions which consist in providing of additional intelligence to the traffic light controller systems in a street intersection, so they can administrate in a better way the traffic flow. This is achieved by taking advantage of the new embedded system technologies applied to IoT by monitoring and sharing in real time the traffic density ($\delta\tau$) in each one of the streets that converge in the intersection and based on this data, dynamically assign "time in green" proportionally to the $\delta\tau$. The proposed system is based on a multi-agent approach, in which each one of the traffic lights in an intersection is an individual agent which can communicate with the other agents through messages to exchange information(for example the $\delta\tau$). The proposed system uses a "leadership shared" scheme in which each one of the agents becomes the agent leader from time to time being this agent the one which calculates the lamp transition times in whole the system. This leader agent is selected in agreement with all the agents in the system, following either a pre-defined way or based on the $\delta\tau$ reported by each agent. The proposed system was implemented using Intel Galileo boards (one per agent) creating a software stack which includes **Yocto** as OS, a Java Virtual machine, a multi-agent middleware called **Jade** and a multilayer software application. The system proposed is flexible enough to be adapted to any traffic light topology or any number of lamps in the traffic lights. This also supports the handling of over-traffic and emergency conditions, and it is able recover itself of any scenario of loss of communication or energy. The system was tested using a specific topology under different scenarios demonstrated that the behavior of the system works as expected.

Table of Content

Table of Figures

Chapter I. INTRODUCTION

Motivations and Problem Background

Transportation is perhaps the main problem that the population in big cities has to face every day as the transport systems can't satisfy the requirements of urban mobility affecting the urban productivity and the life quality of the inhabitants.

The transportation problems that are seen in big metropolis can be classified in any of these areas:

- Longer commuting
- Public transportation inadequacy
- Difficulties for non—motorized transport
- Traffic congestion
- Parking difficulties
- Loss of public spaces
- High maintenance cost
- Environment impacts and energy consumption
- Accidents and safety
- Freight distribution

Congestion is one of the most prevalent transport problems in large urban agglomerations, usually above a threshold of about 1 million inhabitants. It is particularly linked with motorization and the diffusion of the automobile, which has increased the demand for transport infrastructure. However, the supply of infrastructure has often not been able to keep up with the growth of mobility. The congestion is also directly related to some other problems like the **parking difficulties** as many people slow down looking for a parking spot, the **delivery of products** in which most of the times the vehicles double park to unload the products and the **freight distribution** in which the slow heavy trucks use the cities roads to move the freight from one place to other.

Some of the consequences of congestion are long commute periods which is translated in waste of productive time, air pollution, stress in drivers, overspending on fuel for vehicles.
In order to prevent the congestion, several solutions has been attempted:

- Controlling the access to congested highways
- Traffic signals synchronization which can be adjusted to reflect changes in commuting patterns. This also can include some kind of intelligent system which receives information from sensors to detect the presence of cars or to detect that a pedestrian needs to cross the street.
- Car usage restriction which prevents that some cars being used at some specific day of the week.
- Carpooling in which cars with 2 or more passengers can use a preference lane.
- Public transportation such as subway, light rail and buses which offers some alternative to drive.

All these measures only address the issue of congestion partially, as they alleviate, but do not solve the problem [1].

The motivation of this work is focus on trying to reduce the congestion problem in the big cities extending the solution based on traffic signal synchronization. As it is known that the traffic load in the city roads changes continuously during the day and this load is also different day by day. So, the proposed solution pretends to create a traffic light system which based on knowing the traffic load in each one of the streets

converging in to one intersection, it will be able to determine priorities to each one of the traffic light signals, allowing the flow of vehicles during more time in the street with more traffic density, trying with this to reduce the congestion in that intersection. In order to achieve this, each one of the traffic signals is handle as an agent which can stablish communication with the other traffic signals (agents) in the intersection. This communication will create a multi-agent IOT (Internet of the Things) system and the information shared among the agents (i.e. Traffic load or emergency messages, etc) will be used to determine which of the agent or agents will allow the traffic flow and for how long.

The solution presented in this work proposes a methodology in which the system can be easily adapted to any number of traffic signals in any possible topology (i.e. node configuration such as single intersection, double way intersection, traffic circle, etc.)

Each one of the agents is implemented as an embedded system using JADE, which is a software library fully implemented in JAVA language which simplifies the implementation of multi-agent systems through a middle-ware that complies with FIPA specification [2]. In order to implement each one of the agents, it was used the Intel Galileo IOT embedded platform which provides support to Ethernet communication and allow to have a Linux distribution as RTOS which supports to install a Java virtual machine which is needed to run JADE.

Urban Traffic Congestion

As mentioned before, traffic congestion is one of the biggest problems in big cities. Congestion occurs when the transport demand is bigger than the transport supply. In this case each transportation vehicle impairs the mobility of others. So the transport system (including the roads) can be seen as a limited resource that needs to be share among all the people that require to move from one point to other in the city.

In the last decades it has been seen the extension of roads in urban areas, but the growth of urban circulation occurred at a higher rate than the expected by the time new roads were created. So, this demands the building or extending the roads, creating new circulation problems during the construction. Once the construction finishes is perceived as if the city is ready to support more vehicles, which increases the automobile dependency creating new circulation problems and a vicious circle of congestion.

Related to the type of vehicles that circulates over the roads, the urban congestion can be split in 2 domains which normally use the same infrastructure.

- **Passengers.** This is the most common domain in the latest years as the ability to own a car has been facilitated. The access to an automobile provides flexibility in terms of choice the origin, destination, route and travel time. The majority of the congestion is seen during the commuting hours and after that, the vehicle circulation gets normally relaxed.

- **Freight.** This is related to the heavy trucks that circulates over the same roads than the rest of the passenger vehicles during the pickup or delivery of freight. The congestion in this domain is commonly linked with a drop in the frequency of deliveries increasing the capacity to make sure to provide the similar level of service reducing costs.

At this point, it's important to mention that congestion in urban areas is mainly caused by commuting patterns and not much by truck movements.

Important travel delays occur when the roads capacity limit is reached which is the case of almost all of the metropolitan areas. Normally, now the road traffic is slower than what it was 100 years ago, so marginal delay is increased and the driving speed becomes problematic as the level of population density increases. Once that the population in a city exceeds the 1 million, cities start experiencing recurring congestion problems due to numerous factors such as the urban setting, modal preferences and quality of existing urban transportation infrastructure.

The congestion problem was getting more acute during and after 1990s. For example, in China the average car travel speed is now close to 20 km/hr and traffic density exceeds 200 cars per km of road. This figure is comparable to many of the cities in developed countries. Also, related to the congestion time there are 2 major forms:

- **Recurrent congestion**. This is related to the regular demand on the transportation system, such as commuting, shopping or weekend trips. However, even recurrent congestion can have unforeseen impacts in terms of its duration and severity. Mandatory trips are mainly responsible for the peaks in circulation flows, implying that about half the congestion in urban areas is recurring at specific times of the day and on specific segments of the transport system.
- **Non-recurrent congestion**. Non-recurrent congestion is linked to the presence and effectiveness of incident response strategies. As far as accidents are concerned, their randomness is influenced by the level of traffic as the higher the traffic on specific road segments the higher the probability of accidents. Also, congestion due to accidents is directly related to the driving behavior of the people which can be drastically different from one city to another.

Behavioral and response time effects are also important as in a system running close to its capacity, simply breaking suddenly may trigger what can be known as a backward traveling wave. It implies that as vehicles are forced to stop, the bottleneck moves up the location it initially took place at, often leaves drivers puzzled about its cause. The spatial convergence of traffic causes an overload on transport infrastructure up to the point where congestion can lead to the total immobilization of traffic. Not only the massive use of the automobile has an impact on traffic circulation and congestion, but it also leads to the decline in public transit efficiency when both are sharing the same roads. For more information about these facts, see [1].

Congestion explained from a technical point of view.

The fundamental cause of congestion is the friction between the vehicles in the traffic flow. Until certain threshold of traffic flow, the vehicles can flow at a relative free speed specified by the maximum speed limits, intersection frequencies, etc. Although when the volume of vehicles is increased each vehicle becomes an obstacle for others and therefore the congestion phenomena arises. Then a possible objective definition would be: "Congestion is the condition that stands if the adding of a new vehicle into the traffic flow makes the circulation time of the other vehicles increases". As long as the traffic flow increases, the circulation speed is drastically reduced. The figure 1 shows a curve $\boldsymbol{t = f(q)}$ that describes the time $(\boldsymbol{t})$ required to circulate at different traffic volume $(\boldsymbol{q})$. The other curve shown in the figure 1, $\boldsymbol{\delta(qt)/\delta q = t + qf'(q)}$, is generated from the former. The difference between both curves represent, for any traffic

volume (q) the increase of travel time of the vehicles that are circulating due to the introduction of an additional vehicle.

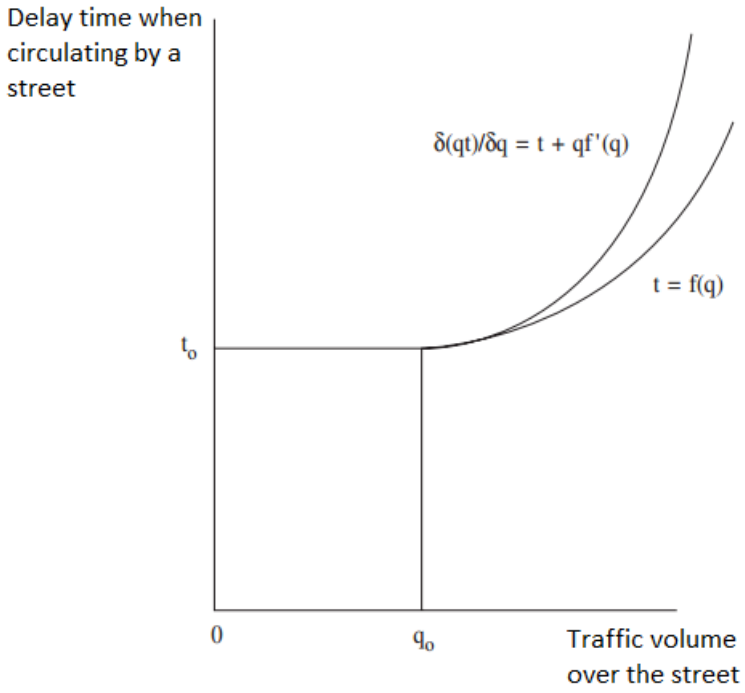


Figure 1. Curves that represent Congestion concept.

It can be seen that both curves coincide until the level of the traffic volume $0q_0$; until this point the change in the travel time of the vehicles is simply the time used by the vehicle that gets incorporated, because the others still can flow at the same speed than before the incorporation. In the other way, from this point on, the two functions diverge by $\boldsymbol{\delta(qt)/\delta q}$ above of t. This means that each vehicle that gets incorporated experiments its own delay, but also it increases the delay of the others that are already circulating. In consequence, the user of the vehicle that gets incorporated perceives only a small part of the congestion that it causes, but the rest of the vehicles are the ones that are really affected by this [3].

Once that the congestion problem is presented, and after noticing that the traffic load varies at every moment of the day and the causes can be varied, it's obvious that in order to mitigate the problem, some kind of intelligence need to be applied to the traffic control systems. In the next sections it will be cover some basic ideas about the intelligent systems and how they can be applied to the traffic control systems.

Intelligent Systems

In [4] an intelligent system is defined as a machine with an embedded computer connected to the Internet that has the capacity to collect and analyze data, communicate with other systems and take the actions for it was designed. Some of the requirements of an intelligent system incudes security, connectivity, ability to adapt itself according to current data and the capacity for remote monitoring and management.

Being more emphatic, in [4] the definition of an intelligent system is extended as anything that contains a functional embedded computer with internet connectivity. An embedded system may be powerful and capable of complex processing and data analysis but it is usually specialized for tasks relevant to the host machine.
Intelligent systems exist everywhere in our daily life. We can see them in point-of-sale (POS) terminals, digital televisions, traffic lights, smart meters, automobiles, digital signage and airplane controls, among a great number of other possibilities. As this ongoing trend continues, many foresee a scenario known as the Internet of the Things (IoT), in which objects, animals and people can all be provided with unique identifier and the ability to automatically transfer data over a network without requiring human-to-human or human-to-computer interaction [4].

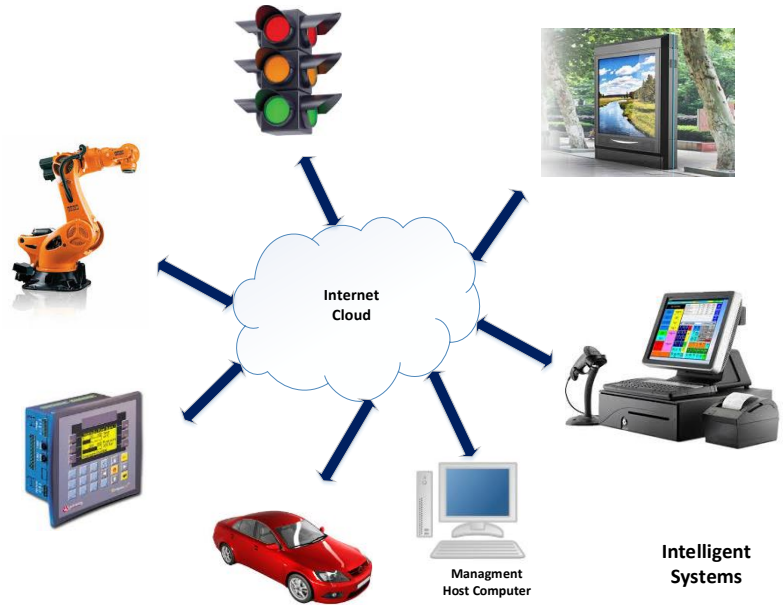


Figure 2. Intelligent systems around our life.

Intelligent Transport Systems (ITS)

The recent technologic development has supported ITS making possible the development and implementation of a wide range of traffic management techniques. The computing and communications applied to ITS have contributed in improving the safety, efficiency, and convenience of road transport, both for people and for freight. ITS is visible today in the most developed countries, as roads are equipped with electronic tolling and variable message sign (VMS); passenger vehicles have navigation systems and emergency notification systems; and public transport vehicles have location and communication systems.

Transport infrastructure in cities has also improved through systems automatically tracking, monitoring and supporting the better management of road traffic. This wide variety of fields and applications confirms that ITS have gained widespread acceptance within the transport community and by the general public. In fact, many cities already use ITS fairly extensively, while increasingly more plan to do so in the next years. The main technologies that have so far been utilized for transport purposes are wireless communications, computational technologies, floating car data, sensing technologies, inductive loop

detection and video vehicle detection technologies. The robustness of the technologies used offers great flexibility to ITS, which is one of their major advantages, as cities have the ability to tailor a method or a system to fit their individual needs and future plans.

These are the three main fields ITS have contributed: congestion management; safety and enforcement applications; and information provision.

In terms of congestion management, common example ITS applications include:

- Electronic toll collection that make faster the process of charging a fee to use a road.
- Variable speed limits that assist in the management of traffic by reducing the congestion build-up rate.
- Traffic light signaling synchronization and coordination; and traffic adaptive controlling to reduce the travel time of vehicles.

In terms of road safety and enforcement, ITS have applications such as:

- Intelligent speed adaptation that warns or prevents drivers from exceeding the local speed limit.
- Automatic road enforcement that by using cameras and sensors can identify vehicles disobeying rules. Some examples are red light cameras, bus lane cameras and speed cameras.
- Emergency vehicle notification systems that can generate emergency calls either manually or automatically using in-vehicle sensors.
- Collision avoidance systems that notify drivers about stalled vehicles and traffic incidents on their route.

Finally, in terms of information provision, example ITS applications include, among others:

- The provision of real-time traffic information through various means, such as VMS, mobile phones, radio etc.
- The supply of anticipated travel times, as well as personalized route guidance through satellite navigation systems and mobile phones.

Most of the ITS solutions so far are focus on private transportation. ITS can help manage different modes of transport through signal phasing and lane control measures. Furthermore, on-trip information and provision of multi-modal journey planning support the delivery of better public transport systems. So far a limited number of ITS applications are oriented towards public transport, with the most representative being:

- Traffic signals programmed to prioritize public transport. Applications so far have demonstrated public transport travel time reductions of 10-15% without observable delays to private transport.
- Bus location data used to provide real-time public transport information at bus stops and to improve travel information services.
- Demand-responsive systems, such as booking journeys on a door-to-door bus service. [5]

Traffic Light Control and Coordination

In [6] the traffic lights or traffic signals are descried as signaling devices positioned at road intersections, pedestrian crossings and other locations where is needed to control the flow of traffic. In order to work properly, the traffic lights require to pay attention in the control and coordination to ensure that the traffic moves as smoothly and safely as possible and also protect pedestrian when they cross the roads. In order

to achieve this, there are several mechanism ranging from simple clock synchronization to sophisticated computerized control and coordination systems that self-adjust to minimize the delay to people using the road.

In modern traffic light systems, the control is included in a cabinet that typically contains a power panel, to distribute electrical power in the cabinet; a detector interface panel, to connect to loop detectors and other detectors; detector amplifiers; flash transfer relays; a police panel, to allow the police to disable the signal; and other components.

Traffic light controllers use the terminology of phases, which are direction of movement grouped together. In a simple intersection exists 2 phases (North/South and East/West). Also the controllers can use rings, each ring is an array of independent timing sequences.

A traffic light controller includes a conflict monitor unit (CMU) which ensures the correct operation (probe of fail). The lamp outputs of the controller is monitored by the CMU, and if a fault is detected(i.e. invalid scenario), the CMU uses the flash transfer relays to put the intersection to FLASH, with all red lights flashing.

Traffic lights receive instructions about when to change phase. It is common they are coordinated so that the phase changes occur in some relationship to other close traffic controller signals or to the press of a pedestrian button or other inputs signals (for example the action of a timer).

In [6] the traffic controllers are classified in 2 ways as described next:

- **Fixed time control.** These use timers that have fixed, signalized intersection time plans. The cycles ranges from 35 to 120 seconds. And once stablished, they only can be modified manually. So the cycle time needs to be calculated based on the traffic flow at certain point but as the traffic flow may change at any time, this could be inefficient.

- **Dynamic control.** In this case it's used information feedback from detectors, which are sensors that inform the controller processor whether vehicles or other road users are present, to adjust signal timing and phasing within the limits set by the controller's programming. It can give more time to an intersection approach that is experiencing heavy traffic, or shorten or even skip a phase that has little or no traffic waiting for a green light. Also, in [6] the detectors can be grouped into three classes: in-pavement detectors, non-intrusive detectors, and detection for non-motorized road users.
  The **in-pavement detectors** are buried under the road. The most common are the detector loops which basically functions as a metal detector. These are used to allow the controller to know if there is any car waiting for the green light. So in case that there isn't any car waiting for green light it will continue allowing the traffic flow in other direction. The **non-intrusive detectors** are sensors or devices located over the road which sometimes is more cost effective than the under road sensors. These type of sensors includes technologies such as video image processors, sensors that use electromagnetic waves, or acoustic sensors to detect the presence of vehicles at the intersection waiting for right of way. These over-roadway sensors are more favorable than in-roadway sensors because they are immune to the natural degradation associated with paved right-of-way, competitively priced to install in terms of monetary and labor cost and danger to installation personnel, and have the capacity to act as real-time traffic management devices. They also act as multi-lane detectors, and collect data types not available from in-roadway sensors. **Non-motorized user detection** are used to detect users such as pedestrians, bicyclists and

equestrians. The detection is normally is done thru buttons and tuned detectors. When the button is pressed this is informed to the controller and it will activate a timing system which once certain time elapse, the system will allow the pedestrian to cross the street. Some traffic lights includes displays with messages indicating when the pedestrian can cross with count down counters, and some recent systems includes signals for specific group of people like audible signals for visually impaired users.

It is desired that traffic signals become part of a coordinated system so that drivers encounter a green wave (i.e. long string of green lights which technically is referred as progression). In [6] is presented the difference between a coordinated system and a synchronized system. Synchronized signals change all at the same time while coordinated (progressed) systems are controlled by a master controller and the lights are set up in "cascade" (progress) sequence so a group of vehicles can advance through a continuous series of green lights. A graphical representation of phase state on a two-axis plane of distance versus time clearly shows a "green band" that has been established based on signalized intersection spacing and expected vehicle speeds. In some countries this "green band" is used to limit speeds in certain areas. Lights are timed in such a way that motorists can drive through, without stopping if their speed is lower than a given speed limit, mostly 50 km/h (30 mph) in urban areas. Sometimes, the green wave is set on two-way streets to operate in the direction more heavily traveled, rather than trying to progress traffic in both directions.
With new technology, the coordinated signal systems facilitate drivers to travel long distances without seeing any red light. This coordination can be stablish easily on one-way streets with constant levels of traffic. In more complex systems the coordination is arranged to satisfy needs at rush hours in the heavier traffic density direction. The coordination can be lost easily if congestion is observed.
In order to prevent congestion, sometimes some of the traffic signals are coordinated to prevent drivers from encountering a long string of green lights. This is done to prevent accumulation of vehicles in a problematic area. Speed must be regulated in coordinated systems; if drivers travel too fast, they will arrive to a red indication and end up stopping while drivers traveling too slowly will not arrive at the next signal on time to catch the green indication. In synchronized the rule is that drivers use excessive speed to get through as many green lights as possible.

Even more sophisticated methods are been employed in these days. Traffic lights are sometimes centrally controlled by monitors or by computers to allow them to be coordinated in real time to deal with changing traffic patterns [6].

Smart Traffic Light

In [7], the Smart traffic light or Smart traffic signals are defined as a new system that combines existing technology with artificial intelligence to create traffic lights that truly think for themselves. In this system, the different devices communicate with each other and adapt to variable traffic conditions to reduce the amount of time that cars spend in idling reducing at the same time the travel times across the city. The result of these systems demonstrated that the idling time was reduced by 40% and the travel times were reduced by 26%.

In [7] it’s also presented that companies such as BMW and Siemens are involved in these systems. Their system works with the anti-idling technology that many cars are equipped with, to warn them of impending light changes. This should help cars that feature anti-idling systems to use them more

intelligently, and the information that networks receive from the cars should help them to adjust light cycling times to make them more efficient.

In [7] it's also presented that some other studies demonstrated that the IDLE time could be reduced over 28% with the introduction of smart traffic lights and that CO_2 emissions could be cut by as much as 6.5%.

The use of Smart traffic lights could be applied also to the public transport systems giving them more preference allowing them to move faster improving the efficiency of them [7].

Multi-agent Systems

In order to explain what Multi-agents systems is, [8] provides a good example. "Imagine a personal software agent engaging in electronic commerce on your behalf. Say the task of this agent is to track goods available for sale in various online venues over time, and to purchase some of them on your behalf for an attractive price. In order to be successful, your agent will need to embody your preferences for products, your budget, and in general your knowledge about the environment in which it will operate. Moreover, the agent will need to embody your knowledge of other similar agents with which it will interact (e.g., agents who might compete with it in an auction, or agents representing store owners)—including their own preferences and knowledge. A collection of such agents forms a multi-agent system".

Another example of this is presented next. This one is taken from [11]. In the next figure is described a multiagent example in which there are 2 barge operators and 2 terminal operators. Here each barge operator has certain limitations about how many containers can be loaded in the barge. Also the terminal operators requires to load certain number of containers in to any of the barges. Every planner has its own agent. This agent runs on the local server of the company it works for.

The agent gets all the relevant information from its planner to make the right decisions. The agent in turn communicates with agents of other companies to make appointments. The agents only exchange only limited information between each other, but enough to make the best decision for their planners.
For instance: based on the information the barge operator agent gets from all the terminal operators a barge has to visit, it searches for the best rotation possible and makes agreements with terminal operator agents. The way this is done is similar to the way the planner would have decided otherwise. Terminal operator agents in turn process the information they get such that they make decisions as their planner would have done otherwise. In case of an exception the planner can be consulted to indicate the decision(s) he thinks is the best [11].

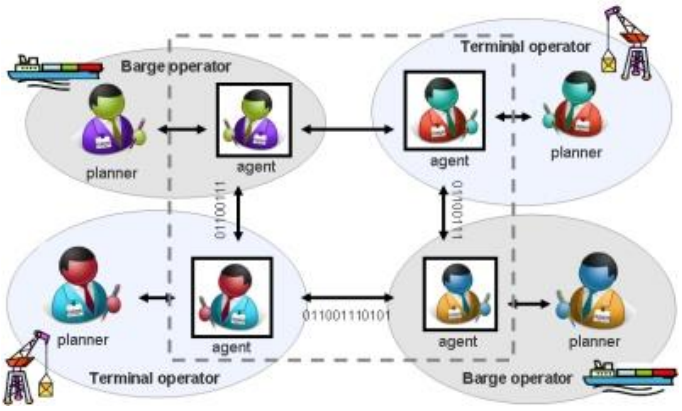


Figure 3. Multi-agent barge example.

In [9], agents are described as sophisticated computer programs that act autonomously on behalf of their users, across open and distributed environments, to solve a growing number of complex problems. This implies that the agents in a multi-agent system (MAS) require to work or interact with each other to solve problems that are beyond the individual capabilities or knowledge of each problem solver.

Advantages of a MultiAgent Approach

In [9] it's presented the advantages of a MAS over a single agent or centralized approach:

- A MAS distributes computational resources and capabilities across a network of interconnected agents. Whereas a centralized system may be plagued by resource limitations, performance bottlenecks, or critical failures, an MAS is decentralized and thus does not suffer from the "single point of failure" problem associated with centralized systems.
- A MAS allows the interconnection and interoperation of multiple existing legacy systems. By building an agent wrapper around such systems, they can be incorporated into an agent society.
- A MAS models problems in terms of autonomous interacting component-agents, which is proving to be a more natural way of representing task allocation, team planning, user preferences, open environments, and so on.
- A MAS efficiently retrieves, filters, and globally coordinates information from sources that are spatially distributed.
- A MAS provides solutions in situations where expertise is spatially and temporally distributed.
- A MAS enhances overall system performance, specifically along the dimensions of computational efficiency, reliability, extensibility, robustness, maintainability, responsiveness, flexibility, and reuse.

Applications of MultiAgent Research

In [10] is also presented some of the MAS applications covering a wide variety of domains:

- Aircraft maintenance.
- Electronic book buying coalitions.
- Military demining.
- Wireless collaboration and communications.
- Military logistics planning.

- Supply-chain management.
- Joint mission planning.
- Financial portolio management.

Characteristics of MultiAgent Environments

Also in [10] is presented the characteristics of a MAS environment which are described next:

1. MultiAgent environments provide an infrastructure specifying communication and interaction protocols.
2. MultiAgent environments are typically open and have no centralized designer.
3. MultiAgent environments contain agents that are autonomous and distributed and may be selfinterested or cooperative.

A multiAgent execution environment includes a number of concerns which are enumerated as possible characteristics in next table:

Properties	Range of values
Design Autonomy	Platform Interaction/ Protocol/ Language/ Internal Architecture
Communication Infrastructure	Shared memory (blackboard) or Message based Connected or Connectionless (email) Point to Point Multicast or Broadcast Push or Pull Synchronous or Asynchronous
Directory Service	White pages, Yellow pages
Message protocol	KQML HTTP and HTML OLE, CORBA, DSOM
Mediation Services	Ontology-basedTransactions
Security Services	Timestamps /Authentication
Remittance Services	Billing/ Currency
Operation Support	Archiving/ Redundancy /Restoration /Accounting

Table 1. Properties for a multi-agent environment

In [10] it's also mentioned that the communication protocols enable agents to exchange and understand messages. Interaction protocols enable agents to have conversations which for our purposes are structured exchanges of messages. As a concrete example of these a communication protocols might specify that the following types of messages can be exchanged between two agents:

- Propose a course of action.
- Accept a course of action.
- Reject a course of action.
- Retract a course of action.
- Disagree with a proposed course of action.
- Counter propose a course of action B.

Internet of Things (IOT)

As mentioned before, the Internet of Things (IoT) is an environment in which "Things" are provided with a unique identifier and the ability to transfer data over a network without requiring human interference. With the new technologies in wireless communications, micro-electromechanical systems (MEMS) and the Internet the IoT has evolved very fast [12].

From [12], The term of "Thing" in IoT, can be a person with a heart monitor implant, a farm animal with biochip transponder, an automobile that has built-in sensors to alert the driver about the condition of the car, or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network.

The unique identifier corresponds to the IP address, which until few years ago (IPv4) it was is a 32-bit number that identifies each sender or receiver of information that is sent in packets across the Internet. So far, the IoT has been most closely associated with machine to machine (M2M) communication and these products are often referred to as being **smart**.

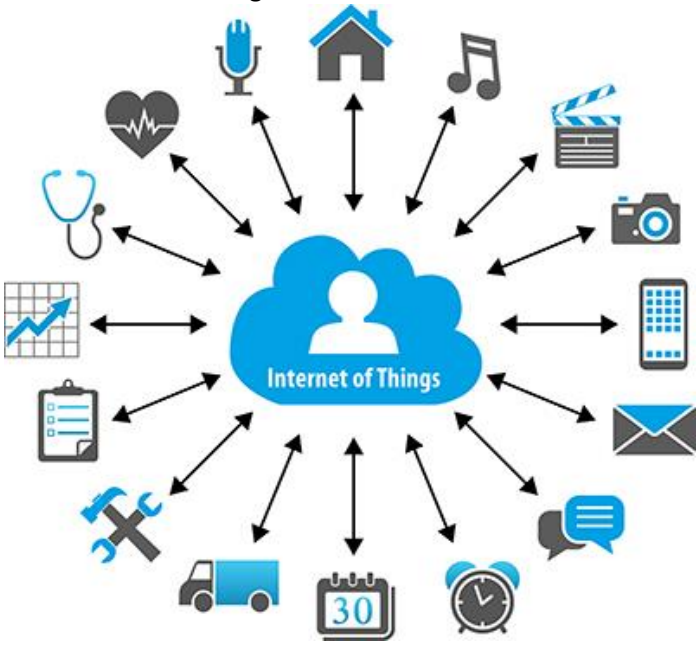


Figure 4. Internet of Things examples [15].

The **IPv6** (Internet Protocol version 6) increases the address space. This is an important factor in development of the IoT as the addresses are lengthened from 32 bits (IPv4) to 128 bits. IPv6 also supports auto-configuration and it has integrated security and mobility features.[12]

In [12] it's described the reason why the IPv6 protocol helps to IoT: "The address space expansion means that we could assign an IPV6 address in every atom on the surface of the earth, and still have enough addresses left to do another 100+ earths. In other words, humans could easily assign an IP address to everything on the planet".
Although wide number of smart devices are expected to be developed in the future, it also raises a new concern about data privacy and security. This is something that each one of the devices needs to address.

Chapter II. PROBLEM STATEMENT AND PROPOSAL

Problem Statement

As mentioned in the chapter I, the traffic congestion is one of the main problems in big cities. This problem is seen when the transport demand (vehicles such as cars, trucks, motorcycles, autobuses, etc.) exceeds the shared capacity of the transportation resources (roads, freeways, highways, etc.), making that the speed of the vehicles gets reduced drastically. The consequences of this problem affect in several ways to all the people that needs to move from one point to another in the city. For example this increases the travel time, it also increases the spending of money in fuel for the vehicles, it causes additional stress to the vehicles drivers, increases the pollution in the city (in form of contaminated air and noise); in general the traffic congestion badly affects quality of life of the city inhabitants.

It was also mentioned that there are several ways in which the traffic congestion problem can be reduced: automatic control of speed limits in the roads, creating pool lanes for vehicles with more than 1 passengers, charging a fee for using a faster road, using traffic lights systems with some degree of intelligence like vehicle or pedestrian detector; or using coordinated and synchronized traffic light systems.

In the case of the traffic light systems, today some of them are able to detect if there is any car or pedestrian needing to have access to the road. If so, it applies some intelligence to give them the right of way as soon as possible. Some other traffic light systems have the ability to be programed remotely from a central station. So they can be programmed to have certain behavior during some period of time of the day so it's possible to give more priority to some roads depending on the hour of the day. Some traffic light systems also can be programed locally or remotely to keep synchronized or coordinated (please refer to previous chapter to understand the difference) in such a way that allow that a car in certain road can get the maximum number of green lights during its course.

The vehicle congestion solutions based on traffic light systems are not as efficient as it's desired because of certain limitations:

- The synchronized and coordinated traffic light systems are limited to few roads (mainly some highly transited). Also, the systems are inefficient when the speed limits are not respected (either going faster or slower) which is a common case in big cities. Also, the synchronization or coordination is not adaptive based on the level of traffic. This is a problem because the level of traffic vary drastically along the day. The problem is increased when there are street intersections in which the left turn is required because in these cases it's required to stop the traffic for some additional time in the opposite direction while the light for the left turn is active.
- The adaptive traffic light systems that are programmed remotely normally are done based on traffic averaged observed day by day and are being adapted in an hour basis. This means that they actually doesn't monitor the traffic in real time, so they also become inefficient when the traffic load is different from these pre-calculated traffic levels.
- The control that traffic light systems provide normally collapses in case of an unexpected event such as a traffic accident or a malfunction of one of the traffic lights in a node. Also, the current traffic light systems don't support any transit priority to emergency vehicles. This is something desirable when the course of these vehicles is long (very common in big cities).

- The intelligence of the current traffic light systems is limited to detect the presence of a vehicle or pedestrian waiting for the green light. In case that there isn't any car waiting for the green light it will remain giving the green light to any of the other directions in the node. This doesn't help much to identify the real traffic load in the different roads that the traffic light system controls, and it would be desirable to measure the traffic load and based on this determine the best time for the green light in each direction.

These limitations in the current traffic light systems to remediate the vehicle congestion in cities are the motivation for the work presented here. For this, it is proposed an intelligent traffic light system based on multi agent technology using the Intel Galileo board for embedded IOT solutions. This proposal will be presented in the next section.

Proposal to mitigate the vehicle congestion problem

In this work it is proposed a multi-agent system integrated by intelligent traffic lights that are continuously monitoring the traffic load (in an emulated fashion) and use this information to decide which of the traffic lights must give the right of way to vehicles in an street intersection and for how long should do it, with the objective of making more efficient the traffic flow diminishing in this way the vehicle congestion problem.

The street intersection will be referred as a node and each one of the traffic light is also referred as an agent. Each one of the agents is connected to a LAN network thru Ethernet so they can stablish communication among them and can send and receive messages. There are several type of messages. One type of messages is used to inform to the other agents about the current traffic conditions (i.e. traffic load) in each one of the directions or streets in the intersection. Other type of messages are sent among the agents to request the green light and these messages come along with its responses to accept or reject these requests. There are other type of messages that are used to inform about the decision of which of the agents is the one who gained the right of way, or also to notify about any transition in the state of the lamps.

As the proposed solution is a multi-agent system, this also includes a container that will be used to coordinate the communication between the agents in the node, creating a directory of the agents that live in the node. Each one of the agents gets assigned a name and an IP address. This information is registered in the container so the destination agent for any message is checked and in case the destination agent is not registered in the container, then this is handled as a communication error.
In this multi-agent traffic control system one of the agents will become the agent leader which will be the agent that has more priority to have the right of way. The leader also will be in charge of coordinate the lamp state in the other agents. The leader selection is executed continuously every time that the time assigned to the current leader elapses.

The intelligence of the system resides in the ability of choosing the agent leader in agreement with the other agents in the node based on the traffic load that each one the agents reports; and in determining dynamically the time in each lamp state for each one the traffic lights in the node. This time is calculated by the leader agent using the traffic loads reported by the other agents. Also the intelligent system is able to handle emergency situations such as vehicle traffic overload and external control thru messages from the container or any other external agent.

The proposed system is easy configurable to support any number of lamps in the traffic lights (only limited by the hardware) and also to define the state of each one of the lamps in certain moment (on/off/blink).

The state of the lamps are defined in a transition table and this state is referred as a step in the table. Each one of the agents has its own transition table and depending on the step defined, the agent will make sure of setting accordingly the corresponding lamp whenever the leader agent sends a step transition notification message. As all the agents receive the step transition message, all of them are in the same step of its respective transition table.

The next figure describes the system architecture in a generic traffic control topology scheme for the system proposed in this work. Here, it can be seen that each traffic light in a street intersection or node is referred as an agent. It also shows that each agent is implemented over the Intel Galileo board. Also it shows how the agents are interconnected with the container over a LAN network.

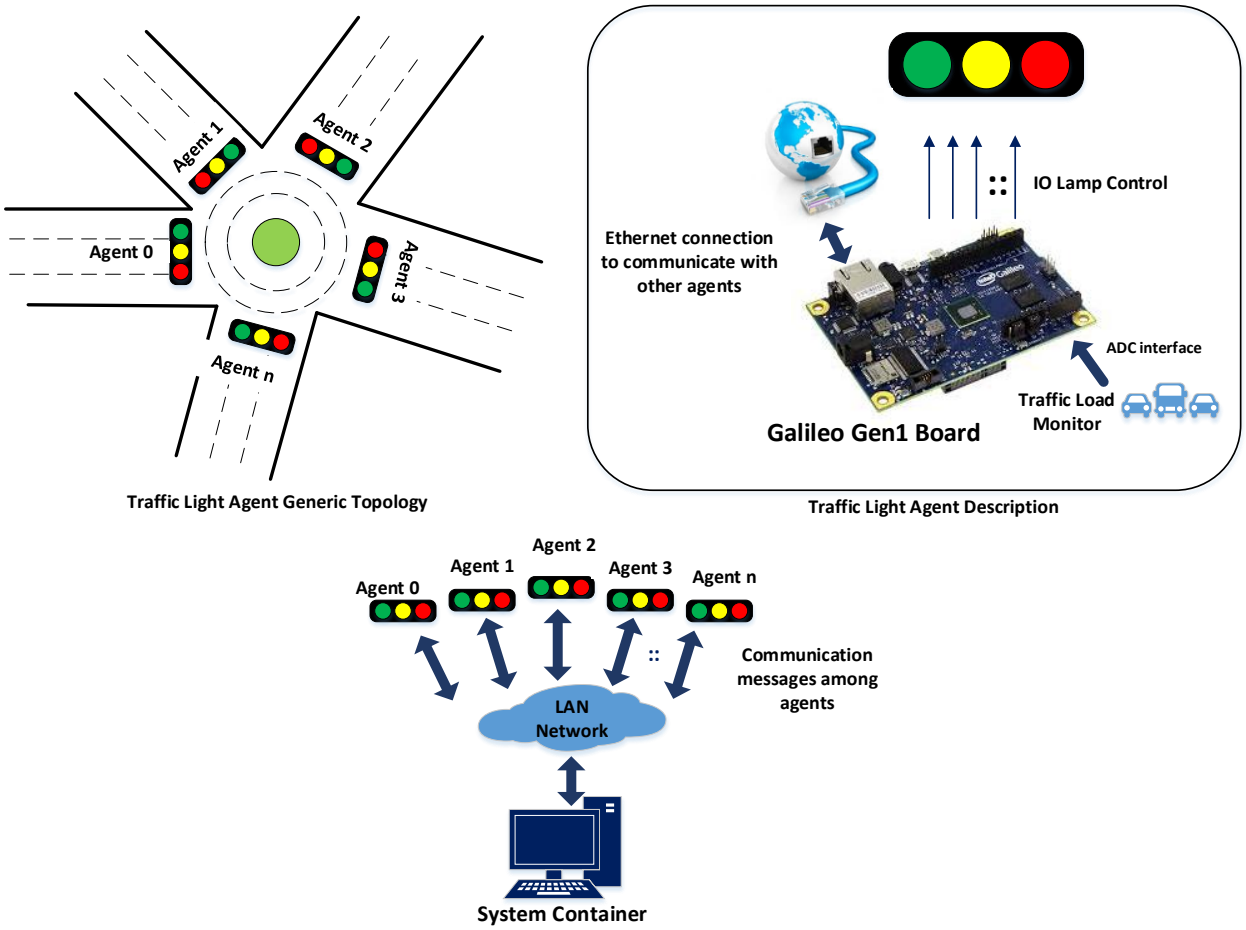


Figure 5. System architecture proposed of the traffic light system controller.

The main features of this multi-agent traffic control system are described next:

- The system can be easily adapted to support any number of agents in any possible topology. The number of lamps to control per-agent can also be adapted easily and only are limited by the number of GPIO pins in the development board. For testing purpose the topology selected is a 2-

way 2 street intersection with 4 traffic lights with support for left turn. The lamps in the traffic light are controlled using the digital pins in the GPIO port in the Galileo board (see figure 5).

- Each agent is developed over the Intel embedded Galileo platform which supports Ethernet communication and operates with a Linux distribution as an Operative System (Yocto). The agents are implemented using JADE, a Java middleware which provides resources to implement objects with some specific behaviors special for multi-agent systems. This library also provides mechanisms to stablish communication between the agents. In order to support Java it is necessary to count with a Java virtual machine which is fully supported by the Linux distribution that runs over the Galileo board.
- The system includes a transition table for each one of the agents in which is set a pre-defined sequence of the state of the lamps in the traffic light controller. Each one of the lines in the table is called a step. The transition tables of each agents are must be synchronized, meaning that each step in a transition table is synchronized with the same step number in the tables for the other agents. So, it is important that all the agents remain synchronized in the same step to avoid any possible conflict. In the transition table for each one of the agents, it is defined the steps in which this agent has priority of right to way (i.e. leadership). In the next table is shown a template of a transition table for **n** agents. In this table can be seen that it's defined **k** steps and the state of each one of the **M** lamps for every agent. The possible states for the lamps could be **ON/OFF/BLINK**. For this project the step 0 is the initial and default step and it was defined that the lamp corresponding to the RED indicator will be in BLINK state (the other lamps are off). In this table, also it is described the steps for each one of the agents when they have the leadership. This is shown in the table with the blue strips. It is possible that these blue strips get overlapped. This doesn't mean that 2 agents can be leader at the same time because this is not allowed. The range indicated by the blue strip actually must be seen as the step in which leadership of each agent should end, while the step in which the leadership for an agent begins will depend on the current step at the point of the leadership transition. Let's say for example that the leadership for Agent0 is from step 1 to 4 and for Agent1 is from 3 to 6. If Agent0 has the current leadership and the next leader is Agent1, then Agent0 will be the leader from step 1 to 4, and the leadership for Agent1 will be from step 5 to 6.

Agent	**Agent 0**					**Agent 1**					…	**Agent n**
Step	**Lamp1**	**Lamp2**	**…**	**LampM**	**Leader**	**Lamp1**	**Lamp2**	**…**	**LampM**	**Leader**	**…**	**…**
0	State	State	…	State		State	State		State		…	…
1	State	State	…	State		State	State		State		…	…
2	State	State	…	State		State	State		State		…	…
:	:	:	:	:		:	:		:		…	…
k-1	State	State	:	State		State	State		State		…	…
k	State	State	…	State		State	State		State		…	…

Table 2. Template for transition table for the agents in the traffic control system proposed

- Each agent monitors periodically the traffic load in the street that it must control. For testing purposes, the traffic load is emulated using a potentiometer generating a controllable voltage which is sampled using an analog to digital converter embedded in the Galileo board (see figure 5).
- The system is able to support the selection of the agent that receives the right of way based on the traffic load information or in a pre-defined way.

- The agents exchange traffic load information and each one requests the right of way or leadership. If the system is configured to select the next leader based on the traffic load, the agent with more traffic load is accepted by the other agents while the others are rejected. The agent accepted is called the agent leader.
- In the case of the system is configured to select the next leader in a pre-defined way, the next agent leader is selected based on the sequence pre-defined in the transition table.
- The agent leader uses the information in the transition table to know the transition steps while the agent has the control.
- Based on the traffic load informed by each one of the agents, the agent leader calculates the time for each one of the steps and the total leadership time. The leader sends Leadership notification to the other agents along with the total time it will have the leadership.
- The leader will switch to a new step accordingly to the transition table when the step time elapses and will send the step notification to the other agents asking them to switch as well.
- After completing the leadership time, the agents will transmit the respective new traffic load and a new leader is selected. Each agent can be the leader only once in a full cycle.
- The system supports emergency overload priority in which if an agent detects that the traffic load is higher than certain threshold this agent will gain the leadership independently of the case this was already selected in the current cycle. This is done in this way trying to reduce the traffic load in the road that has traffic overload.
- The system supports the ability of being controlled externally to transit to any of the possible steps in the transition table. This external control is done through messages that are sent by an external agent which could be an emergency vehicle or a central office.

In the next sections it will be provide more details about the elements used for this project such as the Galileo board and the JADE Java library.

Galileo Gen 1 Board

From [16] it was extracted the description of the Galileo board. This is a microcontroller board based on the Intel Quark SoC X1000 Application processor, a 32-bit Intel Pentium-class system on a chip. It is the first board based on Intel architecture designed to be hardware and software pin-compatible with Arduino Uno Rev3 board.
The core operating voltage of Galileo is 3.3 V, however a jumper in the board enables voltage translation at either 3.3 V to 5 V at the I/O pins. So the board has the flexibility of handle IO devices that work at any of these 2 voltage levels.

The Galileo's compatibility with Arduino Uno Rev3 board includes the following features:

- 14 digital input/output pins. 6 of them can be used as Pulse Width Modulation (PWM) outputs. As mentioned before the pins operate at 3.3 v or 5 v. Each pin can source a max of 10 mA or sink a maximum of 25 mA and has an internal pull-up resistor (disconnected by default) of 5.6 K to 10 K.
- 6 analog inputs, via an AD7289 analog to digital (A/D) converter. Each analog input provides 12 bits of resolution. By default measures from ground to 5 volts.
- I2C bus, TWI using SDA and SCL pins.

- SPI at default 4 MHz, programmable up to 25 MHz. The SPI controller in the Galileo board act as a master and not as an SPI slave.
- UART with programmable speed.
- ICSP (SPI) a 6 pin in-circuit serial programming (ICSP) header. These pins support SPI communication using SPI library.

Besides the Arduino compatibility, the Galileo board has several PC industry standard I/O ports and features to expand native usage and capabilities beyond the Arduino shield ecosystem:

- Full sized mini-PCI Express slot with PCIe 2.0 compliant features. Works with half mini-PCIe cards with optional converter plate. Provides USB 2.0 Host Port at mini-PCIe connector.
- 10/100 Mb Ethernet port
- Micro-SD slot
- RS-232 serial port
- USB 2.0 Host port to support up to 128 USB end point devices
- USB Client port used for programming. Also is a fully compliant USB 2.0 Device controller.
- 8 Mbyte NOR flash

Some additional characteristics of the Galieo board are described next:

- 400 MHz 32 bit Intel Pentium instruction set architecture (ISA) compatible processor. It has 16 Kbyte L1 cache. 512 KB of on-die embedded SRAM. Single thread, single core, constant speed. ACPI compatible CPU sleep states supported. Includes an integrated Real Time Clock (RTC) with an optional 3 V coin cell battery for operation between turn on cycles.
- 10-pin Standard JTAG header for debugging.
- Reboot button to reboot the processor.
- As storage options it counts with these:
    - 8 Mbyte Legacy SPI Flash whose main purpose is to store the firmware (or bootloader) and the latest Arduino sketch. Between 256 Kbyte and 512 Kbyte is dedicated for sketch storage. The upload happens automatically from the development PC, so no action is required unless there is an upgrade that is being added to the firmware.
    - 512 Kbyte embedded SRAM that is enabled by the firmware by default.
    - 256 Mbyte DRAM, enabled by the firmware by default.
    - Optional micro SD card offers up to 32 GByte of storage.
    - USB storage works with any USB 2.0 compatible drive.
    - 11 Kbyte EEPROM can be programmed via the EEPROM library.

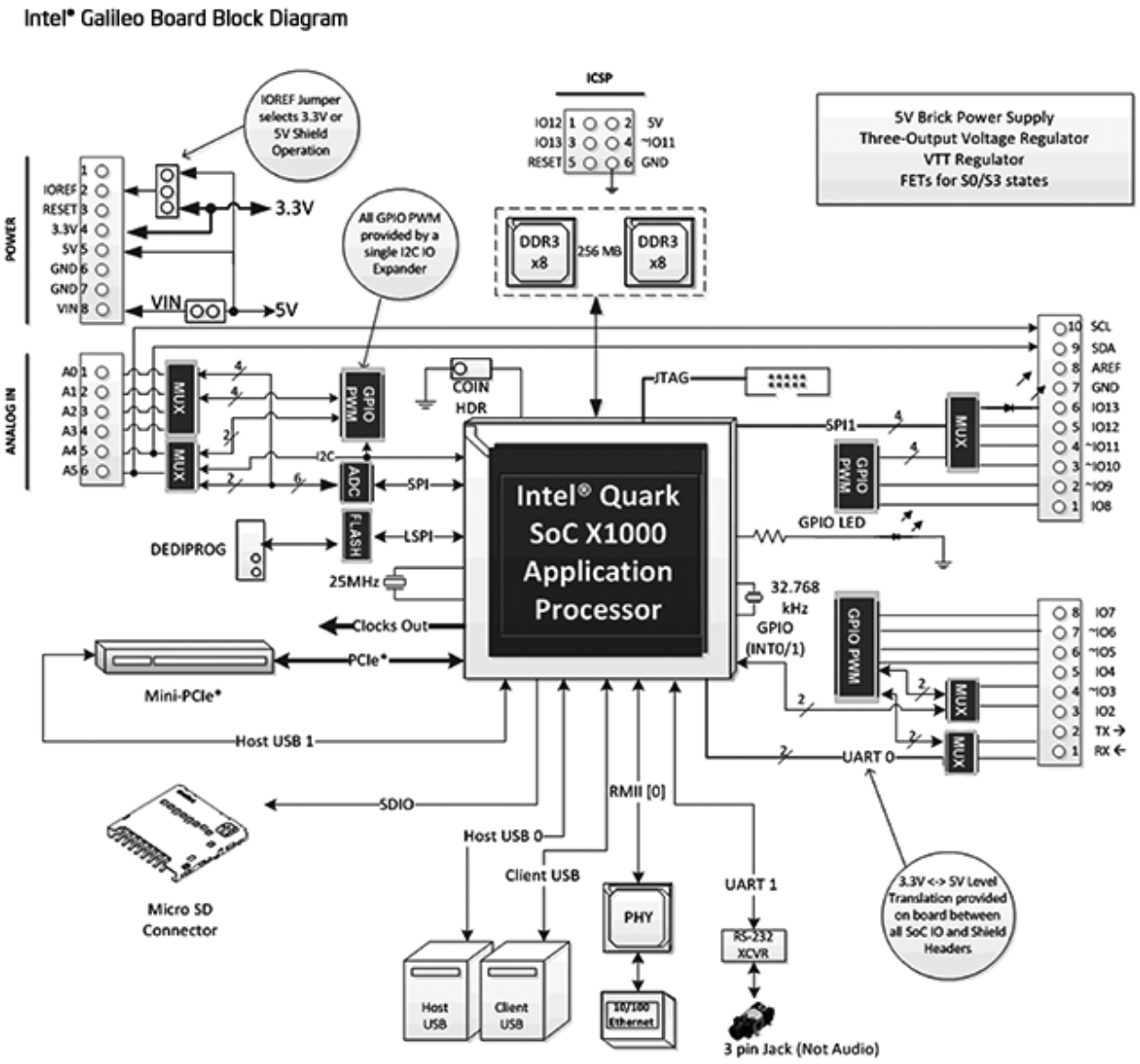


Figure 6. Galileo Board block diagram [16].

When Intel released the Galileo board in 2013, it was the first platform that was able to run an x86 de full Linux distribution and having Linux in this board adds an entire and complex layer that opens a new window to makers to create a wide set of applications. In this context Galileo is designed to run two separate Linux versions. The first is a tiny 8 MB image that sits in flash memory (also referred to as the SPI-Flash version), containing the essentials to run the Arduino side of things. This version is updated/released from time to time uploaded in to the board using the Arduino IDE. The second Linux version can be stored in the SD card. This SD card image, like the tiny version, is based on a community developed (and Intel funded) framework for building Linux distributions, which is **Yocto**. The default full image available on the Intel site is called the LSB, Linux Standard Build. This community has developed libraries to support different kind of drivers such as GPIO or USB and PCIe peripherals [17]. Also adapted

software to run in this board like Node.JS or Java which requires a Java Virtual Machine that run over the board.

In this project it was downloaded a **Yocto** distribution and stored in a SD card. Once that the board is powered up, the SPI firmware in the board loads automatically from the SD card this Linux OS. For the specific application created, it was necessary to install a **MRAA** library used to handle the GPIOs in the board. This library allow to configure the IO ports and control them using a file system structure.
Also in order to run **JAVA** it was necessary to install a **JAVA** virtual machine and finally it was necessary to install the **JADE** library.

JADE – Java Agent DEvelopment framework

In [18] it's descried that JADE is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that claims to comply with the FIPA (see [20] for more details) specifications and through a set of tools that supports the debugging and deployment phase. The agent platform can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI feature that makes it very useful for IoT applications. The configuration can even be changed at run-time by creating new agents and moving agents from one machine to another when required. The only system requirement is the Java Run Time version 5 or later.

The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL (Agent Communication Language) messages, private to each agent. Agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model has been implemented and its components have been clearly distinguished and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol. Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behavior of each state of the protocol. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.
JADE is being used by a number of companies and academic groups, such as BT, Telefonica, CNET, NHK, Imperial College, IRST, KPN, University of Helsinki, INRIA, ATOS, and many others [18].

From [19] it was seen that Jade basically includes these 3 things:

- A run time environment where JADE agents can live and that must be active on a given host before one or more agents can be executed on that host.
- A library of classes that programmers can or have to use to develop their agents.
- A suit of graphical tools that allows administrating and monitoring the activity of running agents.

Jade Containers and Platforms

In [19] is described that each running instance of the JADE runtime environment is called a **'Container'** as it can contain several agents. The set of active containers called a **'Platform'**. A single special **'Main**

Container' must always be active in a platform and all other containers register with it as soon as they start. The rule is that the first container to start in a platform must be the **'Main Container'** while all other containers must be regular containers and must be told where to find (host and port) their main container.

If another main container is started somewhere in the network it constitutes a different platform to which the regular containers can possible register. The next figure illustrates the concepts described above through a sample scenario showing 2 JADE platforms composed of 3 and 1 container respectively. JADE agents are identified by a unique name and, assuming they know each other's name, they can communicate transparently regardless of their actual location: same container (i.e. agents A2 and A3 in the figure), different containers in the same platform (i.e. A1 and A2) or different platforms (i.e. A4 and A5).

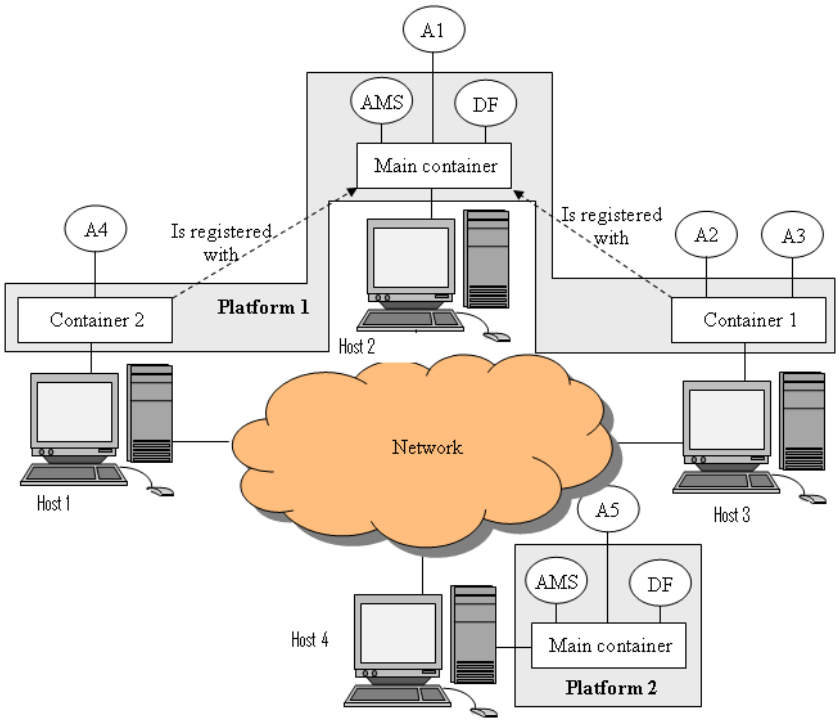


Figure 7. Description of Containers and Platform in JADE [19].

AMS and DF

Also in [19] it's indicated that besides the ability of accepting registration from other containers, a main container differs from normal containers as it holds two special agents (automatically started when the main container is launched).

The AMS (Agent Management System) that provides de naming service (i.e. ensures that each agent in the platform has a unique name) and represents the authority in the platform (for instance it is possible to create/kill agents on remote containers by requesting that to the AMS).

The DF (Directory Facilitator) that provides a Yellow Pages service by means of which an agent can find other agents providing services it requires in order to achieve its goals. A yellow page service allow agents

to publish one or more services (actions that the agent can execute) they provide so other agents can find and successfully exploit them as described in the next figure [19].

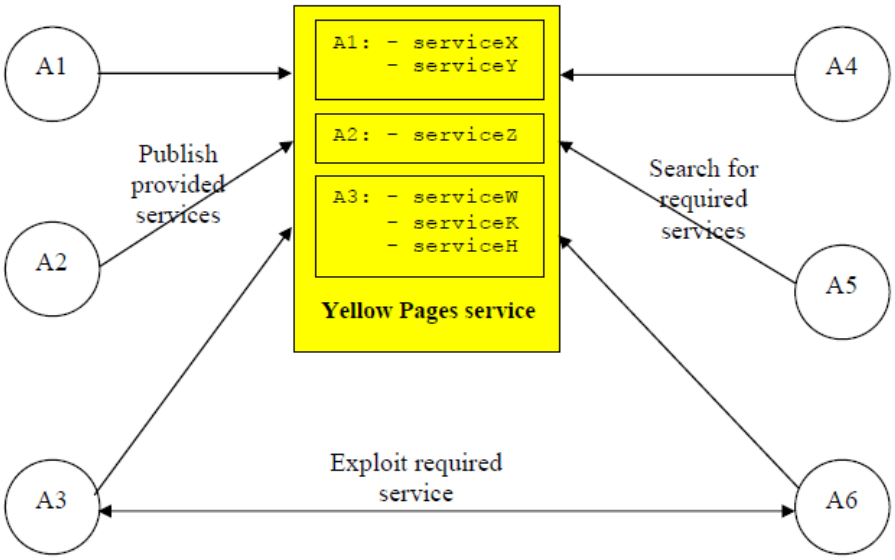


Figure 8. Yellow page service at DF.

Agent Communication with JADE

A fundamental characteristic of multi-agent systems is that individual agents communicate and interact. This is accomplished through the exchange of messages and, to understand each other, it is crucial that agents agree on the format and semantics of these messages. Jade follows FIPA standards so that ideally Jade agents could interact with agents written in other languages and running on other platforms. There are many auxiliary parts to a message in addition to the content, for example: the intended recipients, the sender and the message type. It is essential for the message as a whole to respect a common format. In JADE, messages adhere strictly to the ACL (Agent Communication Language) standard which allows several possibilities for the encoding of the actual content. In particular, Jade supports FIPA's SL (Semantic Language), a LISP-like encoding of concepts, actions and predicates. It also allows the content to be serialized Java objects. For simple application, it is often easiest to treat the content as simply a String whose meaning is application dependent, although in this project this is not the case.

The structure of an ACL JADE message includes these parameters:

- **Performative** - FIPA message *type* (INFORM, QUERY, PROPOSE, ...)
- **Addressing**
  - **Receiver**
  - **Sender** (initialized automatically)
- **Content** - This is the main content of the message
- **ConversationID** - Used to link messages in same conversation
- **Language** - Specifies which language is used in the content
- **Ontology** - Specifies which ontology is used in the content
- **Protocol** - Specifies the protocol
- **ReplyWith** - Another field to help distinguish answers

- **InReplyTo** - Sender uses to help distinguish answers
- **ReplyBy** - Used to set a time limit on an answer

JADE provides methods to send and receive these messages fairly easy independently of the protocol that his being used at lower software/hardware layers [21].

JADE Behaviours and Parallelism

It is in the nature of agents to operate independently and to execute in parallel with other agents. The obvious way to implement this is to assign a java Thread to each agent - and this is what is done in Jade.
However, there is often the need for further parallelism within each agent because an agent may be involved in negotiations with other agents and each negotiation should proceed at its own pace. We could use additional Threads to handle each concurrent agent activity but this becomes very inefficient because Java Threads (in spite of the light-weight connotation of the name) were not designed for large-scale parallelism. Rather, they were designed to allow Java programs to exploit the real parallelism of multi-processor architectures and, in current Java releases, each Java Threads requires one OS Thread. This means that passing control from one Thread to another, is about 100 times slower than simply calling a method.

In order to support efficiently parallel activities within an agent, Jade has introduced a concept called Behaviour.
A behaviour is basically an Event Handler, a method which describes how an agent reacts to an event. Formally, an event is a relevant change of state; in practical terms, this means: reception of a message or a Timer interrupt. In Jade, Behaviours are classes and the Event Handler code is placed in a method called **action**. Behaviour actions are methods, executed one after the other by the agent's Thread after events. Like listeners in graphic interfaces, they cannot pause without blocking all other activity [within the agent]. So this is the important thing to remember about Behaviours is that: "**Each Behaviour execution corresponds to ONE SINGLE instantaneous active phase**" [21].
There are several type of behaviours, the most commons are listed next [18]:

- **SimpleBehaviour.** An atomic behaviour.
- **OneShotBehaviour.** Atomic behaviour that executes just once.
- **TickerBehaviour.** This abstract class implements a Behaviour that periodically executes a user-defined piece of code.
- **WakerBehaviour.** This abstract class implements a OneShot task that must be executed only one just after a given timeout is elapsed.
- **Cycle Behaviour.** Atomic behaviour that must be executed forever.
- **FSMBehaviour.** Composite behaviour with Finite State Machine based children scheduling.
- **ParallelBehaviour.** Composite behaviour with concurrent children scheduling.
- **SerialBehaviour.** Base class for all composite behaviour whose children run serially, i.e. the composite behaviour is blocked if and only if its current child is blocked.

Each one of the behaviours has its own set of methods. Some of these methods are common among them. The most important methods are described next:

- **onStart().**This method is just an empty placeholders for subclasses. It is executed just once before starting behaviour execution. Therefore, it acts as a prolog to the task represented by this Behaviour.
- **reset().**Restores behaviour initial state. This method must be implemented by concrete subclasses in such a way that calling reset() on a behaviour object is equivalent to destroying it and recreating it back.
- **onEnd().** This method is just an empty placeholder for subclasses. It is invoked just once after this behaviour has ended. Therefore, it acts as an epilog for the task represented by this Behaviour.
- **done().**Check if this behaviour is done. The agent scheduler calls this method to see whether a Behaviour still need to be run or it has completed its task.
- **action().**Runs the behaviour. This abstract method must be implemented by Behaviour subclasses to perform ordinary behaviour duty.

The next chapter will provide implementation details on how the mult-agent traffic light controller was planned. There, it will be shown that the project contains some of the behaviours described here which will be used to implement the complete functionality of the traffic controller, including the communication with other agents.

Chapter III. PROJECT SPECIFICATION AND IMPLEMENTATION

Topology Proposed

The traffic controller system proposed for this thesis work is based on a multi-agent system which allows the interaction of the traffic control agents (traffic lights) sharing information and deciding intelligently which of the agents will have the right of way and for how long, in such a way the traffic flow becomes more fluent in the street intersection where the agents are located.

The traffic control agents are interconnected in a LAN network and the interaction among them is performed through messages informing about the current traffic load in each one of the streets in the intersection; making request to own the right of way and accepting or rejecting the requests.

The implementation of each one of the traffic light controllers is done using the Intel board Galileo, which supports Ethernet communication to get connected to the LAN network. For this project, this board requires to run a Linux distribution called **Yocto** as Operative System which allows to run Java. For this, installing a Java virtual machine is necessary. Java is selected as development language because the multi-agent system is implemented in Jade which is a Java middleware or library specific for multi-agents applications. The project takes advantage of Jade as this allows to implement easily objects with the behaviors required in to model traffic light agents including the communication among them.

For this work it is proposed a typical 4-way topology with left-turn support as the multi-agent traffic control system. This topology is perhaps the most common intersection in the cities with wide and long streets. For this topology it is needed 4 agents to control the traffic flow as shown in figure 10. Each agent will allow or block the flow of vehicles in the forward or left direction needing 4 control lamps for this: green, yellow, red and green left-turn. The description of each one of the lamps is shown next:

- **Red**: When this lamp is on, it indicates that the flow of vehicles is blocked and the cars must stop. When this lamp is blinking it means that the traffic light is in an initialization stage and the vehicle drivers must be very cautious when crossing the intersection (in any direction) as is possible that the other traffic lights in the intersection are in the same state. This lamp is controlled by the digital I/O port **D2** in the Galileo board.
- **Yellow:** When this lamp blinks, it indicates that the vehicle driver must be cautious when crossing the street intersection in the forward direction. This lamp starts blinking few seconds before transiting to red. This lamp is controlled by the digital I/O port **D4** in the Galileo board.
- **Green:** When this lamp is on, it indicates that the flow of vehicles in the forward direction is allowed. Note that left-turn is not controlled by this lamp. This lamp is controlled by the digital I/O port **D7** in the Galileo board.
- **Left-turn:** When this lamp in on, it indicates that the left-turn for the vehicles is allowed. When this lamp blinks it indicates that the time in which the left-turn is allowed is close to elapse, so cars drivers need to be cautious when crossing. When this lamp is off, the left turn must be avoided. This lamp is controlled by the digital I/O port **D8** in the Galileo board.

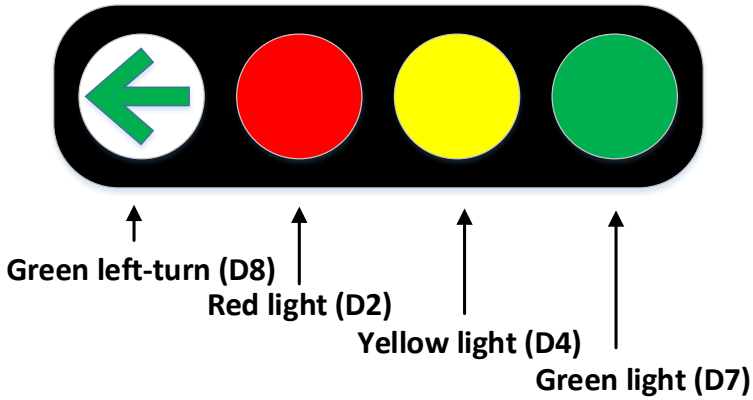


Figure 9. Lamp description in the traffic light agent and the I/O pin assigned in the Galileo board.

The traffic controller topology for the crossing street proposed is described in figure 10. It is a 4-ways street intersection and it is described next:

- **South-North road.** The direction of the vehicles goes from South to North and the left-turn allows the cars get incorporated into the **East-West** road. This road is controlled by **Agent 0**.
- **North-South road**. The direction of the vehicles goes from North to South and the left-turn allows the cars get incorporated into the **West-East** road. This road is controlled by **Agent 1**.
- **West-East road.** The direction of the vehicles goes from West to East and the left-turn allows the cars get incorporated into the **South-North** road. This road is controlled by **Agent 2**.
- **East-West road.** The direction of the vehicles goes from South to North and the left-turn allows the cars get incorporated into the **North-South** road. This road is controlled by **Agent 3**.

In the multi-agent vehicle control system proposed, it is used the **Leadership** term. This basically is referred to the fact that in certain time, one of the agents that integrate the system will be the **Leader** of this group of agents indicating to the other agents which should be the state of their own lamps. In this system only one of the agents can be the leader and the other agents (slave agents) must obey the state transitions indicated by this. This will warranty that all the agents remain synchronized avoiding any possible car accident in the street intersection. In order to visually identify which of the agents is the current leader, it is used the auxiliary on-board led indicator that is in Galileo board. This led indicator will be on whenever the current leader running in this board has the Leadership of the system.

For the topology of the street intersection proposed (see figure 10) it is necessary to define the transition table that describes the state of each table in every one of the transition steps and also the transition sequence during the time that certain agent has the leadership. This table is also shown next.

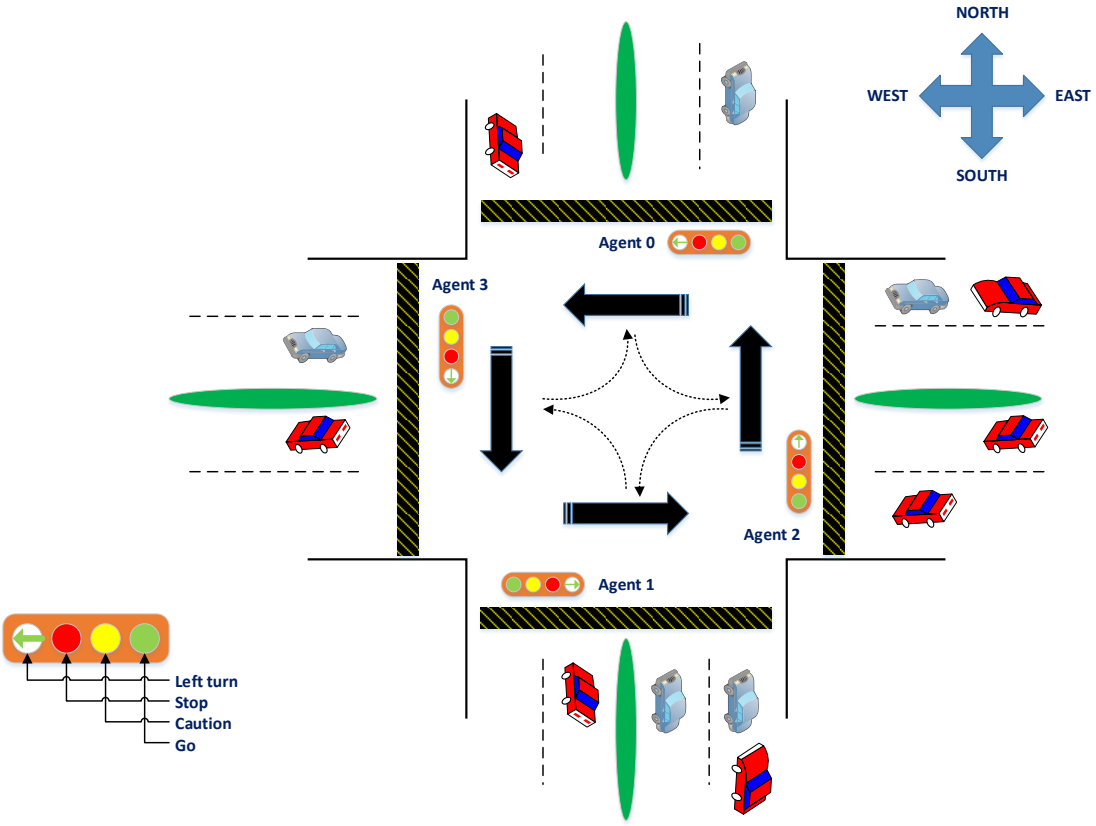


Figure 10. Traffic light system topology.

Agent	A0				LD0	A1				LD1	A2				LD2	A3				LD3
State	R	Y	G	LG		R	Y	G	LG		R	Y	G	LG		R	Y	G	LG	
S0 (Init)	B	0	0	0		B	0	0	0		B	0	0	0		B	0	0	0	
S1	0	0	1	1		1	0	0	0		1	0	0	0		1	0	0	0	
S2	0	0	1	B		1	0	0	0		1	0	0	0		1	0	0	0	
S3	0	0	1	0		0	0	1	0		1	0	0	0		1	0	0	0	
S4	0	1	0	0		0	0	1	0		1	0	0	0		1	0	0	0	
S5	1	0	0	0		0	0	1	1		1	0	0	0		1	0	0	0	
S6	1	0	0	0		0	1	0	B		1	0	0	0		1	0	0	0	
S7	1	0	0	0		1	0	0	0		0	0	1	1		1	0	0	0	
S8	1	0	0	0		1	0	0	0		0	0	1	B		1	0	0	0	
S9	1	0	0	0		1	0	0	0		0	0	1	0		0	0	1	0	
S10	1	0	0	0		1	0	0	0		0	1	0	0		0	0	1	0	
S11	1	0	0	0		1	0	0	0		1	0	0	0		0	0	1	1	
S12	1	0	0	0		1	0	0	0		1	0	0	0		0	1	0	B	

Transition table for the 4 way street intersection with left-turn support

Terminology of transition table:

Lamp	Color
R	Red
Y	Yellow
G	Green
LG	Left Green

Lamp State	Description
0	Off
1	On
B	Blink

LDx: Start and End of the leadership period for the agent x.

In the transition table for the topology proposed, it can see that it contains 13 steps, in which the first step **S0**, corresponds to the starting step after the system gets started. In this step all the agents have the red lamp blinking. In the table, also we can see the leadership steps for each one of the agents, so Agent 0 has the leadership from **S1** to **S4**, Agent 1 has the leadership from step **S3** to **S6**, Agent 1 has the leadership from **S7** to **S10** and finally Agent 3 has the leadership from step **S9** to **S12**.

The traffic controller system supports 2 mode of operations:

1. **Leadership selected based on transition table.** In this case the transition table indicates which will be the next agent to have the leadership based on the current step. The starting step is **S1** meaning that agent 0 will be the initial leader agent after a reset.

2. **Leadership selected based on traffic density (i.e. traffic load), δ_T**. In this mode, the selection of the leader is based on the traffic density that each one of the agents reports plus some additional rules that will be described later.

In both operation modes when a new leader is selected, the starting step is taken from the leadership strip of the new leader agent in the transition table. This starting step will be the lower step in the leadership strip that doesn't get overlapping with the leadership strip of the previous leader agent. This will prevent that some of the steps get repeated in 2 consecutive leadership stages.

Traffic Controller Agent Software Stack

The traffic control system implemented over the Galileo board is developed using a JAVA middleware library called Jade. Linux is the operating system that powers the Intel Galileo board. Linux allows to install a Java Development Kit which include a Java virtual machine which enables to run Java and the Jade library. Also, in order to have control over the GPIO pins in the board to control the lamps in the traffic light and to allow taking analog samples using the ADC pins (needed to read the emulated value corresponding to the δ_T), it is necessary to install the MRRA library over the Linux OS. Once having installed these components, the Traffic Light Agent Controller, developed with Java/Jade, will be able to use them to implement the desired functionality.

In order to have a more flexible and portable SW architecture for the traffic light controller, it is required that this be implemented using different software layers, as indicated in the figure 11. This software structure allows that the traffic agent controller be adapted easily to a different topology which may require different number of agents, different number of lamps or a different behavior described by the transition table.

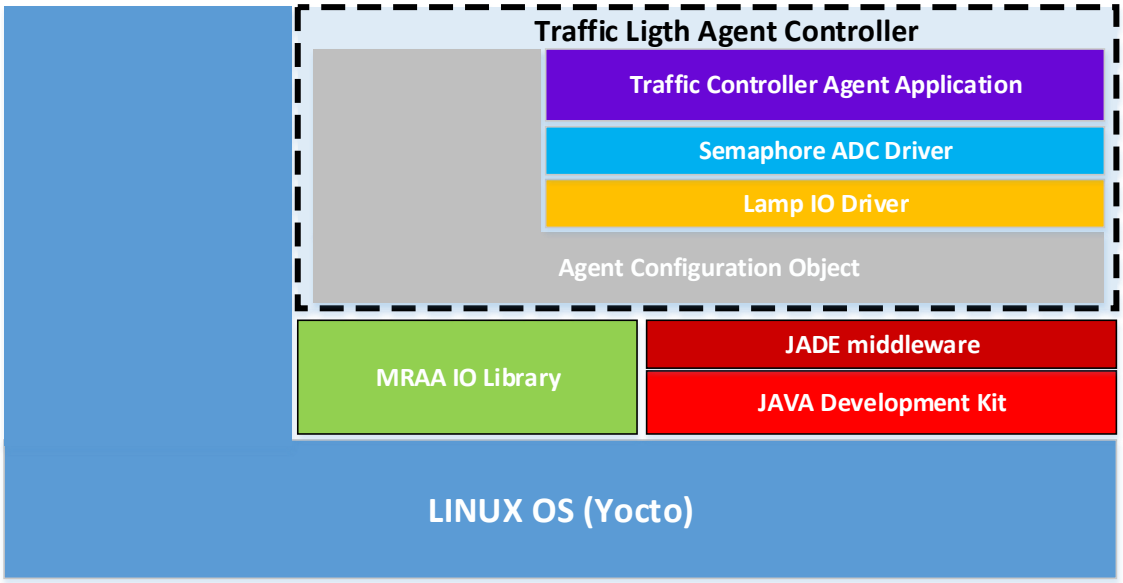


Figure 11. Software Stack for the traffic light controller

A brief description of each one of the software layers is described next:

- **Linux OS:** This is a **Yocto** distribution installed in a SD card which is run automatically once the Galileo board is powered on. This distribution has similar features like any other Linux distribution allowing to run default commands such cp, ls, pwd, grep, more, less, ssh, scp, etc. The version installed in the board is **iot-devkit-latest-mmcblkp0.direct.bz2** obtained from the Intel Developer Zone page [21]. While a base version of Linux is already built in the board, the developer kit version of Linux includes even more libraries and resources to help developers create applications in their favorite programming language. This version already includes GCC, Python, Node.js, and OpenCV, to name a few. Besides this, there are many developers how have created libraries and adapted programs to run over the Galileo board. Some of these libraries and programs were used to develop the traffic light controller agent.
- **MRAA IO library:** Libmraa is a C/C++ library with bindings to Python, Javascript and Java to interface with the I/O on Galileo, Edison & other platforms, with a structured API where port names/numbering matches the board is being used. The intent is to make it easier for developers and sensor manufacturers to map their sensors & actuators on top of supported hardware and to allow control of low level communication protocol by high level languages & constructs. This library provides a set of APIs that allow the developer to interact with all the libmraa functionality [23]. The Libmraa provide methods to support several interfaces such as:

GPIO	Gpio is the General Purpose IO interface to libmraa. Its features depend on the board type used, it can use gpiolibs (exported via a kernel module through sysfs), or memory mapped IO via a /dev/uio device or /dev/mem depending again on the board configuration.
I2C	An i2c context represents a master on an i2c bus and that context can communicate to multiple i2c slaves by configuring the address. It is considered best practice to make sure the address is correct before doing any calls on i2c, in case another application or even thread changed the address on that bus. Multiple instances of the same bus can exist.
AIO	AIO is the analog input & output interface to libmraa. It is used to read or set the voltage applied to an AIO pin.

PWM	PWM is the Pulse Width Modulation interface to libmraa. It allows the generation of a signal on a pin. Some boards may have higher or lower levels of resolution so it's necessary to make sure check this in the board is being used.
SPI	This file defines the spi interface for libmraa. A Spi object in libmraa represents a spidev device. Linux spidev devices are created per spi bus and every chip select available on that bus has another spidev 'file'.
UART	UART is the Universal asynchronous receiver/transmitter interface to libmraa. It allows the exposure of UART pins on supported boards.

In this case the libmraa libray is installed in the Galileo board and the IO and AIO pins are configured and accessed using the Linux commands such as **echo** and **cat**. When the library is installed, some directories and files related to the GPIO or ADC pins are created under the file system. These files allow to configure, control and read the corresponding value in the pin (either digital or analog).

In the case of GPIO the directory created is: **/sys/class/gpio/**
In the case of AIO the directory created is: **/sys/bus/iio/devices/iio:device0/involtageX_raw**

The commands used in this project are these:

echo pin > /sys/class/gpio/gpio/export	Create the gpio directory for the pin specified allowing to configure it. Pin is the Linux pin value no the board pin. It requires a translation. Creates: /sys/class/gpio/gpio/gpio<pin>
echo dir > /sys/class/gpio/gpio/gpio**X**/direction	Set direction of the pin X (in or out)
echo val > /sys/class/gpio/gpio/gpio**X**/value	Set the value in the pin X(0 or 1)
cat /sys/bus/iio/devices/iio:device0/involtageX_raw	Read the integer value from the ADC X pin

For the project presented here, the Linux pin needed for the board pins are shown next:

Board Pin	**Linux Pin**
Digital IO 2	GPIO32
Digital IO 4	GPIO28
Digital IO 7	GPIO27
Digital IO 8	GPIO26
Analog 0	Involtage0_raw; Requires to set GPIO37 = 0

- **Java Development Kit**. A Java Development Kit (JDK) is a program development environment for writing Java applets and applications. It consists of a runtime environment that "sits on top" of the operating system layer as well as the tools that developers need to compile, debug, and run applets and applications written in the Java language[24].

JDK contains the tools needed to develop the Java programs and the Java Run Environment (**JRE**) to run the programs. The tools include compiler (**javac.exe**), Java application launcher (**java.exe**), Applet viewer, etc. Compiler converts java code into byte code. Java application launcher opens a **JRE**, loads the class, and invokes its main method.
JDK is needed, if it's wanted to write a prohram, and to compile it. For running java programs, **JRE** is sufficient. JRE is targeted for execution of Java files i.e. **JRE** = **JVM**(Java Virtual Machine) + Java Packages Classes (like util, math, lang, awt,swing etc) + runtime libraries. **JDK** is mainly targeted for java development.

The Java Virtual Machine (**JVM**) provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run. But, note that **JVM** itself is not a platform independent. It only helps Java to be executed on the platform-independent way. When **JVM** has to interpret the byte codes to machine language, then it has to use some native or operating system specific language to interact with the system. One has to be very clear on platform independent concept. Even there are many JVMs written on Java, however they too have little bit of code specific to the operating systems [25].

Figure 12. Java Development Kit and its components [25].

As the traffic light agent controller is implemented in Java, it is required to install a JDK over the Galileo board. For this, it is required to install the JDK version **jdk-8u45-linux-i586.rpm** which can be found in the **Oracle** web page.

- **Jade middleware.** In the previous chapter it was already described JADE. The traffic light controller requires to use Jade library in order to implement the agent functionality, specially focus in multi-agent behavior such as send and receive messages among the agents. Also provides some specific type of classes that allow to implement some behaviors that may be dependent of time, that need to be repeated cyclically, or perhaps need to be modeled as a finite state machine. Finally, Jade also provide some tools that allow to monitor the communication among the agents in the system.

For the traffic light controller it is required to install the Jade version 4.3.3 which can be obtained from the Jade web page [2].

- **Agent Configuration object**. In this software layer is defined some of configurable parameters for the traffic light controller. This layer is visible by the other software layers that integrates the agent. The parameters that are defined here are for example the number of agents that will integrate the system, the transition table for all the agents. It is also defined some of the times that rules the agent behavior like the blinking period when yellow lamp is on or the relation between the total leadership time and the time assigned to the left-turn green lamp. These parameters are defined in an independent layer because in this way is easier to update them in one single file instead of having to update them when they are spread in several files. Also, the transition table allows to change the behavior of the agent in a very simple way, in case it is needed. In this layer it is also provided some functions related to the transition table such as get the transition steps for a specific leader agent or determine the leadership time for each one of the steps.

- **Lamp IO driver**. In this layer it is defined the lamps that are needed for the traffic light. It provides methods to configure the lamps and to control the state of the lamps. In order to control or configure the lamps, this layer provides some methods that send "**echo**" commands to the **Linux** shell. This layer also handles the translation between the physical board IO pin to the GPIO value that **Linux** understands (MRAA library). The methods that this layer provides allow to turn-on/off any lamp or a full set of lamps based on a step in the transition table. The advantage of this layer is that this allows upper layers to operate independently of the lamps that are defined here. This allows to have flexibility in case it's required to add or remove any lamp from the traffic light agent.

- **Semaphore ADC driver**. This software layer is in charge of handling the ADC conversion needed to emulate the traffic density in the street. This layer is supported in the Lamp IO driver because in order to configure the pin used as ADC it also requires to configure a GPIO. This layer provides methods to create the ADC pin and also a method to read an analog value. The read method will be used only by the application layer.

- **Traffic Controller Agent Application**. This layer describes the real behavior of the traffic light agent controller which includes getting the traffic load in the street it controls, sharing this information with other agents, selecting the new leader agent, getting the time for each leadership period, getting/setting the information of the current step and controlling the state of the lamps. It also is in charge of stablishing communication among the agents and maintaining the synchronization among them. This layer makes use of the classes and methods provided by the lower level layers, for example if it needs to set the lamps in certain state, this layer calls a method in the Lamp IO driver with the state desired and this last layer will be in charge of executing all the procedures required to achieve this. Similarly, if the agent requires to know which the next step to transit is, it will send this request to the agent configuration object which will get the information from the transition table and will return this information to this layer. The implementation of this software layer requires the use of the **Jade** library to implement the functionality of the traffic light controller.

More detailed information for each one of the software layers agents will be provided later when it's described the specification of each one of them.

Software Architecture Specification

This section defines the complete functionality of the intelligent traffic light controller system in terms of the software architecture that needs to be followed during the software development stage. As indicated in previous section, the software architecture is divided in layers. Following the same approach, the software specification will be described in the same layers. The information provided for each layer will include the expected functionality in the layer and the software elements that are required to implement this functionality.

AGENT CONFIGURATION OBJECT

The agent configuration object declares and defines some of the parameters that that the traffic controller agent required to execute the desired functionality. As its name indicates, these parameters are configurable depending on how the agent needs to operate.

This configuration object is defined as a **class** named **semaphore_cfg**. Inside of this class is defined the next configurable parameters:

- **number_of_agents**. Number of agents that conforms the traffic control system. In the case of the topology selected for this project this value is 4.

- **agentCfgTable**. Transition table definition. The transition table was already presented in the **Topology** section in this chapter. This transition table includes the information related to the steps transition for each one of the agents that integrates the system. For each one of the agents it is defined the leadership strip (leadership starting step and the leadership ending step) and the state of the lamps for each one of these steps. In order to define the lamps state in each step it is required to create the **LampsLightSt** class. This will be describe latter.

- **NumCfgSteps**. Number of steps in the transition table. For this topology, the number of steps is defined as 13.

- **MinStepTime**. This is the minimum step time in seconds. When the step time is calculated, the resulting time must be at least the value defined in this variable. For this project this value is defined as 3 seconds.

- **YellowTime**. This defines the step time in seconds in which the yellow lamp must be on or blinking as a warning indication. For this project this variable is defined as 3 seconds.

- **GreenBlinkTime**. This defines the step time in seconds in which the green lamp must be blinking. For this project this variable was defined by 3 seconds.

- **GreenLeftBlinkTime**. This defines the step time in seconds in which the left-green lamp must be blinking. For this project this variable was defined by 3 seconds.

- **LeftGreen2LeaderTimeRatio**. In this project the calculation of the time in which the left-green lamp will be on was defined as a percentage of the total time assigned to the leader agent. For this project, this percentage was defined as 30%.

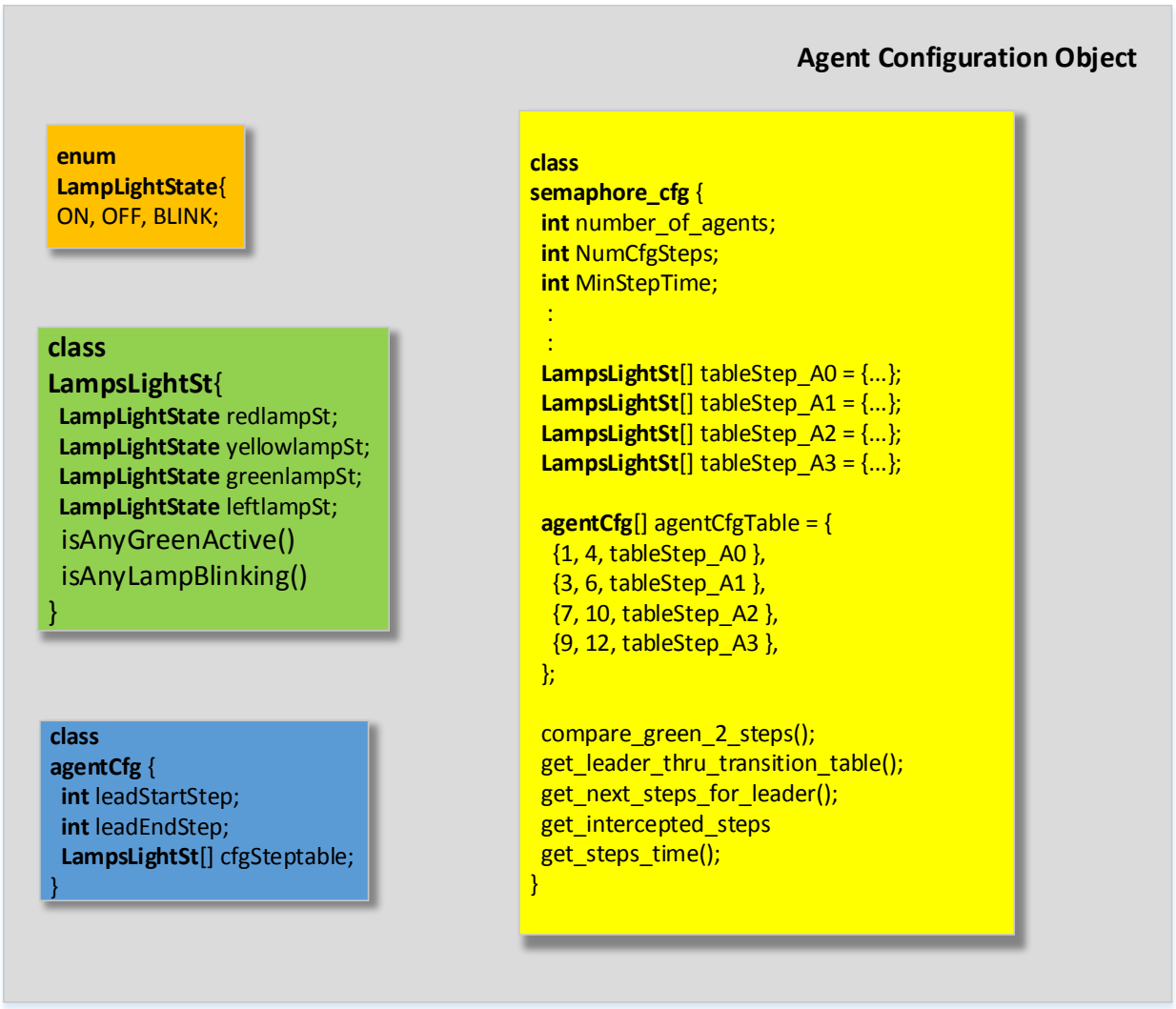


Figure 13. Agent Configuration object structure.

Before continue with the rest of the functionality of the configuration object, it is necessary to describe the **LampsLightSt** class which is used to capture the state of each one of the lamps in the traffic light controller.
For this project, this class defines these **LampsLightSate** variables:

- **redlampSt**. State of the red lamp.
- **yellowlampSt**. State of the yellow lamp.
- **greenlampSt**. State of the green lamp.
- **leftlampSt**. State of the left-green lamp.

The **LampsLightSate** type indicate the possible states of the lamps. The possible values could be any of these, which can be defined as an enumeration:

- **ON**. The lamp is on.
- **OFF**. The lamp is off.
- **BLINK**. The lamp is blinking.

Besides of the definition of the lamp state in the **LampsLightSt** class, it is also required to define some support methods:

- **isAnyGreenActive()**. Checks if in any of the green lamps (green or left-green lamp) is ON.
- **isAnyLampBlinking()**. Check if any of the lamp is blinking.

Continuing with the definition of the functionality of **semaphore_cfg** class, it is required to define some other support methods related to the transition table. Here are these methods:

- **compare_green_2_steps()**. This is a Boolean method which compares two states provided as parameters and returns true if there is any green promotion in any of the green lamps. This means that if there is any transition from OFF to ON this method will return true, otherwise it will return false. This method will be used by the agent to detect a green transition. If this is the case, the transition to the new step will be delayed to avoid any possible accident during the lamps transition.

- **get_leader_thru_transition_table()**. This method returns the next leader when the agent is in the mode in which the leader is selected from the transition table. This method receives as parameters the current step and the current agent. If the current step is 0, it will return as the next leader Agent 0, because it means that the agent is in IDLE state, so the default leader will be Agent 0. If the current step is different than 0, then the next leader will be the current leader + 1. If the current leader is the last agent, the next leader will be Agent 0 again (a modulus operation can be applied here).

- **get_next_steps_for_leader()**. This method provides a list of the remaining leadership steps for a given agent based on the current step. The agent and current step are given as parameters. For this, the function requires to obtain the maximum and minimum leadership steps for the agent (min/max_step_in_range), and check if the current step is inside of this range. If so, the function must return the remaining steps in the range (i.e. from [current_step + 1] to max_step_in_range). If the current step is not in the leadership range, then the step list to return goes from min_step_in_range to max_step_in_range. This function is used by the leader agent to know the steps in which it needs to lead.

- **get_intercepted_steps().** This function generates a list of overlapped steps between the current agent and the previous one. This function receives as parameters the current and previous leader agents. This function uses the information captured in the leadership transition table to find out which the overlapped steps are. Something important to notice about this function is that the returned overlapped steps are only those which have a green or left green lamp ON. Also, only the overlapped steps are analyzed if the previous and current agent in the parameters are different and if they are actually valid agents.

- **get_steps_time()**. This function generates a list of times in seconds. Each time in the list corresponds to each of the steps for a given agent, based on a total time. The agent, total time and list of steps are provided as parameters. In order to calculate the step times, it is required to obtain the lamps states for each one of the lamps. Having the lamps states, this function needs to take into account these constrains:
    1. Known times for steps. There are steps which a pre-defined duration that need to be assigned. This is the case of steps having yellow lamps on or any blink lamp whose times are assigned using the configurable parameter defined in this class.

    2. It's necessary to consider the case in which for a given step this agent has only the green lamp on, but in the same step other agent has any known state. In this case, the step time should be limited to the known time.

    3. The time for LeftGreen lamp when this is ON is also a known value, which is calculated in this way:

$$TotalLeftTime = \frac{totalTime * \text{LeftGreen2LeaderTimeRatio}}{100}$$

       In this case the added time for the steps that have LeftGreen lamps in ON state must add the $TotalLeftTime$ value obtained from previous equation. If there are more than one steps in this condition, then the best effort should be done to make the step times as similar as possible.

    4. The time for the steps in which the green lamp is ON and the rest are in OFF state must be calculated by the remaining time from $totalTime$ after having assigned the time for the rest of the steps. If it's the case that exist more than 1 step in this state, it should make the best effort to make the step times as similar as possible.

    5. For all the steps it needs to be warrantied that the *MinStepTime* time get satisfied. If any of the steps during its calculation result lower than this value, then it needs to be re-assigned. The re-calculation needs to be done during the execution of each one of the above points (not at the end of the function) in order to try to satisfy the $totalTime$ specified. At the end, if it's not possible to satisfy the $totalTime$ when trying to satisfy the *MinStepTime* then this is an acceptable condition.

    This function is used by the leader agent to know the time for each one the steps within its leadership period.

The methods and data structures described here will be used by the other software layers to support the implementation of their own functionality.

LAMP IO DRIVER

This driver is in charge of create the objects for each one of the lamps in the agent and provides methods to control the state of them. This library interacts with the Linux File System created when the MRAA library was installed. This interaction is done thru system calls doing read/write operations over the GPIO files associated with the corresponding lamp.

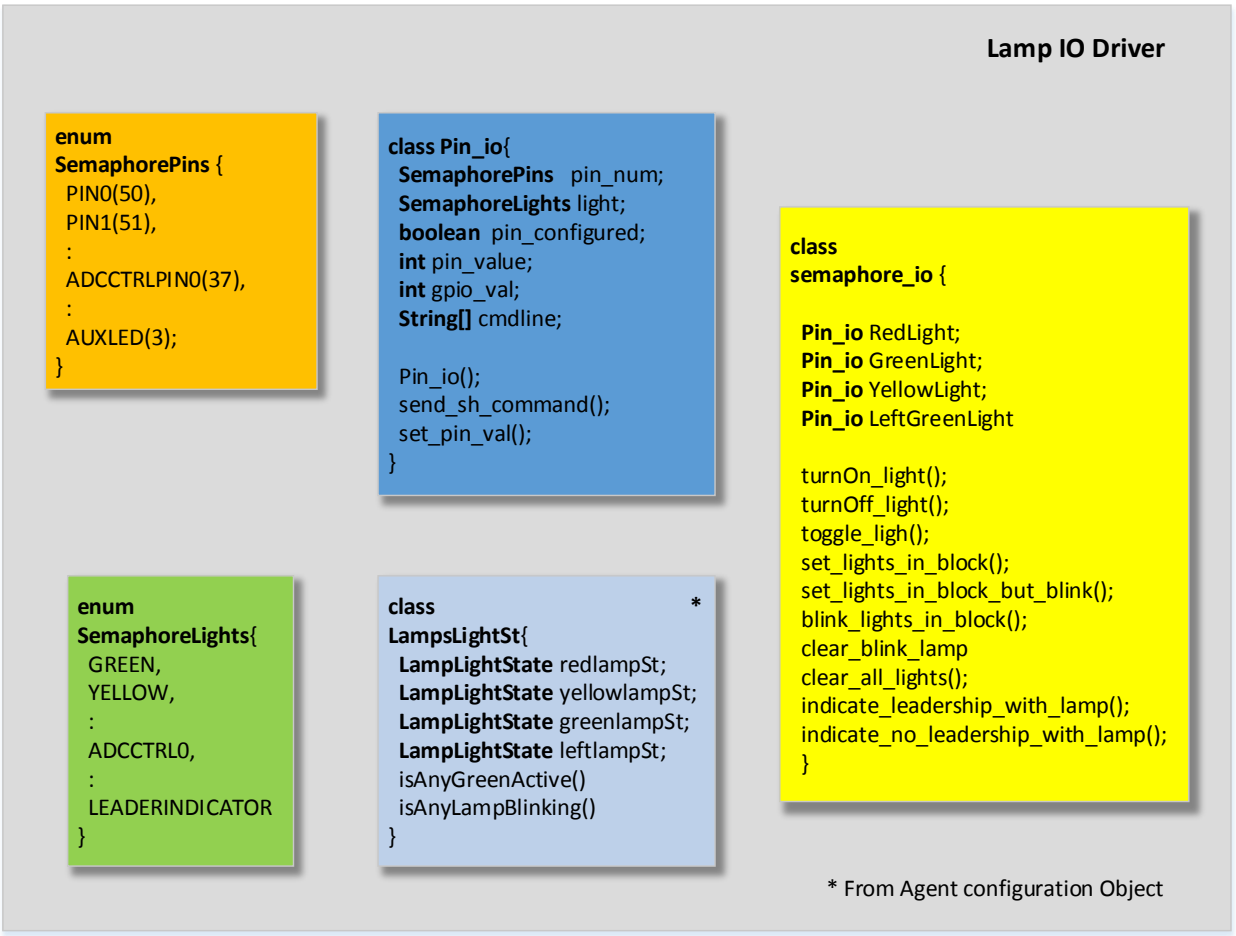


Figure 14. Structure of the Lamp IO driver software layer.

In order to map the board pins with their corresponding GPIO number (i.e. the GPIO value that Linux understands) it's necessary to create a mapping table which includes all the possible pins in the board (this is done in case additional pins/lamps are required). This can be handled easily with the enumeration **SemaphorePins**. For this it's necessary to include the values in the next table:

Pin	GPIO	Pin	GPIO	Pin	GPIO	Pin	GPIO
PIN0	50	PIN5	17	PIN10	16	ADCCTRLPIN0	37
PIN1	51	PIN6	24	PIN11	25	ADCCTRLPIN1	36
PIN2	32	PIN7	27	PIN12	38	ADCCTRLPIN2	23
PIN3	18	PIN8	26	PIN13	39	ADCCTRLPIN3	22
PIN4	28	PIN9	19	AUXLED	3		

This table also includes the auxiliary lamp on-board and the ADC pins. For the ADC pins it's required to define also a GPIO because in order to configure them it's required to set a value in the GPIO associated to them.

Also, in order to facilitate the identification of a lamp it's better to provide a name related to the lamp function or color. For this, it is also required to define the **SemaphoreLights** enumeration with these possible values, which can be extended as needed: **GREEN, YELLOW, RED, LEFT_GREEN, ADCCTRL0, ADCCTRL1, ADCCTRL2, ADCCTRL3 and LEADERINDICATOR**.

Once having defined these 2 enumerations, it's necessary to create a class that defines a lamp. This class is called **Pin_io**. The object created with this class basically will be a lamp, so this class requires to provide methods to control the lamp.

The **Pin_io** class contains mainly these elements and functions:

- **SemaphoreLights light**. This variable indicates which lamp is this object about (GREEN, LEDERINDICATOR, etc).

- **SemaphorePins pin_num**. This corresponds to the name of the board pin (PIN4, AUXLED, etc).

- **int gpio_val**. This is the value that Linux understands for the pin associated with this lamp. This is the value extracted from the **SemaphorePins** enumeration.

- **int pin_value**. This is the digital value assigned to the pin associated to the lamp to be controlled (0 or 1).

- **String[] cmdline**. As mentioned before, the board pins are controlled using the Linux file system writing in to some specific GPIO files created when the MRAA gets installed. In order to access to these files it is required to send commands to the Linux shell. As example this could be a command that may be sent: **echo** value **>** path_file. The **cmdline** variable is an array of strings which is used to store the command that want to be sent. In this array the element 0 and 1 are fixed as indicated below:

  String[] cmdline = {"sh", "-c", " "};

  These 2 initial elements are required in order to make that the command gets executed successfully. The real command to be sent needs to be inserted in the array in the position 2.

- **boolean pin_configured**. When the object for the lamp is created, the corresponding GPIO pin needs to be configured as an output. This variable will indicate **true** if the configuration was done or **false** if the configuration has not been done or failed.

- **send_sh_command()**. This function sends the Linux command stored in the **cmdline** variable. So in order to call this function this array needs to be already filled. This function returns **true** if the command was successfully sent and **false** if was detected any error. The command is sent using the Java Runtime **exec()** function. This requires to handle any possible exception.

- **set_pin_val()**. This function assigns a digital value to the pin to turn the lamp on or off. The value to be assigned is given as a parameter. This function uses the **send_sh_command()** to control the pin state. For this, the **cmdline[2**] value needs to be assigned in this way:

  cmdline[2] = “echo “ + *pin_val* + “> /sys/class/gpio/gpio" + *gpio_val* + "/value"

  Before sending the command, the corresponding pin needs to be previously configured as output. This function will return **true** if the setting succeeds, otherwise it will return **false**.

When the object for the lamp is created from **Pin_io** class, the corresponding GPIO pin needs to be configured. The configuration requires that the file structure for the corresponding pin gets created, then the pin needs to be configured as an output and finally assign a default value of 0 to the pin. The information required to initialize the object are the lamp and pin (from **SemaphoreLights** and **SemaphorePins** enumerations respectively). With this information is possible to determine the GPIO associated with this lamp. The initialization flow is shown in the next simplified diagram:

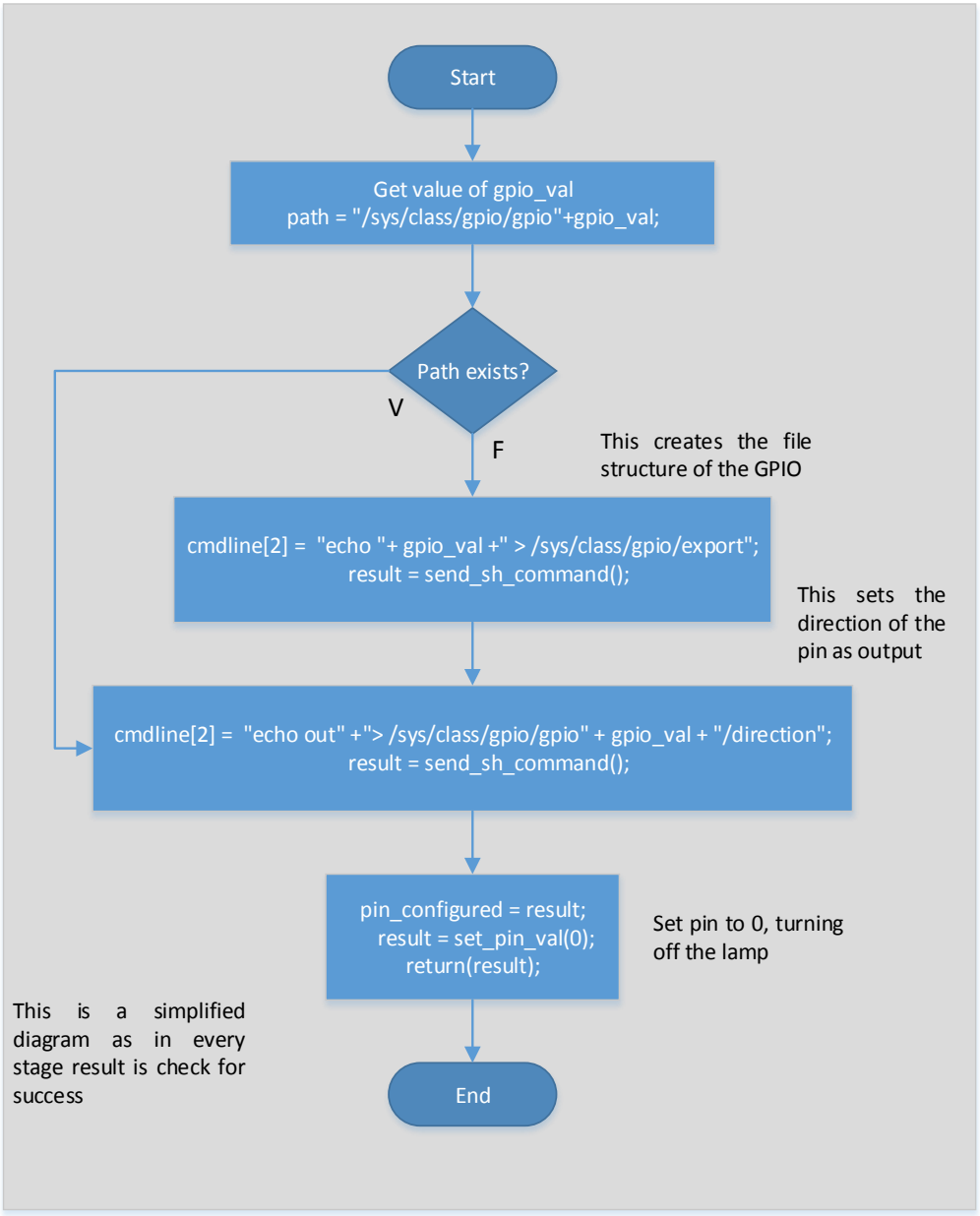


Figure 15. Initialization flow of a lamp from Pin_io class.

Finally, it is required to instantiate all the lamp objects in one single class. This is done in the **semaphore_io** class.

In this class besides of create the objects for lamps there are also provided some support methods to control the lamps. A brief description of the object and functions created in this class are shown next:

- **Pin_io RedLight**. Red lamp object in the traffic light.
- **Pin_io GreenLight**. Green lamp object in the traffic light.
- **Pin_io YellowLight**. Yellow lamp object in the traffic light.
- **Pin_io LeftGreenLight**. Left-green lamp in the traffic light.

- **Pin_io LeaderIndicatorLight**. Leader led indicator object in the board.
- **turnOn_light()**. Method that turns-on a specific lamp given as a parameter (LEFT_GREEN, RED, etc).
- **turnOff_light()**.Method that turns-off a specific lamp given as a parameter (LEFT_GREEN, RED, etc).
- **toggle_light()**.Method that toggles a specific lamp given as a parameter (LEFT_GREEN, RED, etc).
- **set_lights_in_block()**. Method that sets all lamps accordingly with the lamp state defined in the **LampsLightSt** type parameter provided. If the lamp is **ON**, then the lamp is turned-on. Otherwise the lamp is turned off. This only controls the traffic light lamps.
- **set_lights_in_block_but_blink()**. This method is similar to **set_lights_in_block()** method. The only difference is that in case any lamp is in BLINK state, this is not modified.
- **blink_lights_in_block()**. This method toggles the lamps that are in **BLINK** state. The lamp state is given in the **LampsLightSt** type parameter.
- **clear_blink_lamp()**.This method turns-off the lamps that are in **BLINK** state. The lamp state is given in the **LampsLightSt** type parameter.
- **clear_all_lights()**. This method turns-off all the lamps in the traffic light.
- **indicate_leadership_with_lamp()**. This method turns on the leadership led, indicating that the agent on the board has the leadership.
- **indicate_no_leadership_with_lamp()**. This method turns off the leadership led, indicating that the agent on the board doesn't have the leadership anymore.

An object of the **semaphore_io** class must be declared inside the class that defines the traffic light agent. From there, the agent only needs to indicate which lamp or set of lamps wants to be turned on or off without worry about the pin associated with the lamp nor the flow required to do it. This is the great advantage of using a software layered architecture.

SEMAPHORE ADC DRIVER

The semaphore ADC driver is in charge of create the object associated with the ADC pin used to capture analog samples during the emulation of the traffic density ($\boldsymbol{\delta_T}$). This object is in charge to talk with the lower software layer IO driver to configure the ADC pin and also provides a method to sample the $\boldsymbol{\delta_T}$. The ADC has a resolution of 12 bits, meaning that the sampled value could vary from 0 to 4095. This software layer uses data structures and methods from lower layers (IO driver) to configure the ADC pin. In the same way, upper software layers will use the methods provided by this object to take a sample.

The next figure shows the implementation of this software layer:

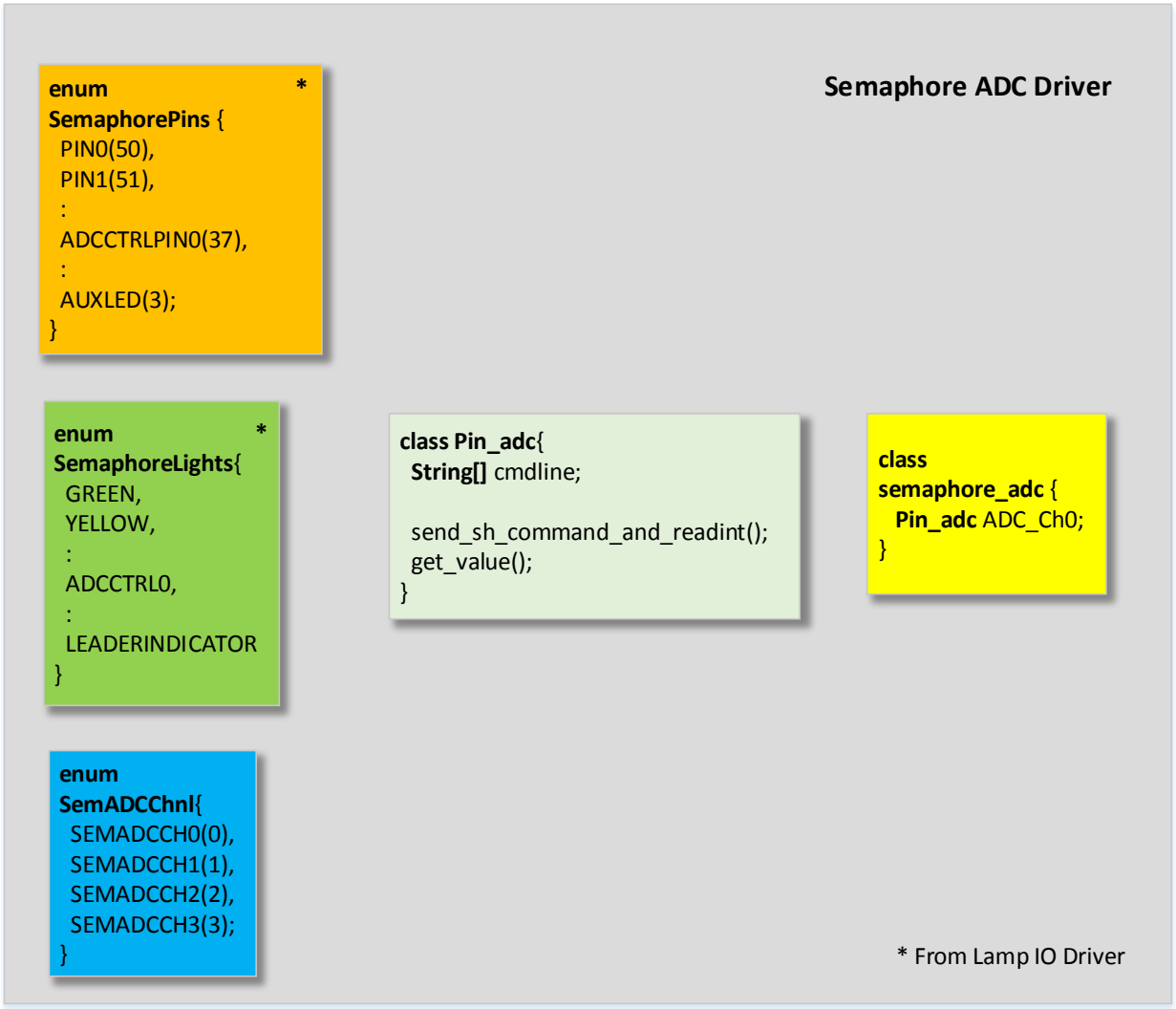


Figure 16. Structure of the Semaphore ADC Driver software layer.

The Galileo board supports 6 ADC channels (1 per pin), and in order to identify each one of them, a name and a value is assigned to each one of them. This is done using the **SemADCChnl** enumeration. The name given to each one of the channels has this format: **SEMADCCHx**, where **x** is the channel number. The number given to each one of the ADC channels in the enumeration corresponds to a specific number in the path of the file used to sample the analog value. For example for the pin 0 and pin 1, the paths used are:

/sys/bus/iio/devices/iio\:device0/in_voltage0_raw
/sys/bus/iio/devices/iio\:device0/in_voltage1_raw

In the **SemADCChnl** enumeration only ADC channels from 0 to 3 are defined. The reason of this is that the configuration from channel 4 and 5 is done different than for the first 4 channels and it is thought that 4 ADC channels is enough for this project (the configuration will be explained later).

For each one of the ADC channels is required to create an object that allow its configuration and provide the sample methods. This is achieved with the **Pin_adc** class. When an object from this class is created,

the ADC channel gets configured. The configuration of the channel is quite simple as only is required to write a 0 in to a GPIO pin associated to the ADC channel. In order to write a value to this GPIO, it is required to create a new IO pin object from the **Pin_io** class taken from the Lamp IO Driver. In order to create this object, it's necessary to provide the names associated to this ADC channel defined in the **SemaphorePins** and **SemaphoreLights** enumerations. The advantage of re-use the **Pin_io** class is that by the time the object for the GPIO is created, it is assigned a value of 0 to this which is the value required to configure the ADC channel associated to this GPIO. This is the only configuration require for the ADC channel. The ADC resolution of 12 bits is the only one supported, so this doesn't requires any configuration.

Besides of configuring this channel, the **Pin_adc** class requires to provide some support methods. The variables and methods that this class requires are described next:

- **String[]** cmdline. In order to read an ADC sample, it is also required to send a command to the Linux shell. For this, necessary to create a string array where the full command to send is stored. Similarly to the **Pin_io** class, the elements 0 and 1 are pre-defined and the element 2 is modified depending on the channel we want to access. The initial value assigned to this cmdline array is shown next:

  String[] cmdline = {"sh", "-c", " "};

- **send_sh_command_and_readint()**. This method is used to send a command to the Linux shell waiting for a return value from the command. This return value corresponds to the ADC sampled value. The command to be sent must be already in **cmdline** variable defined above. This command is also sent with the Java Runtime function **exec()**. In order to read the return value from the command, it is used the Java Runtime function **getInputStream()** together with the **BufferedReader** and **InputStreamReader** classes. The reading operation requires to be repeated until the buffer where the response gets stored is no longer empty. This is shown in this piece of code:

```
 r= Runtime.getRuntime();
p = r.exec(cmdline);
BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
 while ((s = br.readLine()) != null)
```

  This routine also requires to handle any possible exception. The data read by the above code is in a string format, so it's required to convert this to an integer value. This can be done with the **Integer.parseInt()** method. The value deliver by this last method is the value that the **send_sh_command_and_readint()** must return.

- **get_value()**. This function is used by upper software layers to take an ADC sample. As indicated before, the value is read from the file: /sys/bus/iio/devices/iio\:device0/in_voltage**x**_raw, where **x** is the channel number associate to this ADC pin. So, in order to read this path, the **send_sh_command_and_readint()** method is used setting previously the full command in **cmdline** array variable. For this, the 2nd element in this array needs to be filled in this way:

  cmdline[2] = "cat /sys/bus/iio/devices/iio:device0/in_voltage"+ adc_chnl +"_raw";

  In this command, **cat** is a Linux command used to see the content of a file, so basically with this full command we are indicating that it's wanted to read the content of the **in_voltageX_raw** file.

This function returns the value that the **send_sh_command_and_readint()** delivers after the command is processed.

Finally, once that the **Pin_adc** class is defined, it is necessary to create the object for the ADC channels that will be supported. For this, the top **semaphore_adc** class is defined. In the project presented here, only one ADC channel from **Pin_adc** class is required and this is declared inside **semaphore_adc** class. The object is associated with channel 0 and it's named **ADC_Ch0.** In order to create this channel object it's necessary to provide the channel name taken from **SemADCChnl** enumeration:

Pin_adc ADC_Ch0 = new **Pin_adc**(SemADCChnl.SEMADCCH0);

If it's necessary to create any other channel, then this needs to be instantiated in the same way inside this class.
In order to use the ADC channel created, it is necessary to instantiate an object from the **semaphore_adc** class inside the traffic agent class in an upper software layer.

TRAFFIC CONTROLLER APPLICATION

In this layer is implemented the real functionality of the intelligent traffic light controller agent being supported by the rest of the lower software layers described previously.

Agent Arguments

The traffic control system proposed in this work is thought to belong to a bigger system conformed by several nodes. Every one of the nodes has a unique identifier. At the same time, each one of these nodes is integrated by several traffic light agents which also have a unique identifier defined by the topology selected. This means that each one of the agents must receive a unique identifier number in the node and also which is the node that this belongs to. Also, it was indicated that the agent could be operating in 2 different modes when looking for the next agent leader:

a) Select the next leader agent following the transition table
b) Select the next leader agent using the traffic density ($\boldsymbol{\delta_T}$) metric as a decision parameter.

This operating mode is also something that the agent requires to know.

This set of data must be provided to the agent as parameters by the time the agent is started. These 3 parameters are named like this:

- **agent_num**: Agent number in the topology. Range [0 ... (number of agents -1)].
- **semaphore_node:** Node number in which the agent is located. Range [any integer value].
- **TransitionModeFollowTable:** Operation mode. Range [true, false]. True indicates that the agent operates following the transition table to choose the next leader agent. False indicates that the agent operates using $\boldsymbol{\delta_T}$ to select the next leader agent.

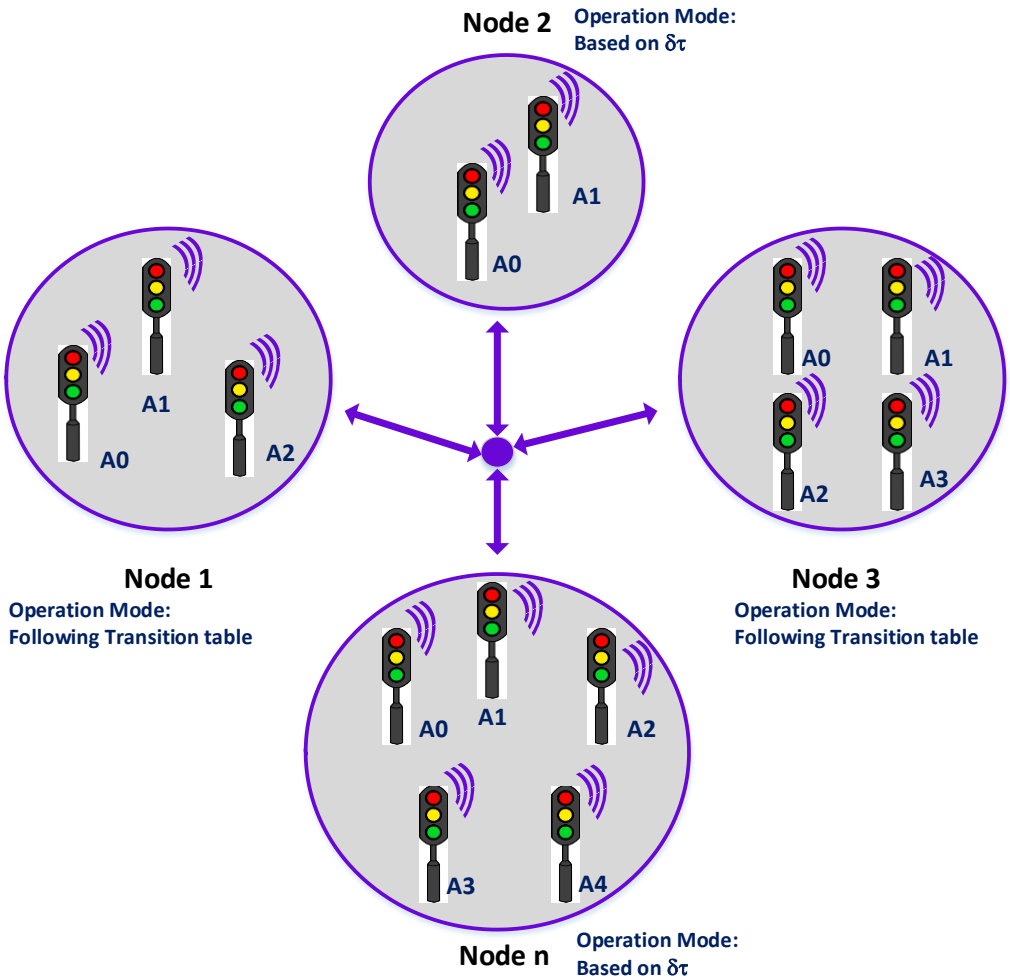


Figure 17. Node, Agent and Operating Mode definition.

Main Functional Areas

In order to describe the functionality and the software architecture of the application layer, this was divided in 5 main functional areas:

- **Traffic Monitor**. This is in charge of measuring the traffic density ($\boldsymbol{\delta_T}$) and detect if there is any emergency or overload condition.

- **Time Controller.** This keeps track of time continuously since the agent is started and is used to support timeouts or time trigger events.

- **Message Handler.** This is in charge of handle the Agent incoming messages taking the required action for each of the supported messages. In this functional area it is also covered the tracking of the incoming messages to check if the rest of the agents responded to any proposal message or if any agent sent any notification.

- **Main Behavior State Machine.** This is basically a finite state machine (FSM) that is in charge of coordinate all the processes required to achieve the expected functionality in the traffic light

agent. This FSM is composed of several states, in which in each one of them is implemented a specific task as part of the full system functionality.

- **Lamp Controller.** This is in charge of handle some specific behaviors related to the lamp control such as some required delays or blinking states.

These 5 areas cover the complete functionality of the traffic light controller. As the implementation language for this project is Java, these functional areas are implemented in form of classes, being the main class for the traffic light agent the **semaphore_agent** class which is extended from the Jade **Agent** class. Java allows to implement outer or inner classes, so some of the classes created to implement the functionality are created as inner class and some others as outer class basically depending if they may be useful for any other software layer. The next figure shows the functional areas and how they are integrated in the application layer indicating which of the classes are defined as inner or outer class from **semaphore_agent**.

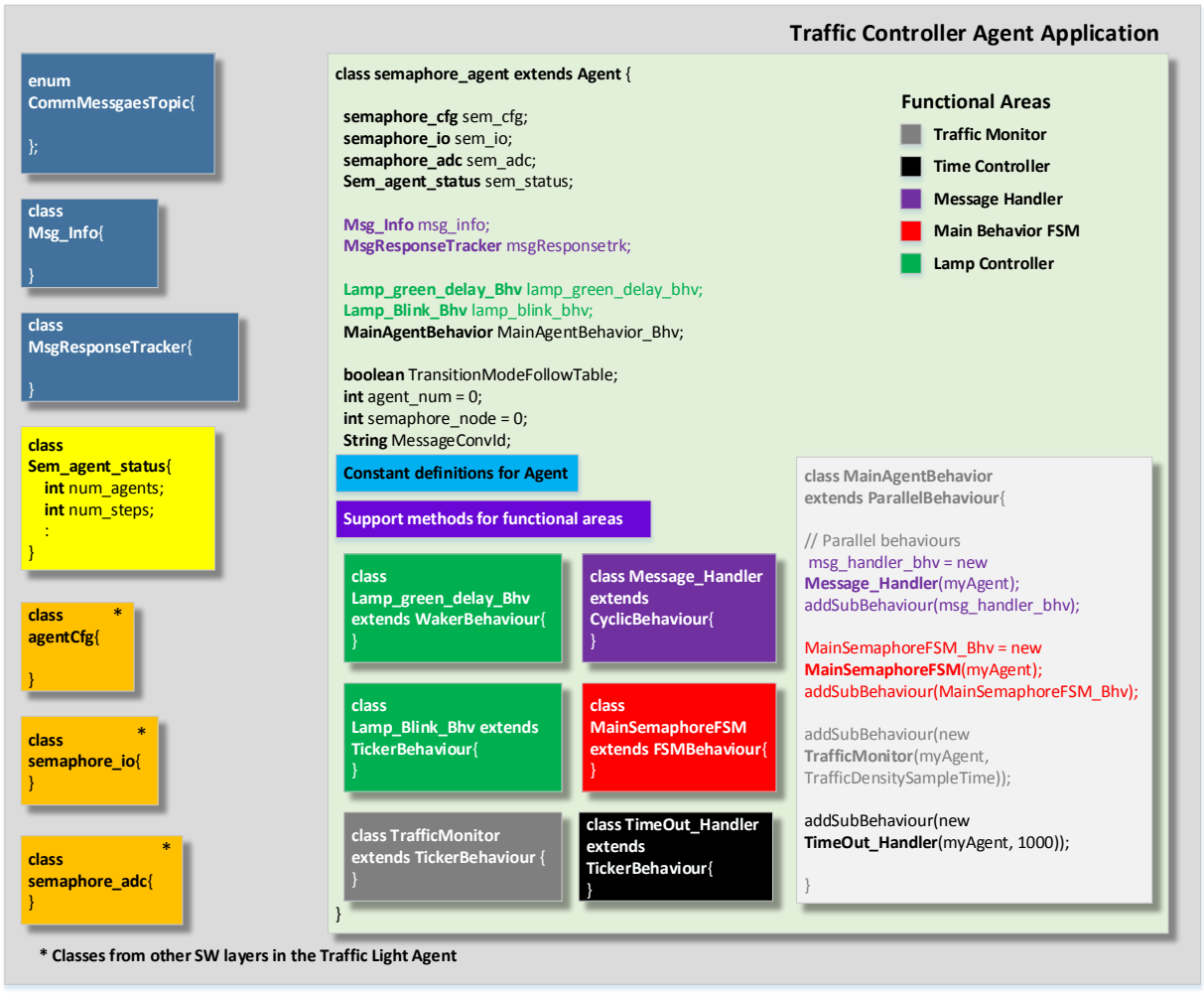


Figure 18. Application layer SW architecture and main functional areas.

In the figure it can be seen that inside of the **semaphore_agent** class are instantiated the classes that implements each one of the functional areas described above. Also the variables used to store the parameters provided to the agent (**agent_num**, **semaphore_node** and **TransitionModeFollowTable**) are instantiated as well. Also the objects for the configuration structure, the lamp IO driver and ADC driver are created inside this class. The classes from which these object are created are taken from lower software layers.
Moreover, inside this class there is a section where some specific constants used by the traffic light agent are defined. These constants define some values related to the behavior of the agent such as maximum time for a response, traffic density threshold to detect traffic overload, sample time for traffic monitoring, etc. These constants will be reviewed in detail by the time each one of the functional areas gets described.

In the figure it also can be seen that the **Sem_agent_status** class is defined. This class basically works as a data structure which contains variables used to keep the current status of the traffic controller agent. For example this contains a set of variables that indicates which of the processes are active or which of the agents is the current leader, and for how long its leadership will last. Some of the information obtained from the transition table from the Agent Configuration object is also stored in variables defined inside this structure. This can be seen in the figure as well. Similarly to the constants and status variables related to each one of the functional areas will be reviewed in the section corresponding to this functional area.

The **Traffic Monitor, Time Controller, Message Handler and Main Behavior State Machine** functional areas required to run in parallel as these are concurrent processes which need to be active all the time (this is not the case of the **Lamp Controller** functional area because the specific lamp behavior that this controls is only active under certain scenarios). In order to implement this parallel behavior, Jade provides the **ParallelBehaviour** class. For this project, the agent behavior is described in the **MainAgentBehaviour** class which is extended from the **ParallelBehaviour** Jade class.

Finally, the **semaphore_agent** class provides a set of methods and variables used to support the different functional areas. These methods and variables also will be described when the corresponding functional area gets reviewed.

TRAFFIC MONITOR SW SPECIFICATION

The traffic monitor is in charge of sampling the traffic load or density in the street that the traffic light agent is controlling. The traffic load is emulated using an adjustable voltage sampled using an ADC pin the Galileo board. In order to use the ADC port, the SW application layer requires to instantiate an object from the **semaphore_adc class** defined in the Semaphore ADC driver. This allows that the application layer makes use of the **ADC_Ch0.get_value()** method to get a $\boldsymbol{\delta_T}$ sample from ADC channel 0.
Each sample must be taken every 3 seconds. This parameter is defined as **TrafficDensitySampleTime** in the constant definition area inside of the semaphore agent.
In order to implement the traffic monitor functionality, the **TrafficMonitor** class must be implemented. This class can be extended from the Jade **TickerBehavour class.** This class is perfect for this because the code inside of the **onTick()** method is executed periodically based on the parameter given to the constructor when the class is created (in this case **TrafficDensitySampleTime** is referred to milliseconds). So, every time this method is executed, a sample of $\boldsymbol{\delta_T}$ must be taken and stored in the **traffic_density** global variable. Also in order to determine if this $\boldsymbol{\delta_T}$ is inside of a regular traffic range, this must be compared with an over-traffic pre-defined threshold value. This is defined as **EmergencytrafficDensity** in

the **Constant Definition** area. The result of this comparison is register in the **over_trafficdensity_st** variable defined in the **Status structure** and its value is determined in this way:

If $\delta_T \geq$ **EmergencytrafficDensity** then **over_trafficdensity_st** = **True**, otherwise is **False**.

This status variable will be used by other functional areas as a parameter when trying to determine which agent will be the next leader.
An object from the **TrafficMonitor** class must be declared inside the **MainAgentBehaviour** parallel class. The SW architecture of this functional area is shown in the next figure:

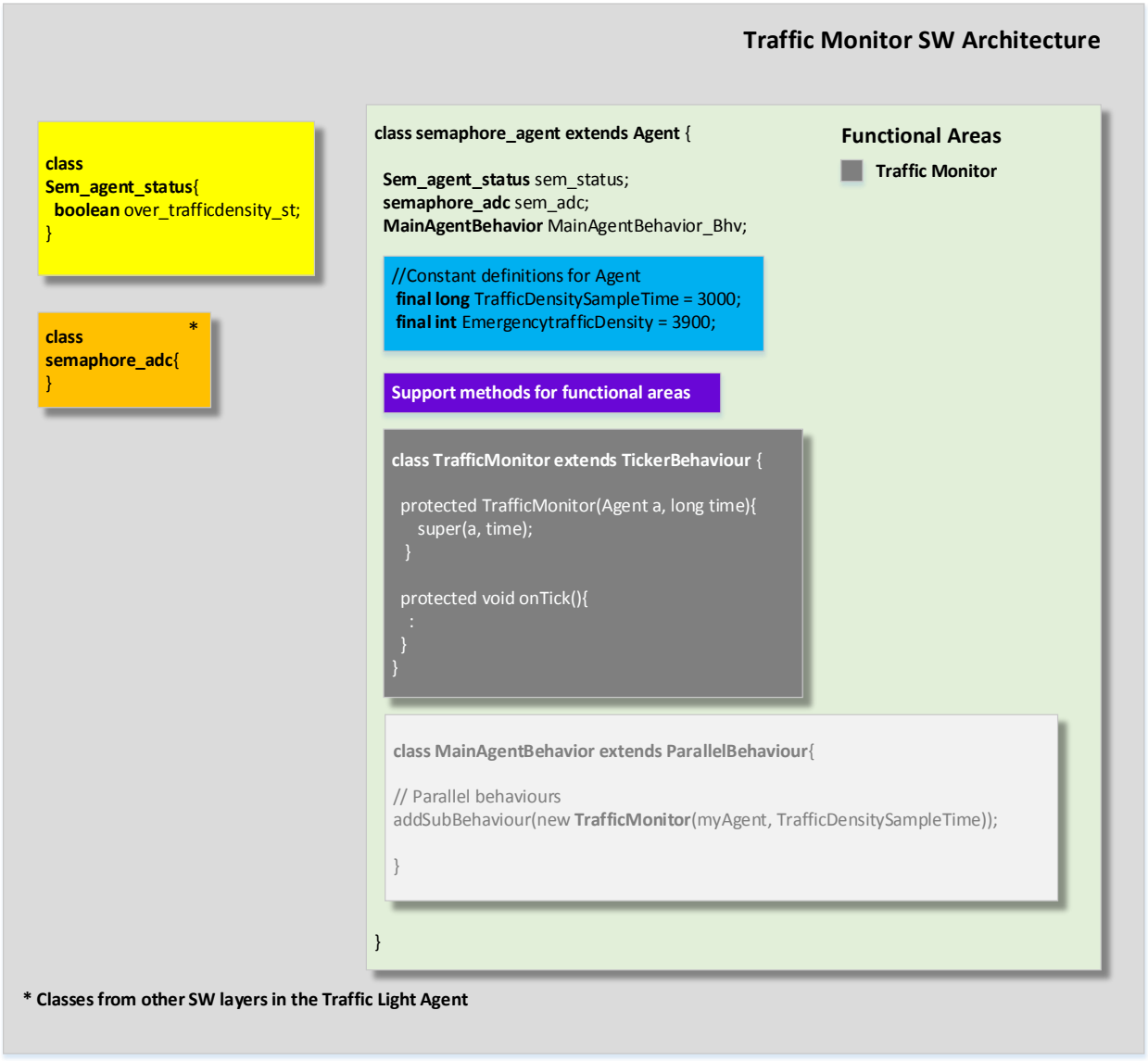


Figure 19. SW architecture of the Traffic Monitor.

TIME CONTROLLER SW SPECIFICATION

This functional area provides a time reference for other functional areas to support them to determine when they need to take any specific action. In order to achieve this, this functional area defines a variable that is increased every 1 second. This variable is defined as **TimeCount** in the Status structure. As this functionality can be modeled as a periodic task, the **TimeOut_Handler** class must be implemented extending the **TickerBehaviour** class in which **sem_status.TimeCount** is increased by 1 every time the **onTick()** method is executed. The period time (1 second or 1000 milliseconds) is provided as a parameter for the constructor when this object is created. An object from the **TimeOut_Handler** class must be declared inside the **MainAgentBehaviour** parallel class.
The software architecture for this functional area is shown in the next figure:

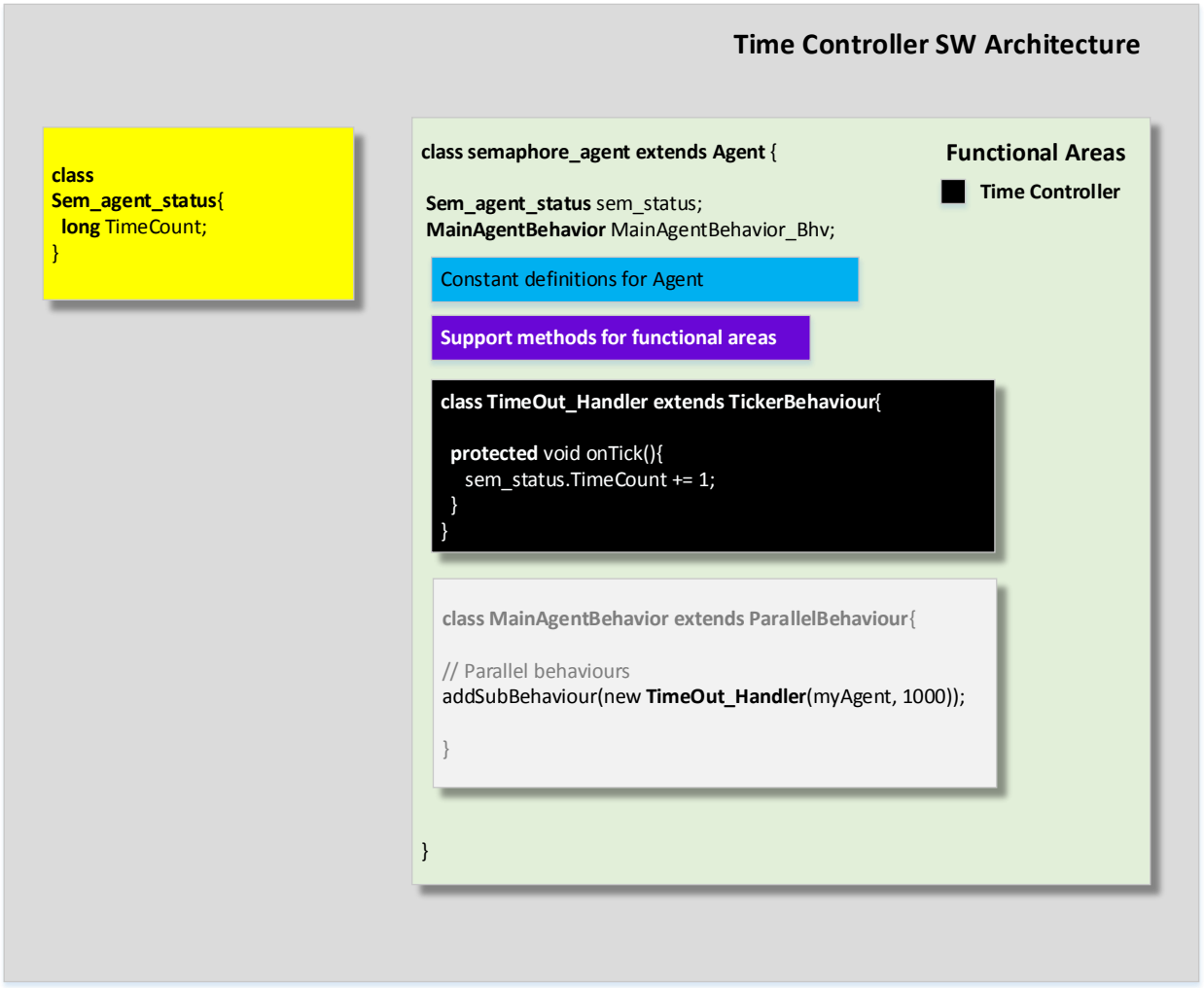


Figure 20. SW architecture of the Time Controller functional area.

LAMP CONTROLLER SW SPECIFICATION

This functional area is in charge of provide some specific behavior to the traffic light lamps based on the current step in the transition table in which the agent is located. The lamps states for one agent are controlled in group based on the information provided by the transition table depending on the step to transit to, indicated by the leader agent. In the transition table it's indicated the lamp state for each one of the lamps. The possible lamp states could be ON, OFF or BLINK. The agent requests to control the lamps using the IO driver functions **set_lights_in_block_but_blink(lamp_state)** and **blink_lights_in_block(lamp_state)** where **lamp_state** is taken from the transition table.

The next figure shows the software architecture of this functional area:

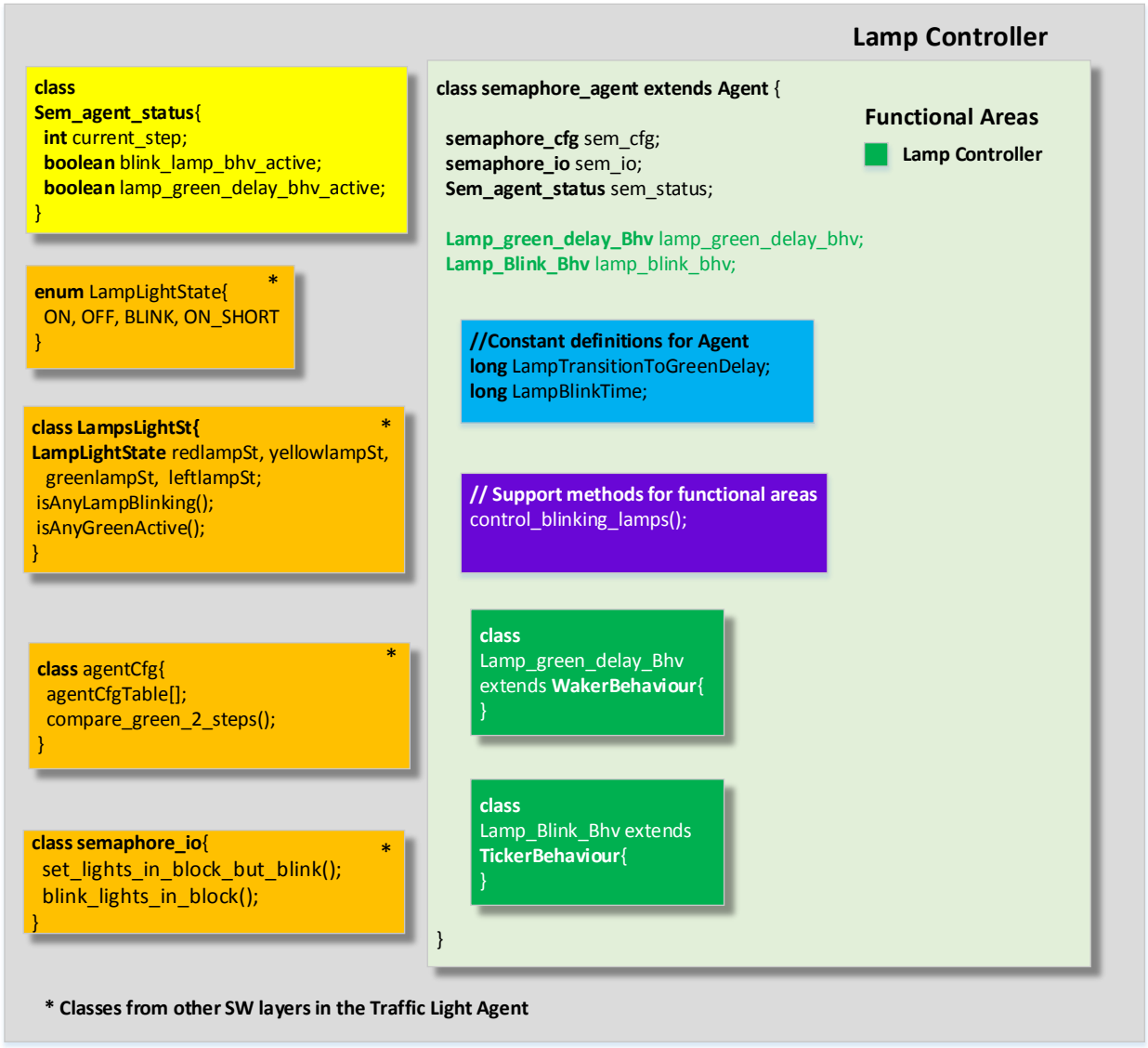


Figure 21. Software Architecture of the Lamp Controller functional area.

Additionally to turn on or off the lamps, this functional area provides 2 additional controls by means of 2 behaviors that requires some timing support. These behaviors are described next:

a) **Blinking lamps.** When is detected that any of the lamps is blinking, then, the agent requires to control the blinking time to turn on and off the lamp at a pre-defined rate.

b) **Delay on green transition.** When any of the agent lamps transits to green while any other lamp in the same agent was not in green in the previous step, it is required to delay the green transition for a brief period of time to prevent any possible collision with the vehicles that transits in the street whose traffic light transits to red.

For these 2 behaviors it's required to specify a couple of constants that defines the times used in them. For the blinking control, the **LampBlinkTime** variable is defined having a value of ½ second. For the green delay control, **LampTransitionToGreenDelay** variable is defined with a value of 1 second. These variables are declared in the **semaphore_agent** class.

Related to the status structure, also 2 Boolean variables are defined to indicate if any of the 2 processes is active. In the case of the blinking process, the **blink_lamp_bhv_active** is defined, while for the green delay process, the **lamp_green_delay_bhv_active** is declared. These variables are then used in case it's required to stop or abort any of these processes (in case of an abnormal scenario).

In order to support the whole functionality of this area, 2 Jade behaviours must be implemented.

1. **Blinking Lamp Process.**
   For the blinking lamp process it is required that the blinking lamp toggles every certain time, defined by **LampBlinkTime**. For this, a class extended from the **TickerBehaviour** must be implemented (**Lamp_Blink_Bhv**). This class is helpful for this purpose because this allows to execute an action periodically. When an object from this class is created, it is provided the execution period. The periodic action is executed in the **onTick()** method. In this method the blinking lamp must be toggled using the **blink_lights_in_block()** method from the IO driver providing as parameter the lamp state for current step in the transition table.

   When the behaviour is started it's necessary to be sure that this process is not being already executed. For this, in the **onStart()** function it's necessary to check the **blink_lamp_bhv_active** status variable. If it's in true state, this means that the process is already running and it's necessary to kill it calling the **stop()** function. This will prevent to have several blinking processes active at the same which will cause a fuzzy behavior in the blinking lamp. Now, if it's found that the process is not being executed then simply set the **blink_lamp_bhv_active** status variable in true as indication that the process is running. When the blinking time finishes, this process must be stopped. In this case the **blink_lamp_bhv_active** status variable must be set in false. Also it's necessary to make sure that the blinking lamp gets turned off. In order to achieve this, the **clear_blink_lamp()** method must be called also using the information of the lamp state of the current step in the transition table.

2. **Green Delay Process.**
   This process requires to generate a known delay in the green lamps when they are turned on. For this, the **WakerBehaviour** Jade behaviour can be extended to create the **Lamp_green_delay_Bhv**

class. When a process from this class is created it's required to provide as a parameter the delay time which is stored in the **LampTransitionToGreenDelay** variable.

When the process is started, the **lamp_green_delay_bhv_active** status variable must be set to true. This can be done in the **onStart()** function.

When the delay time elapses, the lamps in the agent are set according to the lamp state in the current step of the transition table. This is done in the **onWake()** method using the **set_lights_in_block_but_blink()** method. Also at this point it's necessary to activate the process to control any blinking lamp using **control_blinking_lamps()** method (this will be explained next).

Finally when this process ends, after the delay time elapses, it's necessary to set the **lamp_green_delay_bhv_active** status variable in false, as indication that the process is no longer running.

An instance of each one of these two classes must be declared inside the **semaphore_agent** class. Also, inside of this class it's necessary to provide a support method that is in charge of detect if there is any condition in which the blinking lamp process requires to be started. This support method is called **control_blinking_lamps()** and receives as a parameter the lamp state from the current step in the transition table.

The **control_blinking_lamps()** function must check if here is any blinking lamp in the current lamp state using the **isAnyLampBlinking()** function in the **LampsLightSt** class. If there is any blinking lamp, it must be checked if the blink lamp behavior is enabled using the **blink_lamp_bhv_active** status variable. If the process is not active, then a **Lamp_Blink_Bhv behaviour** must be started.
If it's the case that none of the lamps is in blinking state, it's necessary to check if the blink process is active. If so, then blinking process must be ended with the **stop()** function that is part of **Lamp_Blink_Bhv** class.

The lamp control methods and behaviours are used in the **handle_step_table()** function because the calling of this function implies a lamp state transition.

MESSAGE HANDLER SW SPECIFICATION

This functional are is very wide as it is in charge of handling all the aspects related to the communication among the agents and container including the sending of messages and handling the incoming messages. The next figure shows the software architecture for the Message Handler functional area:

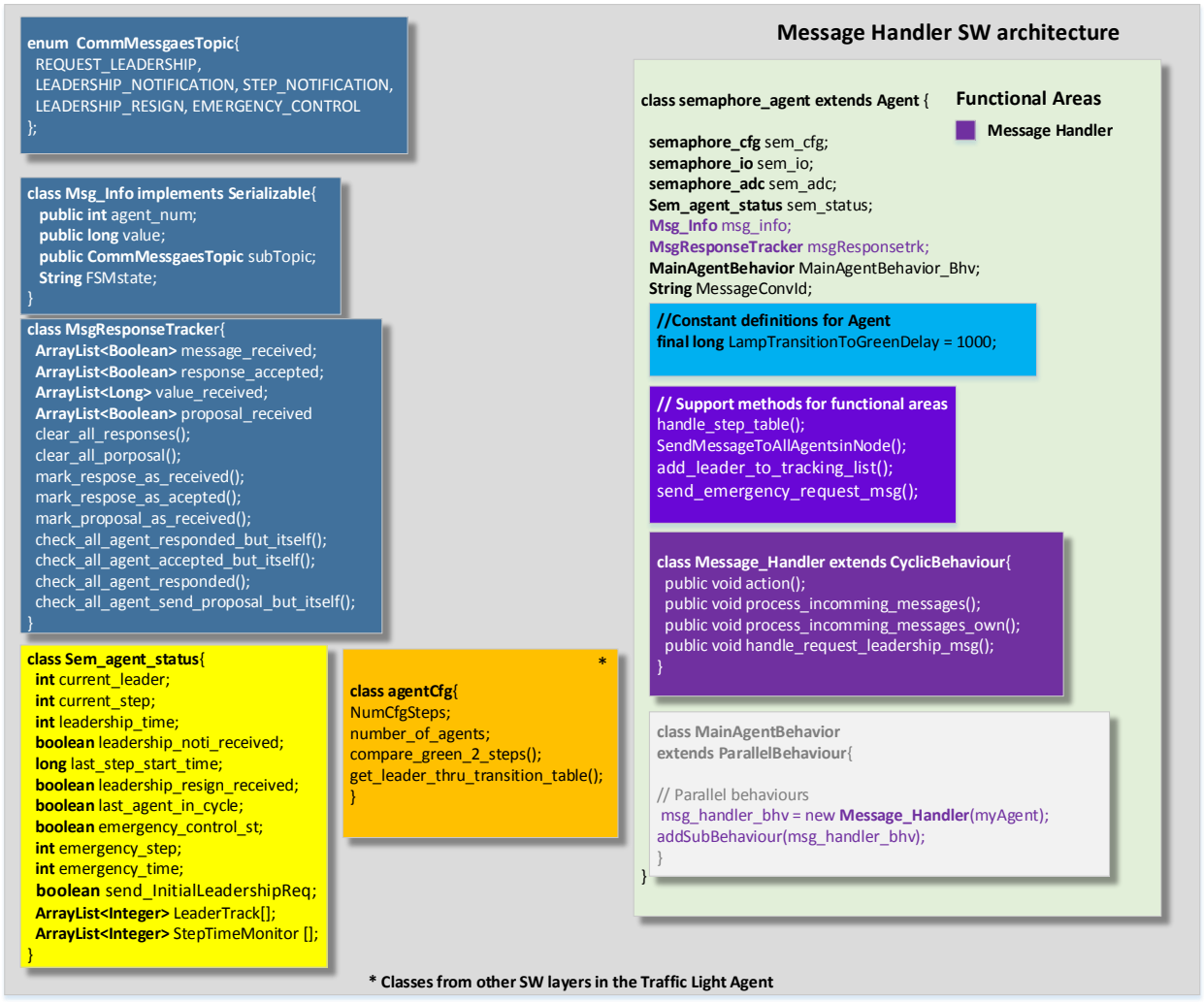


Figure 22 SW Architecture Specification of the Message Handler functional area.

The messages contain the information that is shared by the agents. For this work, the messages supported can fall into one of these categories or topics, which are defined in the **CommMessgaesTopic** enumeration.

- **REQUEST_LEADERSHIP.** Used by an agent to request being the leader of the group.
- **LEADERSHIP_NOTIFICATION.** Used by the agent that just gained the leadership of the group to notify the others about this.
- **STEP_NOTIFICATION.** Message sent by the agent leader to notify other agents that they need to transit to a step of the transition table.
- **LEADERSHIP_RESIGN.** Message sent by the agent leader to notify the other agents that this is no longer the leader of the group.

- **EMERGENCY_CONTROL.** This message can be sent by any agent (inside the node or not) or container to notify the other agents that there is an emergency situation and is needed to transit to the state in the transition table that this agent is requested.

For this project, the message payload is integrated by a set of data which is contained in an object of the **Msg_Info** class. The fields that composes this class are described next:

- **agent_num**: This is the agent that sends the message.

- **value**: This is any integer/long value that is part of the message to indicate something. For example, this field could be the $\boldsymbol{\delta_T}$ or the leadership time depending of the type of message sent.

- **subTopic**: This is the topic of the message that is being sent. This could be any of the types defined in **CommMessgaesTopic** enumeration.

- **FSMstate**: This is the current state of the agent that is sending the message. This state will be explained in detail later when the Main FSM Behavior functional area gets described.

The **Msg_Info** class is implemented of a **Serializable** type, which is Java specific type where an object can be represented as a sequence of bytes which can be fully recovered if it's stored in memory or transmitted in some way to another unit.

The JADE messages include a field called **ConversationID**. This field is used to differentiate among different types of conversations that the agents could have. All the messages that are transmitted in this node must have one single **conversationID** which needs to be stored in a class variable (**MessageConvId**). This variable is of String type and is created using the Node information provided as an input when the agent is started. The **conversationID** is created in this way: "**MessagesNodeX**", where X is the node number.

As indicated before, this functional area is in charge of handling the information embedded in a communication message. For this, it uses the **Status data structure** to keep this information. The most important fields in this structure are described next:

- **current_leader:** Indicates which agent is the current leader.

- **current_step:** Indicates in which state or step form the transition table the agent is at the moment.

- **leadership_time:** This registers the duration of the leadership of the current leader.

- **leadership_noti_received:** This registers if the current agent has already received the leadership notification or not. The agent that doesn't gain the leadership requires to receive the leadership notification message to transit to the next valid state.

- **last_step_start_time:** This registers the time, based on the **sem_status.TimeCount**, in which the current step notification message was received. This time is used to keep track of how long an agent stayed in certain step of the transition table.

- **leadership_resign_received:** Indicates if the leadership resign notification message has already received or not.

- **last_agent_in_cycle:** The behavior expected from the agent under a normal scenario is that each agent must be leader only and at least once in a cycle. This status variable indicates if this agent is the last one in the cycle. More detail about this will be explained when the Main FSM functional area gets described.

- **send_InitialLeadershipReq:** This indicates if the agent it's in process of sending the very first leadership request message. This is set to false before the 2nd leadership request message is sent.

- **emergency_control_st**: This indicates if the agent has received the EMERGENCY_CONTROL request message indicating that an emergency situation has occurred.

- **emergency_step and emergency_time**: When the EMERGENCY_CONTROL message is received it provides the step to which this agent needs to transit to and also for how long. So these variables stores this information.

- **LeaderTrack[]:** This is an array in which the agent keeps track of the previous leaders in the current cycle. This is used in the leadership selection process, specifically to warranty that only one agent can be leader once in a cycle under a normal scenario. The size of the array is defined by the number of agents in the node.

- **StepTimeMonitor[]**: This is an array in which the agent keeps track of the step time during one cycle. This is being track in order to re-calculate leadership time in case a step has already been touch in one cycle. Size is defined by the number of steps in the transition table.

In the agent communication, there are messages that requires a response either to accept or reject a message from a proposal message. This is the case of the **REQUEST_LEADERSHIP** message. In order to identify if an agent gained the leadership, it's needed to keep track of the responses. For this it is required to implement a message tracker. This message tracker is implemented by the **MsgResponseTracker** class. When an object from this class is created, it requires as a parameter the number of agents that integrates the traffic light system topology. This is needed because it's required to create dynamic arrays whose size depends on this number. The elements that integrate this message tracker are described next:

- **message_received[]** . This is a Boolean array whose size is the number of agents in the system. In this array is registered if a response (either accepted or rejected) by each one of the agents is received. This is used to know if the communication with the other agents is up.

- **response_accepted[]**.This is a Boolean array whose size is the number of agents in the system. In this array is registered if a response was accepted by each one of the agents. This is used to know if the agent who sent the request can be leader or not.

- **proposal_received[]**. This is a Boolean array whose size is the number of agents in the system. In this array is register if the current agent already received the **REQUEST_LEADERSHIP** proposal from the other agents. This is needed because it's necessary to have the traffic density from each one of the streets by the time the leader agent calculates the leadership time.

- **value_received[]**. This is Long Array whose size is the number of agents in the system. In this array is registered the $\boldsymbol{\delta_T}$ of each one of the agents when the **REQUEST_LEADERSHIP** message is received. This is used to calculate the leadership time for the agent that gain the leadership.

- **clear_all_responses()**. Method that clears the content of the **message_received[]** and **response_accepted[]** arrays defined above.

- **clear_all_porposal()**.Method that clears the content of the **proposal_received[]** and **value_received[]** arrays defined above.

- **mark_respose_as_received()**. Method that sets in true the **message_received[]** array for the agent specified as a parameter. Basically this means that the agent which sent the request received a response from a given agent.

- **mark_respose_as_acepted()**.Method that sets in true the **response_accepted[]** array for the agent specified as a parameter. Basically this means that the agent which sent the request received an accept message from a given agent.

- **mark_proposal_as_received().** Method that sets in true the **proposal_received[]** array for the agent specified as a parameter. Basically, this means that the agent which receives the proposal sets this flag as indication that other agent already sent its information.

- **check_all_agent_responded_but_itself()**. Method that evaluates if all the other agents responded to the request (checks in **message_received[]**, not check for itself). If so, then this responds true, otherwise responds false.

- **check_all_agent_accepted_but_itself()**. Method that evaluates if all the other agents accepted the request (checks in **response_accepted[]**, not check for itself). If so, then this responds true, otherwise responds false.

- **check_all_agent_responded()**. Similar to **check_all_agent_responded_but_itself()** method but in this case it also checks for the response from the message that sent the request.

- **check_all_agent_send_proposal_but_itself().** For one specific agent, this method checks that the other agents already sent their proposal to this agent, so it already has the information provided by the other agents.

An object from **MsgResponseTracker** class must be declared (**msgResponsetrk**) inside **semaphore_agent** class, so the agent can access to the elements defined in this tracker.

The message handling also requires some support functions defined in the main **semaphore_agent** class. These functions complements the functionality

- **handle_step_table()**. This is charge of updating the agent status when the **STEP_NOTIFICATION** message is received. This includes to update the status of the lamp and also the status of some variables related to the step. This function receives as a parameter the new step in the transition

table to which the agent is expected to transit to. The process in this function needs to follow these steps:

1. Check if the step is a valid step in the transition table. If not exit displaying an error.

2. Get the lamp states from current step and new step.
3. Check if there is any green lamp promotion with **sem_cfg.compare_green_2_steps()** method. If so, then delay the green transition using the **Lamp_green_delay_Bhv**. The delay will be 1 second (**LampTransitionToGreenDelay** in the constant definition section) and it is used to give enough time to other agents to clear any possible green. If doesn't exist any green promotion, then just update the lamps states with sem_io.set_lights_in_block_but_blink() method and make sure that the blinking process gets enabled if any lamp is in blink state(this is achieved by simply calling **control_blinking_lamps()** method).

4. Update **sem_status.StepTimeMonitor**[] for the current step (not new step) with the time that this step lasted. This time is calculated in this way:
   StepTimeMonitor[] = **sem_status.TimeCount** - **sem_status.last_step_start_time**

   This is updated only if **last_step_start_time** is different than 0, meaning that it's a valid time.

5. Update the **current_step** status variable with the step parameter provided.

6. Update the **last_step_start_time** status variable with **sem_status.TimeCount**;

- **add_leader_to_tracking_list()**. This method adds an agent to the status **LeaderTrack[]** list. The agent is provided as a parameter. This method also checks if the agent is not already in the tracking list. If so, then it will display an error, as this may be an invalid condition. Finally, once the agent is added to the tracking list, it checks if the size of the list if equal to the total number of agents. If so, the **sem_status.last_agent_in_cycle** needs be set to true, meaning that a cycle will be completed by the end of current leadership.

- **SendMessageToAllAgentsinNode()**. This method sends a message to the agents in the system deciding to send it or not to itself depending on a parameter provided. The message information must be pre-filled in the **msg_info** global variable. The 2 parameters that this method receives are these:

  1. **Message Performative**. This indicates what the nature of the message is. For this project this can be **ACLMessage.INFORM** or **ACLMessage.PROPOSE**.

  2. **filter_itself**. This is a Boolean flag that when True the current agent is discarded from the receiver list. When it's false, then the current agent becomes part of the receiver lists.

  The first step is build the message. For this, a message from **ACLMessage** class must be created using as a parameter for the constructor the message performative type specified. Once that the message is created, it is needed to assign a conversation ID using the **setConversationId()** method passing as parameter the ID for this node stored in the **MessageConvId** variable. Also, it's

necessary to attach to the message the list of the message receivers. This is done with the **addReceiver()** method passing one by one the receivers of the message in this way:

msg.addReceiver(new AID(SemX , AID.ISLOCALNAME))

where **X** corresponds to the number of the receiver agent. If the **filter_itself** parameter has a value of **TRUE**, then the current agent must be removed from the list, otherwise all the agents in the node must be added to the list.

- **send_emergency_request_msg()**. This method sends the **Emergency_Control** request message. This is the case in which independently of the scenario the other agents will follow the step transition indicated by this message. This function receives 2 parameters. They are the step to transit and the time in which the receiver agents must remain in this step. The message information needs to be filled like this:
    1. **msg_info.agent_num**: Any value. Really don't care.
    2. **msg_info.subTopic** = **CommMessgaesTopic.EMERGENCY_CONTROL**
    3. **msg_info.FSMstate** = **sem_status.agt_state**. Really don't care
    4. **msg_info.value** = (time << 8) | step;

  The message need to be sent using this function. Note that **filter_itself** is set to false meaning that the message need to reach to this agent as well.

  SendMessageToAllAgentsinNode(**ACLMessage.INFORM**, false)

The processing of the incoming messages must be handled by the **Message_Handler** class. This class is extended from the Jade **CyclicBehaviour** whose **action()** method is executed continuously (kind of infinite loop). This Behaviour is specific to handle incoming messages. The methods that this class provides are described next:

- **process_incomming_messages()**. This method handles the incoming messages that are only supposed to be sent by an agent different than itself. This method receives as a parameter an **ACLMessage** message (**msg_rcv**). The content of the message must be extracted by:

  msg_content = (Msg_Info) msg_rcv.getContentObject();

  The **msg_content.subTopic** requires to be extracted and compared with the possible options that the system supports (defined in **CommMessgaesTopic** enumeration). In each case a different action needs to take place as indicated next:

    1. **REQUEST_LEADERSHIP**. Call the **handle_request_leadership_msg()** function to process this message.

    2. **LEADERSHIP_NOTIFICATION**. This message provides the new leader in the **agent_num** field and the leadership time in the **value** field. Based on this it is required to update the next status variables using the information provided in the message:

       sem_status.current_leader = **msg_content.agent_num**;
       sem_status.leadership_time = **msg_content.value**;
       sem_status.leadership_noti_received = true;

Also the new leader needs to be added to the leader tracking list:

add_leader_to_tracking_list(**msg_content.agent_num**);

3. **STEP_NOTIFICATION**. This message indicates that it's requested to transit to a new step from the transition table. This message must be processed only if the agent that sends the message is the leader agent. In order to process this message the **handle_step_table**() method needs to be called providing as a parameter the new step to transit to (for this type of message this is provided in the **msg_content.value** field). If it's the case in which the new step is 0 (initial step), the system needs to be restarted using the **restart_system()** function which will be described later.

4. **LEADERSHIP_RESIGN**. This message is received when the current leadership is giving up its leadership to other agent. When this message is received, the status variable **sem_status.leadership_resign_received** needs to be set to true. Also if it's found that that all the agents already got the leadership in the current cycle (i.e. **sem_status.last_agent_in_cycle** is true) then the step times registered in **sem_status.StepTimeMonitor** and **sem_status.last_agent_in_cycle** needs to be cleared.

5. **EMERGENCY_CONTROL**. This message is received under an emergency scenario and it's requested that the receiver agent goes to one specific step of the transition table for certain time. This step and transition time is obtained from the **msg_content.value** and these status variables must be updated in this way:

   sem_status.emergency_control_st = true;
   sem_status.emergency_step = msg_content.value & 0xFF;
   sem_status.emergency_time = (msg_content.value >> 8)*1000;

   In order to transit to the **EmergencyControl** state it's required to abort any process that it's running. This is achieved calling the **restart_system()** function which will be explained later when the Main FSM functional area gets described.

For all of the possible message types (except **EMERGENCY_CONTROL),** they are executed only if it's not in emergency control state (**sem_status.emergency_control_st == false**), otherwise the message must be ignored.

- **process_incomming_messages_own()**. This method is similar to the **process_incomming_messages()** but it only process messages that can be sent by this same agent. So under this category falls the **EMERGENCY_CONTROL** subtopic. The actions that need to be implemented when this message is received, are exactly the same than in the ones described in the **process_incomming_messages()** method.

- **handle_request_leadership_msg**(). This method is in charge of handling the incoming message with subtopic equal to **REQUEST_LEADERSHIP**. This function receives as an incoming parameter the message that was received. The messages with this type of subtopic can have 3 different type of performative depending on the context in which the message was sent: **PROPOSE**, **ACCEPT_PROPOSAL** and **REJECT_PROPOSAL**. The actions that needs to be follow in any of these cases are described next:

1. **PROPOSE**. This message is received from an agent that requested to be the leader. The information provided in the **Msg_Info** object inside of this type of message is relative to the agent that sends the message. The information provided in this data structure is the number of agent in the topology, the $\boldsymbol{\delta_T}$ measured in the street that the agent controls, the subtopic (**REQUEST_LEADERSHIP)** and the FSM state of the agent. The $\boldsymbol{\delta_T}$ information received in the message must be tracked in the **value_received[]** array of the message tracker and the proposal also must be registered in the proposal tracking array using **mark_proposal_as_received()** method. For this type of message, it is expected that the receiver agent responds to this message accepting or rejecting the leadership request, although there are certain conditions in which the message must be ignored. The cases in which this message is ignored will help to synchronize the agents at the start up or in case of a restart process, because a missing response must generate a restart in the sender agent.

   In order to decide if accept, reject or ignore the message, the agent must analyze the information provided within the message and the local conditions stored in the status structure. The condition evaluation must be done in this way (note that the FSM states mentioned next, will be explained in detail later):

   a) Ignore the message if the receiver agent is in **FSM_IDLE_ST** state and it hasn't acquire enough $\boldsymbol{\delta_T}$ samples. The second condition can be checked with the **send_InitialLeadershipReq** status variable which is true during the first leadership request.

   b) Ignore the message if the sender agent is in **GET_INITIAL_LEADERSHIP_ST** and the receiver agent doesn't come from IDLE state. This second condition also can be checked with the **send_InitialLeadershipReq** status variable.

   c) Ignore the message if the receiver agent is in **GET_INITIAL_LEADERSHIP_ST** or **FSM_IDLE_ST** and the sender agent is in any state different than **GET_INITIAL_LEADERSHIP_ST.**

   d) Get the $\boldsymbol{\delta_T}$ from the incoming message and add it in the corresponding index for the sender agent in **msgResponsetrk.value_received**[] to register this value for leadership calculation.

   e) If there is traffic overload (either in the street of the sender agent or in the street of the receiver agent), the leadership will be accepted only if the proposed $\boldsymbol{\delta_T}$ is greater than local $\boldsymbol{\delta_T}$. If the local $\boldsymbol{\delta_T}$ is greater or equal the proposed $\boldsymbol{\delta_T}$ then the proposal will be rejected. The indication of local traffic overload is register in the **over_trafficdensity_st** status variable, while the traffic overload threshold is defined by the **EmergencytrafficDensity** constant.

   f) If there isn't any traffic overload. The accepting or rejecting of the leadership could follow 2 ways: assign leadership following transition table or assign leadership based on $\boldsymbol{\delta_T}$. The operation mode is defined by the **TransitionModeFollowTable** parameter.

g) In the leadership mode based on transition table, the next leader is obtained from the **get_leader_thru_transition_table()** function in the configuration object. If the sender agent matches with the agent that the function returns, then the proposal will be accepted, otherwise will be rejected.

h) In the leadership mode based on $\boldsymbol{\delta_T}$. There are also some checks to do. If the current agent is already in the **LeaderTrack**[] list for this cycle, this agent needs to discard itself from the leadership arbitration accepting the leadership proposal. If the sender agent is already in the **LeaderTrack**[] list for this cycle, then the proposal needs to be rejected to avoid that the sender agent gains the leadership again in this cycle. Finally if the proposal $\boldsymbol{\delta_T}$ is greater than the local $\boldsymbol{\delta_T}$ the proposal must be accepted, otherwise this must be rejected.

The response message is created using the Jade function **createReply().** The response fields also needs to be filled. If the proposal is accepted, the **setPerformative** field needs to be set as **ACLMessage.ACCEPT_PROPOSAL**. If the proposal was rejected this field needs to be set as **ACLMessage.REJECT_PROPOSAL**. The message content of **Msg_Info** type is also filled with the current agent number, the value field is set with 0, the subtopic is set as **REQUEST_LEADERSHIP** and the agent state will be the current agent state. This data structure is included in the response message with the **setContentObjec()** method and the message is sent with the **send()** method.

2. **ACCEPT_PROPOSAL.** This message means that the leadership proposal was accepted by the agent to which the request message was sent. As the proposal was accepted, the response and information must be filled in the message response tracker (**msgResponsetrk).** In the case of **message_received**[] and **response_accepted**[], they need to be marked with TRUE in the location indicated by the number of agent that accepted the proposal. In the case of **value_received**[] list, this needs to be filled with the current $\boldsymbol{\delta_T}$ at the location corresponding to the local agent.

3. **REJECT_PROPOSAL.** This message means that the leadership proposal was rejected by the agent to which the request message was sent. As the proposal was rejected, this agent is not going to become the next leader, so in the message response tracker (**msgResponsetrk**), only **message_received**[] needs to be set as true in the location corresponding to the agent that rejected the proposal.

- **action**(). As indicated before, this method is part of the Cyclic Behaviour class and inside of this is located the code that is executed continuously. Basically inside this method, the process must wait for any incoming message that matches with a template format. In this case, the template is defined by the conversation ID set for this node and stored in **MessageConvId** variable. The template is set with the **MessageTemplate.MatchConversationId()** method. Then it gets one of the incoming messages that matches with the template using the **myAgent.receive(template)** method. Once that a message is received, it needs to be processed either by **process_incomming_messages()** or **process_incomming_messages_own()** function depending if the agent sender is itself or a different agent. After that, the agent blocks itself using the **block()** method until a new incoming message is received. This flow can be better appreciated in the next flow diagram:

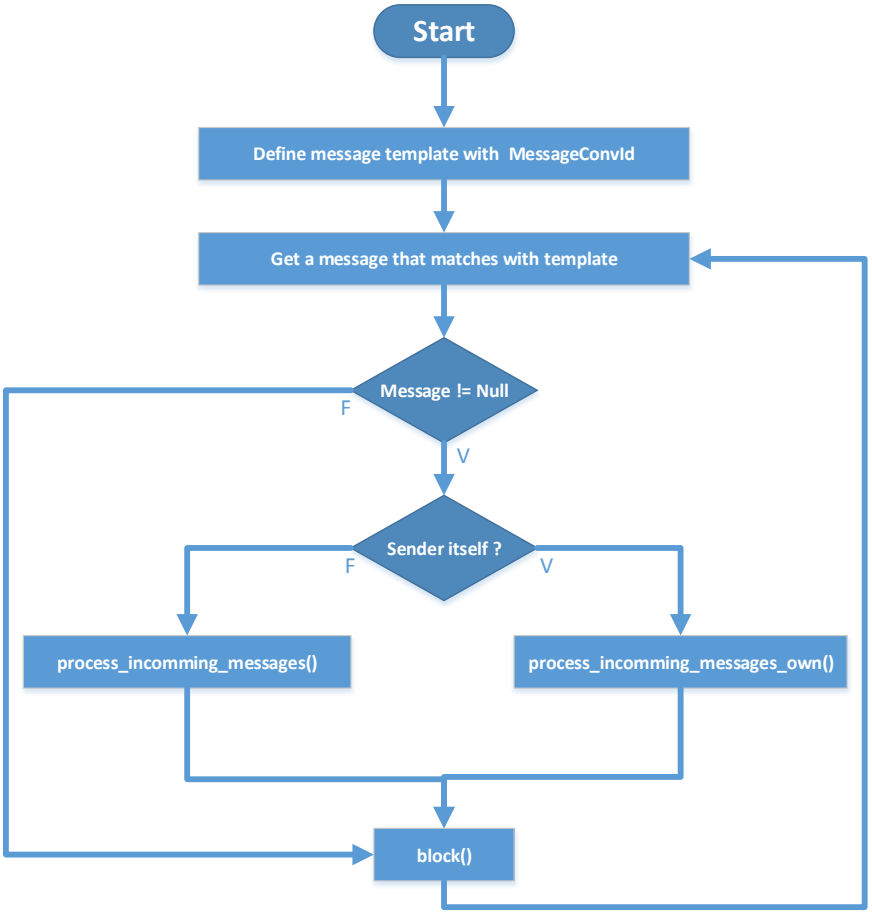


Figure 23. Main Message Handler flow in action () method.

MAIN STATE MACHINE AGENT CONTROLLER SW SPECIFICATION

This functional area is in charge of controlling any synchronizing all the processes of the traffic light agent. As its name indicates, the control is implemented by means of a finite state machine (FSM) being supported on the methods and processes provided by the other functional areas. The FSM is integrated by 7 states which provides the overall control that the traffic light controller requires. The states that integrates the FSM are listed next, providing the functional description of each one of them:

1. **FSM_IDLE_ST**. This is the initial and default state. The system must stay in this state for enough time to allow obtaining valid and stable values of $\boldsymbol{\delta_T}$ before transiting to the next state. In this state, the traffic light lamp state must follow what the step 0 in the transition table indicates. Once that this time elapses, the system must go to the **FSM_GET_INITIAL_LEADERSHIP_ST** state. This is also the state to which the system must go in most of the cases in which a failure condition is detected.

2. **FSM_GET_INITIAL_LEADERSHIP_ST**. The agent enters in this state only once after passing by **IDLE** state. In this state the traffic light controller must request the leadership to the other agents in the system sending to them the **REQUEST_LEADERSHIP** message along with its $\boldsymbol{\delta_T}$. In this state

the agent also must be ready to receive leadership request from the other agents and provide a response (accept or reject) based on the current conditions (its own $\boldsymbol{\delta_T}$ and leadership selection mode). The agent must register each one of the $\boldsymbol{\delta_T}$ received as they will be used later to determine leadership time in case the agent becomes the leader agent. The agent must set a timeout to obtain any response. The agent must keep tracking of each one of the agents responses. If all other agents accepts the proposal, the agent must transit to **FSM_LEADERSHIP_GAINED_ST** state. If any of the agents rejects the proposal, then it must transit to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state. In case the agent doesn't get a response from any of the agents before the timeout expires, then it must restart all the process in this state.

3. **FSM_LEADERSHIP_GAINED_ST**. This is the state in which the agent leader must move once it gains the leadership. In this state, the agent must turn-on the leader indicator lamp to indicate that this is the leader agent. In this state the agent must get the list of steps for the current leadership period. This step list is taken from the transition table and the selection of the 1st step must be done avoiding having overlap with the list of steps for the previous leader agent. Once that it's determined the list of steps, it is required to get the leadership time. This leadership time is calculated in several stages as described next:

   - The initial calculation of the leadership time is obtained using the $\boldsymbol{\delta_T}$ reported by each one of the agents, the maximum $\boldsymbol{\delta_T}$ reported by the agents in the last round, the maximum $\boldsymbol{\delta_T}$ possible, the maximum leadership time that an agent could have and the minimum leadership time allowed. The value of the last 3 variables are pre-defined for the system. This calculation is as follow:

     Add the $\boldsymbol{\delta_T}$ reported of each one of the agents:

     $$Sum\delta_T = \sum_{i=0}^{\#agents-1} \delta_T[i]$$

     Get the minimum leadership time using the minimum step time and the number of leadership steps:

     $$leadershipTimeMin = StepTimeMin * \#LeaderSteps$$

     Get the cycle time in seconds based on the relation between the highest $\boldsymbol{\delta_T}$ reported and the maximum $\boldsymbol{\delta_T}$ possible, taking as reference the Maximum leadership time:

     $$CycleTime = \frac{MaxCycleTime*1000*\boldsymbol{\delta_T High}}{\boldsymbol{\delta_T Max}} \text{ in ms}$$

     Get the leadership time based on the relation between the $\boldsymbol{\delta_T}$ reported by the leader agent and the $Sum\boldsymbol{\delta_T}$ reported by the agents, taking as reference the cycle time calculated:

     $$leadershipTime = \frac{\boldsymbol{\delta_T[leader]*CycleTime}}{\boldsymbol{Sum\delta_T}*1000} \text{ in sec}$$

     Adjust the leadership time to warranty that this is at least the minimum leadership time ($\boldsymbol{leadershipTimeMin}$).

- It's possible that between 2 consecutive leadership periods some of the steps get overlapped. In this case it's is necessary to adjust the leadership time to discard the time that was already

touched by the overlapped steps that were already executed in the previous leadership period. In order to obtain the overlapped steps, it's required to call the **get_intercepted_steps()** method in the Object configuration agent. The step time registered in the step time tracker is used to obtain the times of the overlapped steps which requires to be subtracted from the leadership time previously calculated. Only do this if no overload condition is detected.

- After getting this initial leadership time it's required to get the time for each one of the steps. The step time calculation is done based on the pre-calculated leadership time and the default time that some steps has assigned (this is the case of yellow time, blink time or left green time). The step time calculation was already presented when it was described the **get_steps_time()** method in the Object configuration agent. This will produce a list of time steps.

- It's possible that in the same cycle some steps have been already touched under the leadership of other agent. If this is the case, and the agent is not detecting a traffic overload condition, then the step time for the repeated steps need to be readjusted in order to remove the time already spent in that step always warranting the minimum step time. The step time is adjusted subtracting to the calculated time the step time register in the tracking list. At this point is important to remember that the step time is register in the tracking list every time the **STEP_NOTIFICATION** message is sent or received. Only do this if no overload condition is detected.

- Once that the time for each step is adjusted, the total leadership time must be recalculating adding the step time of each step:

$$leadershipTime = \sum_{i=0}^{\#steps-1} stepTime[i]$$

The current leader needs to be added to the leaders tracking list for the current cycle and it must send the **LEADERSHIP_NOTIFICATION** message to all the agents to indicate that the leadership has be gained by this agent indicating also the leadership time.
After this, the agent must setup a very short timeout (~2 seconds) to allow the other agents to receive the **LEADERSHIP_NOTIFICATION** message. Also it must setup a timeout used to re-send its new leadership request programed to expire in:

$$timeout = leadershipTime - LeaderShipReqTimeBeforeLeaderFinish$$

This last value in the equation is a pre-defined parameter in the system.

Once that the short timeout expires, the agent must iterate over the step list updating the lamps state based on the steps in the list sending also the **STEP_NOTIFICATION** message to other agents so they can update their own lamp state. In every iteration the agent must setup a timer with the duration of the current step, and once the timer expires then it needs to repeat the process for the next step in the list.

Once that initial time out expires (set with $LeadershipReqTimeBeforeLeaderFinish$), the agent must send again the **REQUEST_LEADERSHIP** message to the others agents with the new $\boldsymbol{\delta_T}$. After sending this message, the agent must wait for a response from the agents. In case any of the agents doesn't respond the agent must transit to **FSM_IDLE_ST** state. In case all the agents responded, either accepting or rejecting the proposal, the agent must transit to **FSM_SEND_LEADERSHIP_RESIGN_ST**.

4. **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST**. The agent enters in this state when its leadership request was rejected. In this state the agent waits for the **LEADESHIP_NOTIFICATION** message sent by the agent who gained the leadership. For this, the agent sets a timeout using the **WaitForLeadershipNotificationTimeout** parameter. If this message is not received before the timer expiration, then the agent must transit to **FSM_IDLE_ST.**
   In case the **LEADESHIP_NOTIFICATION** is received, it needs to turn off the leader indicator led, register the new leader agent into a tracking list and also the agent needs to setup a timeout defined by:

$$timeout = leadershipTime - LeaderShipReqTimeBeforeLeaderFinish$$

   The $leadershipTime$ parameter comes along with the message. This timeout should expire few seconds before the current leadership period finishes and once that this expires it needs to send the **REQUEST_LEADERSHIP** message with the new $\boldsymbol{\delta_T}$. After this, the agent must transit to the **FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST** state.

5. **FSM_SEND_LEADERSHIP_RESIGN_ST**. The agent enters in to this state after its leadership period completes. At entry to this state, the agent must check in the incoming messages tracking list if this agent will be the next leader or not (depending on the acceptance or rejection of the **REQUEST_LEADERSHIP** message). After this, the agent must send the **LEADERSHIP_RESIGN** message to the other agents in the system.
   In case this agent become the next leader, it needs to check if this agent already received all the leadership requests from the other agents. If so, it must transit to **FSM_LEADERSHIP_GAINED_ST**, otherwise the system must be re-started. If another agent will become the next leader then it must transit to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST.**

6. **FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST**. The agent enters in this state from **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** when it already transmitted the **REQUEST_LEADERSHIP** message to the other agents. On entry to this state, the agent must setup a timer based on **WaitForLeadershipResignTimeout**. This timeout will be used to give sometime to the leader agent to send the **LEADERSHIP_RESIGN** message.
   If this message is not received before the timer expires, then the agent most transit to **FSM_IDLE_ST**. In the other hand, if the message is received the agent must check in the incoming message tracking list if all the agents responded to the **REQUEST_LEADERSHIP** message. If any of the agents didn't respond to the message, then the agent must transit to **FSM_IDLE_ST** state. If all the agents accepted the message then the agent must transit to **FSM_LEADERSHIP_GAINED_ST** state, otherwise it must transit to the **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST**. In the case of having gained the Leadership, it also must check if this agent already received the Leadership request from the other messages. If not, then it must re-start de system.

7. **FSM_EMERGENCY_CONTROL_ST**. This is an emergency state in which the agent enters when it receives the **EMERGENCY_CONTROL** message. When the message is received, the agent must abort any process running and moving to this state. For this, each one of the processes created in the application layer must have a flag that indicates if this process is active. If the process is active,

then this must be stopped and then enter in this state. The **EMERGENCY_CONTROL** message contains some information embedded. The information embedded is like this:

msg_content.value = (emergency_time << 8) | emergency_step

On entry of this state, the agent must transit to the step specified in the message, indicated by emergency_step, for the number of seconds indicated by emergency_time. After this time, the agent must go to **FSM_IDLE_ST** and restart the whole flow in normal mode. If the **EMERGENCY_CONTROL** message is received in the **FSM_EMERGENCY_CONTROL_ST** state, the agent must remain in the same state, but setting the new step and restarting the time for which it must remain in this state.

In any of the states described above, if an invalid scenario is seen and it requires to transit to **FSM_IDLE_ST** state, the agent must abort any of the running processes before transiting to this new state.

The next figure shows the diagram of the FSM described:

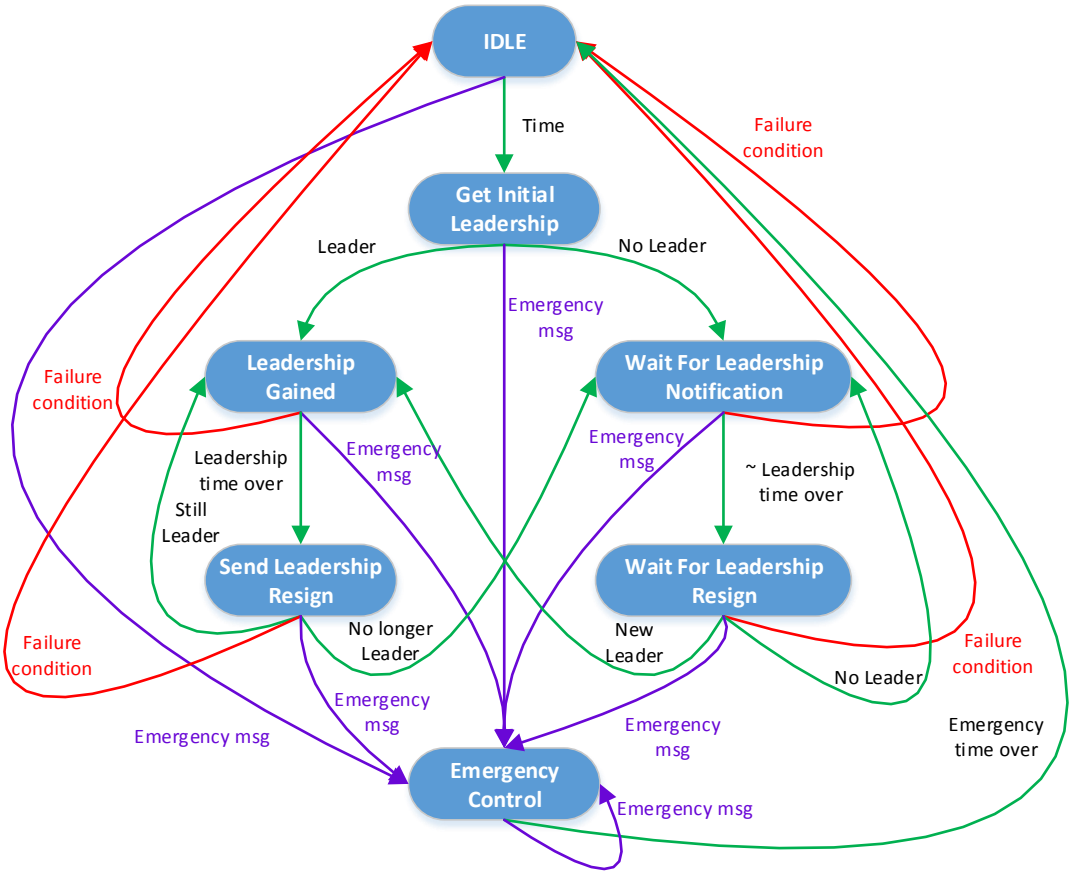


Figure 24. Main Traffic Light Controller FSM.

Next it provided some implementation details using the Jade middleware.

As the main controller is actually a FSM, it is required to implement this in the application layer. For this, it is defined the **MainSemaphoreFSM** class which is extended from the Jade **FSMBehaviour** class. This

class allows to create specific Jade behaviours for each one of the FSM states, allowing to execute the required actions in each one of them. An object of this class must be created under the **MainAgentBehavior** parallel class and a sub-behaviour of this object must be added when creating this parallel class.

The next figure shows the big picture of the SW architecture of the FSM controller. The structure of the **MainSemaphoreFSM** class will be shown later.

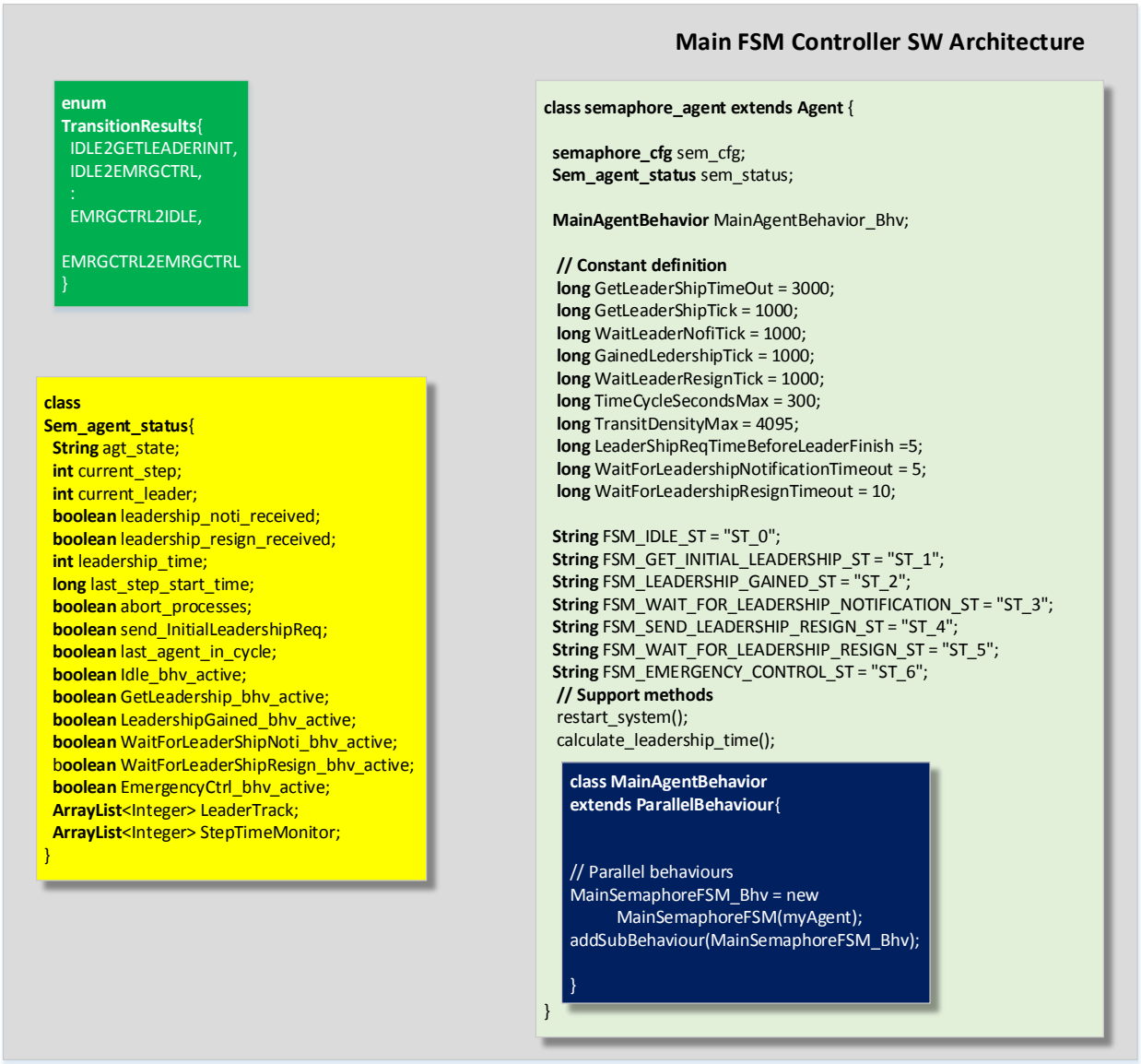


Figure 25. Main FSM Controller SW Architecture.

The FSM implementation is supported by a set of status variables and constants to have tracking and control of process running as part of this FSM.

First it's presented a list of the constant used:

- **FSM_IDLE_ST, FSM_GET_INITIAL_LEADERSHIP_ST, FSM_LEADERSHIP_GAINED_ST, FSM_SEND_LEADERSHIP_RESIGN_ST, FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST, FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST, FSM_EMERGENCY_CONTROL_ST**. These are string type constants which are referred to each one of the FSM states. These constants get a string value going from "ST_0" to "ST_6" respectively. These constants are required when the states and transitions are defined in the FSM class implementation.

- **GetLeaderShipTimeOut**. This is a timeout value that is used to wait for a response from other agents when the initial leadership is requested. The value assigned to this constant is 3 seconds.

- **GetLeaderShipTick, WaitLeaderNofiTick, GainedLedershipTick, WaitLeaderResignTick**. These constants are related to the period of execution of the processes defined by the GET_INITIAL_LEADERSHIP, WAIT_FOR_LEADERSHIP_NOTIFICATION, LEADERSHIP_GAINED, SEND_LEADERSHIP_RESIGN states respectively. These value are defined by 1 second and is needed based on the way in which these processes were modeled.

- **TimeCycleSecondsMax**. This constant defines the maximum cycle time in which all of the agents have the ownership once regardless of the leadership order. In this case it's defined as 300 seconds (5 minutes). This constant warranties that under normal conditions each of the agents will have the ownership again in less than this time. In the above equations this constant is referred as $CycleTime$.

- **TransitDensityMax**. This constant defines the maximum value that the $\boldsymbol{\delta_T}$ could have. The value is defined as 4095, based on the resolution of the ADC in the Galileo board (12 bits). In above equations this constant is referred as $\boldsymbol{\delta_T Max}$**.**

- **LeaderShipReqTimeBeforeLeaderFinish**. This constant defines the time in which the agents needs to request again the leadership under the leadership period. In this case, this constant is referred to the time before the leadership time expires. The value assigned to this constant is 5 seconds, meaning that 5 seconds before the current leadership ends, all the agents must re-send the new leadership request message.

- **WaitForLeadershipNotificationTimeout**. This constant defines a timeout set by the agents who didn't gain the leadership to wait for the leadership notification from the agent who actually gained it. In this case this was defined as 5 seconds.

- **WaitForLeadershipResignTimeout**. This constant defines a timeout set by the agents who didn't gain the leadership when the current leadership period is close to end. At this time is expected that the leader agent indicates the leadership resign, so this timeout is set to wait for this notification. In this case this constant was defined as 10 seconds.

The status variables used by this application layer are described next:

- **agt_state**. This variable indicates the current FSM state of the agent.

- **current_step**. This variable indicates which is the current step in the transition table in which the agent is located.

- **current_leader**. This variable indicates which agent is the current leader.

- **leadership_noti_received**. This variable indicates if the leader agent already sent the leadership gained notification to the rest of the agents during the current leadership period.

- **leadership_resign_received**. This variable indicates if the leader agent already sent the leadership resign notification to the rest of the agents during the current leadership period.

- **leadership_time**. This variable contains the current leadership period.

- **last_step_start_time**. This variable is used by the slave agents and contains the time in which the current step started. This is referred to **sem_status.TimeCount** continuous counter. This value is used to determine the duration of the current step in order to keep track of this in the step time monitor.

- **abort_processes**. This variable is a flag that indicates that a failure or invalid scenario occurred and the running processes need to be aborted.

- **send_InitialLeadershipReq**. This is a flag that indicates that the initial leadership request is running. After coming out from reset this gets a value of true, and then it's cleared just after the 2nd leadership request message is sent.

- **last_agent_in_cycle**. This variable is set to true to indicate that the leadership cycle (i.e. all the agents were already leader) is close to end and the current leader agent is the last one in the cycle.

- **Idle_bhv_active, GetLeadership_bhv_active, LeadershipGained_bhv_active, WaitForLeaderShipNoti_bhv_active, WaitForLeaderShipResign_bhv_active, EmergencyCtrl_bhv_active**. These variables are used to indicate that the processes running in each one of the FSM states are active or not. These flags are use in the case of a failure condition, in which the agent state needs to return to FSM_IDLE_ST and the current active processes need to be stopped.

- **emergency_control_st**. This indicate if the agent has received the EMERGENCY_CONTROL request message indicating that an emergency situation has occurred.

- **LeaderTrack[]**. This variable is a list that contains all the leader agents during the current cycle. It is used to determine if certain agent has already been the leader or not.

- **StepTimeMonitor[]**. This variable is a list that contains the step times during the current cycle. It is used to recalculate step times in case more than 1 agent exercises the same step.

This application layer requires to provide a method to calculate the initial leadership time for the leader agent (before adjusting each one of the step times). This is done following the equations described

previously in this chapter using the $\boldsymbol{\delta_T}$ reported by each one of the agents. For this, the **calculate_leadership_time()** method must be created. This method returns an integer value which corresponds to the leadership time. The steps required to achieve this are these:

1. Get the maximum value of the $\boldsymbol{\delta_T}$ reported.
2. Get the sum of all the $\boldsymbol{\delta_T}$ reported.
3. Get the cycle time proportional to the maximum $\boldsymbol{\delta_T}$ reported as a function of TimeCycleSecondsMax, max $\boldsymbol{\delta_T}$,TransitDensityMax.
4. Get the leadership time as a function of the local $\boldsymbol{\delta_T}$**,** the cycle time and the sum of all the $\boldsymbol{\delta_T}$**.**
5. Make sure that the leadership time is greater than MinmimumLeadershiptime.

In order to support the scenario of a failure condition the **semaphore_agent** class provides a support method that is used to reset or abort any possible running process. This method is called **restart_system()**. The functionality of this methods is described next:

1. Set to true the **sem_status.abort_processes** variable
2. Clears the leader tracking list (**sem_status.LeaderTrack**).
3. Clears the message response tracker (**msgResponsetrk**).
4. Checks if any of the FSM state processes is running, if so it must stop it.

As mentioned before, in Jade **FSMBehaviour**, the functionality implemented in each one of the states is implemented using behaviours (Jade classes). The transition between one state to another is defined by the value that these behaviours return. So in order to have a better control over the state transition it was defined an enumeration that includes all the possible valid state transitions. This enumeration is called **TransitionResults**. A variable from this type of enumeration called **transit_response** is defined inside of the **MainSemaphoreFSM** class.

The SW architecture of **MainSemaphoreFSM** class is shown in the next figure. This will be described in the following lines.

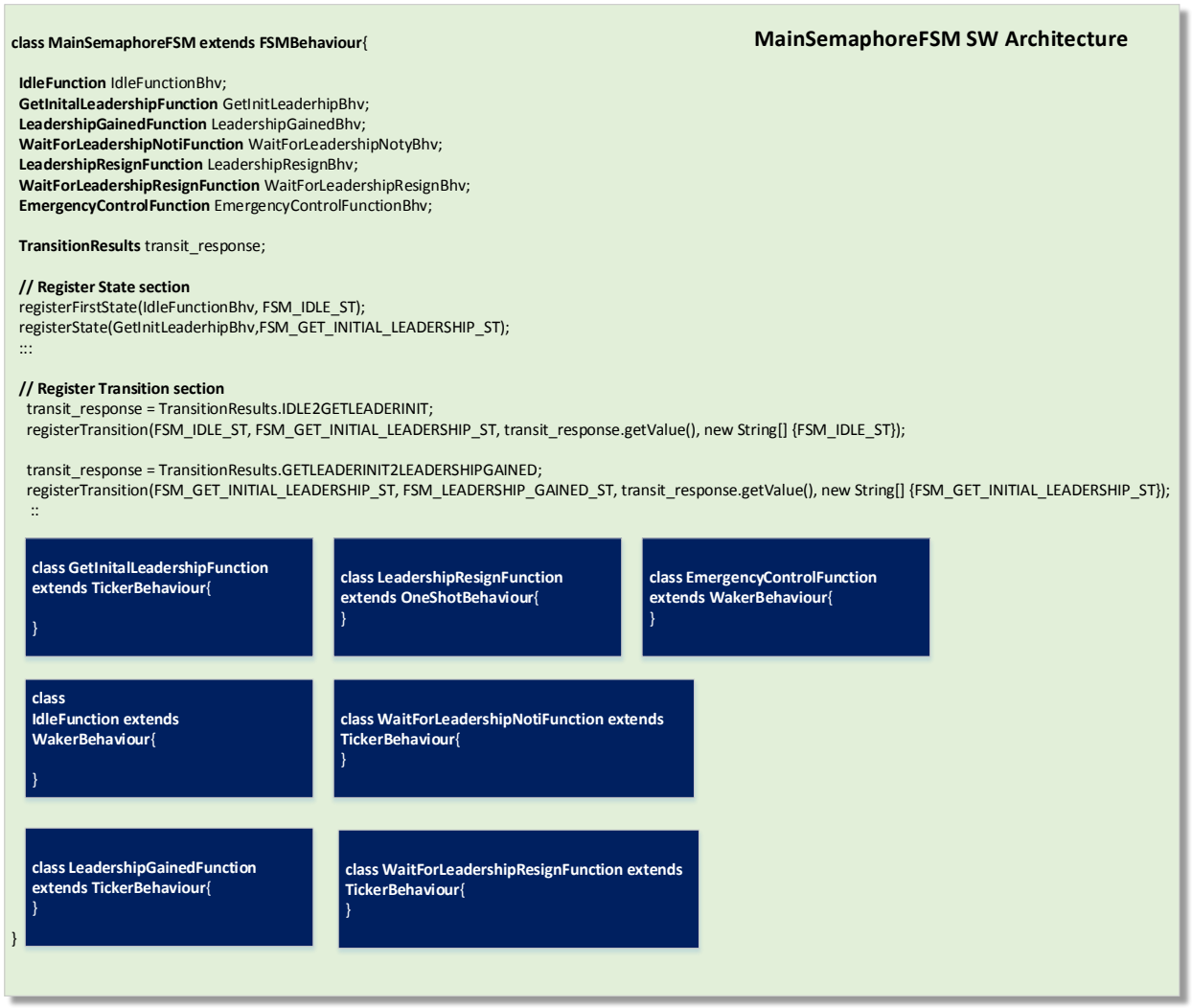


Figure 26. MainSemaphoreFSM SW Architecture.

As mentioned above, in order to create the FSM it is needed to create JADE behaviours for each one of the states. The desired functionality in each one of the states is developed inside of these behaviours. For each one of the behaviours it's required to instantiate an object from its class. It is also required to register each one of the states using **registerFirstState()** or **registerState()** functions inherited from the **FSMBehaviour** class. These functions receive 2 parameters:

1. The object created from the class that implements the behaviour for the FSM state to register.
2. The string variable that defines the state to register.

Besides of the state registration, it's also required to register all the possible state transitions as described by the FSM diagram. This is done with the **registerTransition()** method, also inherited from the **FSMBehaviour** class. This function receives 4 parameters:

1. The string variable that defines the current state.
2. The string variable that defines the state to transit to.

3. The integer value that the behaviour of the current state must return to transit to the new state. Here is where a value taken from **TransitionResults** enumeration is used to indicate the transition condition. Based on this parameter, each one of the behaviors implemented for the class must handle the integer value to return (from **TransitionResults** type) to indicate to which state it needs to transit based on the current conditions.
4. A list of string variables that indicates which states must be reset when the state transition occurs.

The implementation details of each one of the behaviours required for this FSM is described next.

FSM_IDLE_ST Behaviour Implementation

For this behaviour, the **IdleFunction** class must be created. This one is extended from the **WakerBehaviour**. This type of behavior is executed after sometime than this is started. This time is provided as a parameter for the constructor, but this can be modified calling the **reset()** method.
This behaviour is executed when the agent comes out from reset or after detecting an invalid scenario. The main function of this behaviour is to give some time for the rest areas of the agent to get a steady condition, before start executing the traffic light functionality.
For this project, it is required to implement the **onStart()**, **onWake()** and **onEnd()** methods as described next, and shown in the following figure.

- **onStart() method**. This method is executed by the time the behaviour is started. The implementation details are listed here:
    1. Update the status variables accordingly to the current state. As this is the IDLE and initial start, the variables must be set to the default values. This includes set in true the **Idle_bhv_active** variable to indicate that this process is active.
    2. Update the lamps states to match with the step 0 in the transition table.
    3. If it's detected that the Emergency Control process is active (meaning that the emergency control process was executed and finished and after this it jumped into IDLE state) the waiting time must be reduced to get only one $\boldsymbol{\delta_T}$ sample. If this is not the case, then wait for 3 samples. In the first case only once sample is required because the sample process remains active even in emergency control state and it's not required to wait to get a steady $\boldsymbol{\delta_T}$ value.
    4. If it gets into IDLE state having the **abort_processes** status variable active, it's necessary to restart the system killing any of active process running.
    5. If it get into IDLE state when the current agent it's still the leader, this needs to notify to the other agents that need to set their lamps to match the IDLE state. This means that the **STEP_NOTIFICTION** message must be sent.
    6. Clear the leader indicator lamp to indicate that this agent is not the leader. Also clear the **current_leader** status variable to indicate an invalid leader agent.
    7. Clear the **last_agent_in_cycle** status variable and also clear the information in the step time tracker.

- **onWake method**. This method is executed by the time the timeout set when created or when reset expires. In this method the next status variables are updated: **abort_processes**(false) and **leadership_noti_received**(false). This is used to indicate that the abort process was completed if was the case and also to indicate that the leadership has not been assigned yet to any agent.

IdleFunction Behaviour (WakerBehaviour)

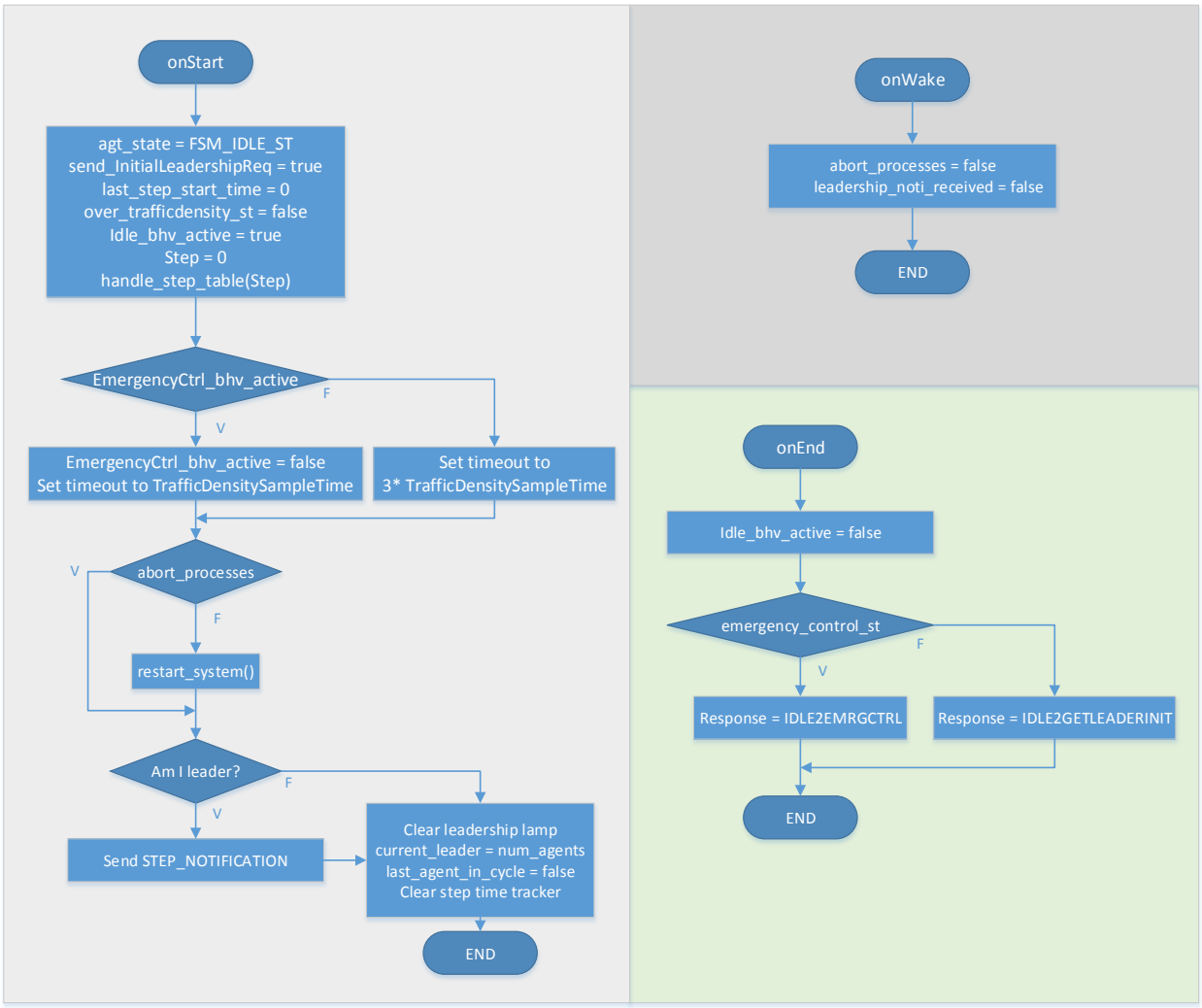


Figure 27. IdleFunction Behaviour Flow Diagram.

- **onEnd method**. This method is executed by the time the process is finished. In this behaviour it's executed just after the **onWake()** method is executed. In this method it's indicated that this process is no longer running (**Idle_bhv_active**). Also it's required to indicate to which state it's required to transit to. In this case, it's only possible to move to 2 states. This is done generating an integer response in this method using the **TransitionResults** enumeration. If it's detected that the **EMERGENCY_CONTROL** message was received (**emergency_control_st** status variable is in true) then move to **FSM_EMERGENCY_CONTROL_ST**, otherwise move to **FSM_GET_INITIAL_LEADERSHIP_ST**.

FSM_GET_INITIAL_LEADERSHIP_ST Behaviour Implementation.

For this behaviour the **GetInitalLeadershipFunction** class must be implemented. This class is extended from the **TickerBehaviour** class. In this type of behaviour the method **onTick()** is executed periodically. This period is defined by the parameter that is provided to the constructor of this class. As indicated before, while active this is executed every 1 second as defined by **GetLeaderShipTick** constant.
This behaviour is executed only once after passing from **FSM_IDLE_ST** state. In this behaviour, the agent will send the **LEADERSHIP_REQUEST** message to other agents and will wait for the response of them setting a timeout for this. If the responses are not obtained, the agent must to repeat the process of sending the **LEADERSHIP_REQUEST** message. Once that the agent receives responses from all the agents it must check if the leadership was accepted or rejected and based on this it must decide if moving to the **FSM_LEADERSHIP_GAINED_ST** or **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state.
For this project, it is required to implement the **onStart()**, **onTick()** and **onEnd()** methods as described next, and shown in the following figure:

- **onStart() method.** This method is executed by the time the behaviour is started. The implementation details are listed here:
  1. Update status variables accordingly to this state. The variables to update are agt_state (FSM_GET_INITIAL_LEADERSHIP_ST), GetLeadership_bhv_active(True), last_step_start_time (0), last_agent_in_cycle(false). Also clear the step time information in the step time tracker.
  2. Calculate and set the timeout to get a response from **REQUEST_LEADERSHIP**. This timeout is set using **GetLeaderShipTimeOut** and copy this into a counter variable. The timeout will represent the maximum number of seconds that this process will remain active waiting for a response.
  3. Check if the LEADERSHIP_NOTIFICATION message has been received. If this is the case, stop the process, otherwise send the REQUEST_LEADERSHIP message to other agents along with the local $\boldsymbol{\delta_T}$.

- **onTick() method.** As mentioned before, this method is executed periodically until the process is stopped. In this case this is executed every 1 second. When this method is executed, it will decrease the timeout variable. It will check if the variable has already a value of 0 or if all the agents provided a response to the **REQUEST_LEADERSHIP** message and the current agent received all the proposal from the other agents. In any of these cases the behaviour is stopped. When the behaviour is stopped, the **onEnd()** method is executed. The 2nd checking is used to warranty that the agent won't leave this state until it has all the $\boldsymbol{\delta_T}$ form the other agents and also know if its request was accepted or rejected.

- **onEnd() method.** This function is executed when the timeout set to receive all agents' responses for **REQUEST_LEADERSHIP** message expires or when all the agents already responded to this message. In this function the agent clears the function that indicate that this process is active, **GetLeadership_bhv_active**(false). Also, the agent must determine which will be the next state it needs to move to. For this several conditions are checked:

    1. If **abort_processes** status variable is set to TRUE, it means that an invalid scenario happened. So in this case it needs to check if the **EMERGENCY_CONTROL** message was received using **emergency_control_st** status variable. If this is set, the response must be set to **GETLEADERINIT2EMRGCTRL**, indicating that needs to move to **FSM_EMERGENCY_CONTROL_ST**. If **emergency_control_st** is set to false, then it needs to move to **FSM_IDLE_ST** setting the response to **GETLEADERINIT2IDLE**.

2. If **abort_processes** status variable is set to FALSE, it needs to check if the **LEADERSHIP_NOTIFICATION** message has been received. If this is the case, then this means that some of the agents already gain the leadership so the current agent must become a slave agent. In this case the agent needs to move to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state setting as response **GETLEADERINIT2WAITFORNOTILEADERSHIP**.
3. If **abort_processes** status variable is set to FALSE and the **LEADERSHIP_NOTIFICATION** message has not been received yet, then it needs to check if all the agents responded to the **REQUEST_LEADERSHIP** message. If this is the case, it needs to check if all agents accepted this message. If this is the case, this agent will be the next leader, so it must respond **GETLEADERINIT2LEADERSHIPGAINED** indicating that it needs to move to **FSM_LEADERSHIP_GAINED_ST** state. If any of the agents didn't accept the request, it means that the any other agent has more priority to be the leader agent so the agent then needs to respond **GETLEADERINIT2WAITFORNOTILEADERSHIP** indicating that it needs to move to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state.

GetInitalLeadershipFunction Behaviour (TickerBehaviour)

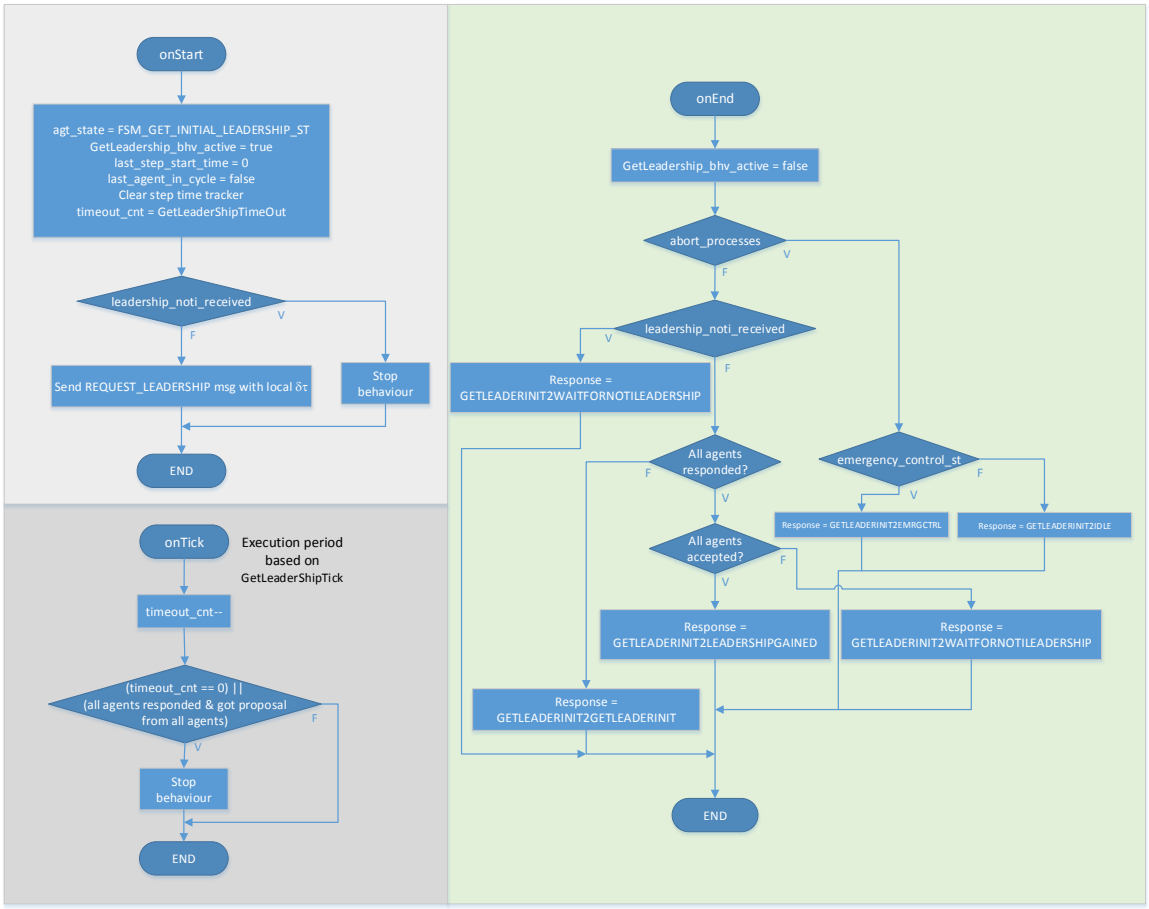


Figure 28. GetInitialLeadershipFunction Behaviour Flow Diagram.

4. If **abort_processes** status variable is set to FALSE and the **LEADERSHIP_NOTIFICATION** message has not been received yet and it was found that any of the agents didn't respond

to the **REQUEST_LEADERSHIP** message, this means that any of the agents lost communication or is not in the appropriate state to respond, so this agent needs to respond with **GETLEADERINIT2GETLEADERINIT** indicating that the agent needs to remain in **FSM_GET_INITIAL_LEADERSHIP_ST** state and repeat the full process since **onStart()** method giving chance to the agent that didn't respond to the **REQUEST_LEADERSHIP** message to respond under a new request.

FSM_LEADERSHIP_GAINED_ST Behaviour Implementation.

For this behaviour the **LeadershipGainedFunction** class must be implemented. This class is extended from the **TickerBehaviour** class. In this type of behaviour the method **onTick()** is executed periodically. This period is defined by the parameter that is provided to the constructor of this class. As indicated before, this is executed every 1 second as defined by **GainedLedershipTick** constant.
The agent will execute this behavior once that this gains the leadership and also once that all the other agents send to this agent the leadership request message, so this agent can know the traffic load that they report. Based on this data, the agent must calculate the leadership time and the time in each one of the steps from the transition table that applies for this leadership period. Also, the agent must send the **LEADERSHIP_NOTIFICATION** message and also send the **STEP_NOTIFICATION** messages for each step to other agents setting for this a series of timeouts. Finally the agent also must calculate the time to re-send the new **LEADERSHIP_REQUEST** message and after this time it must send it. For this, it's required to set another timeout.
For this project, it is required to implement the **onStart()**, **onTick()** and **onEnd()** methods as described next, and shown in the following figures.

- **onStart() method**. In this method the next steps must be implemented:
  1. Update the status variables accordingly with the current state: **agt_state** (**FSM_LEADERSHIP_GAINED_ST**); as this agent enter in to this state it means that this is the current leader, so the **current_leader** must be updated; **LeadershipGained_bhv_active** (true) indicates that this process is active.
  2. Obtain the list of steps for this leader using the configuration object **get_next_steps_for_leader()** method.
  3. Calculate the minimum leadership time based on the number of steps for this leadership period multiplied by the minimum step time.
  4. Obtain the initial leadership time using the **calculate_leadership_time()** support method.
  5. Adjust the leadership time subtracting the time of overlapped steps between the current and last leadership period. Only do this if no overload condition is detected.
  6. Clear the content of the message response tracker.
  7. Get the step time list using the **get_steps_time()** method from the configuration object, using the step list and the leadership time.
  8. Adjust step time list subtracting the time of the steps that has been already executed in this cycle using the information of the step time tracker. Only do this if no overload condition is detected. After updating the step list, re-calculate the leadership time adding the time in the step time list.
  9. Send the **LEADERSHIP_NOTIFICATION** message to other agents, indicating the current state of the agent and the leadership time.
  10. Indicate that this agent has the leadership turning on the leadership lamp in the Galileo board.

11. Set a timeout to start sending the notification of the 1st step to the rest of the agents. For this, the continuous time counter (**TimeCount**, with reference of 1 second) is used. A variable called **time_counter_expire** is used to register this time. Initially a period of 2 seconds is given to start sending this message. This 2 seconds period is used to warranty that the agents process the **LEADERSHIP_NOTIFICATION** message.
12. In this behaviour, the next **LEADERSHIP_REQUEST** message must be sent. For this is needed to determine the time (also based on the **TimeCount**) in which this agent needs to re-send this message. This must be sent **LeaderShipReqTimeBeforeLeaderFinish** seconds before the end of the leadership period. For this timeout the **leadership_request_time** variable is used. Also, a variable called **leadership_request_sent** must be created as a flag to indicate if the message has been sent or not. Initially this must be set as false.
13. Set **iteration** variable in 0.

- **onTick method**. This method is executed periodically based on **GainedLedershipTick**. Every time that the method is executed it will need to check if the timeout variables configured in the **onStart()** method matches with the continuous time counter (**TimeCount**). This is done in stages:
    1. Check if the **time_counter_expire** variable matches with the **TimeCount** variable. If this is the case, a step is taken from the step list using as index the **iteration variable**. Using this information, the **STEP_NOTIFICATION** message is sent to the agents to make them transit to the new step. Also, this agent needs to process this step to update its lamps state. The step time is also obtained from the step time list using as index the **iteration** variable and with this value the **time_counter_expire** variable is updated. If the process already cover all the steps in the step list and it covered all the step times, then the agent must stop the behaviour. But if there are more steps in the list, then the **iteration** variable is increased by 1. When the behaviour is stopped, the **onEnd()** method is executed.
    2. Check if the **leadership_request_sent** variable is in false and if the **leadership_request_time** variable matches with the value of **TimeCount** status variable. If this is the case, then it means that the **LEADERSHIP_REQUEST** message must be re-send using the current $\boldsymbol{\delta_T}$ value and the current state. Also the **send_InitialLeadershipReq** status variable must be set to false to indicate that the process of sending the 1st **LEADERSHIP_REQUEST** message has finished at all. Also the **leadership_request_sent** variable needs to be set to True.

LeadershipGainedFunction Behaviour (TickerBehaviour)

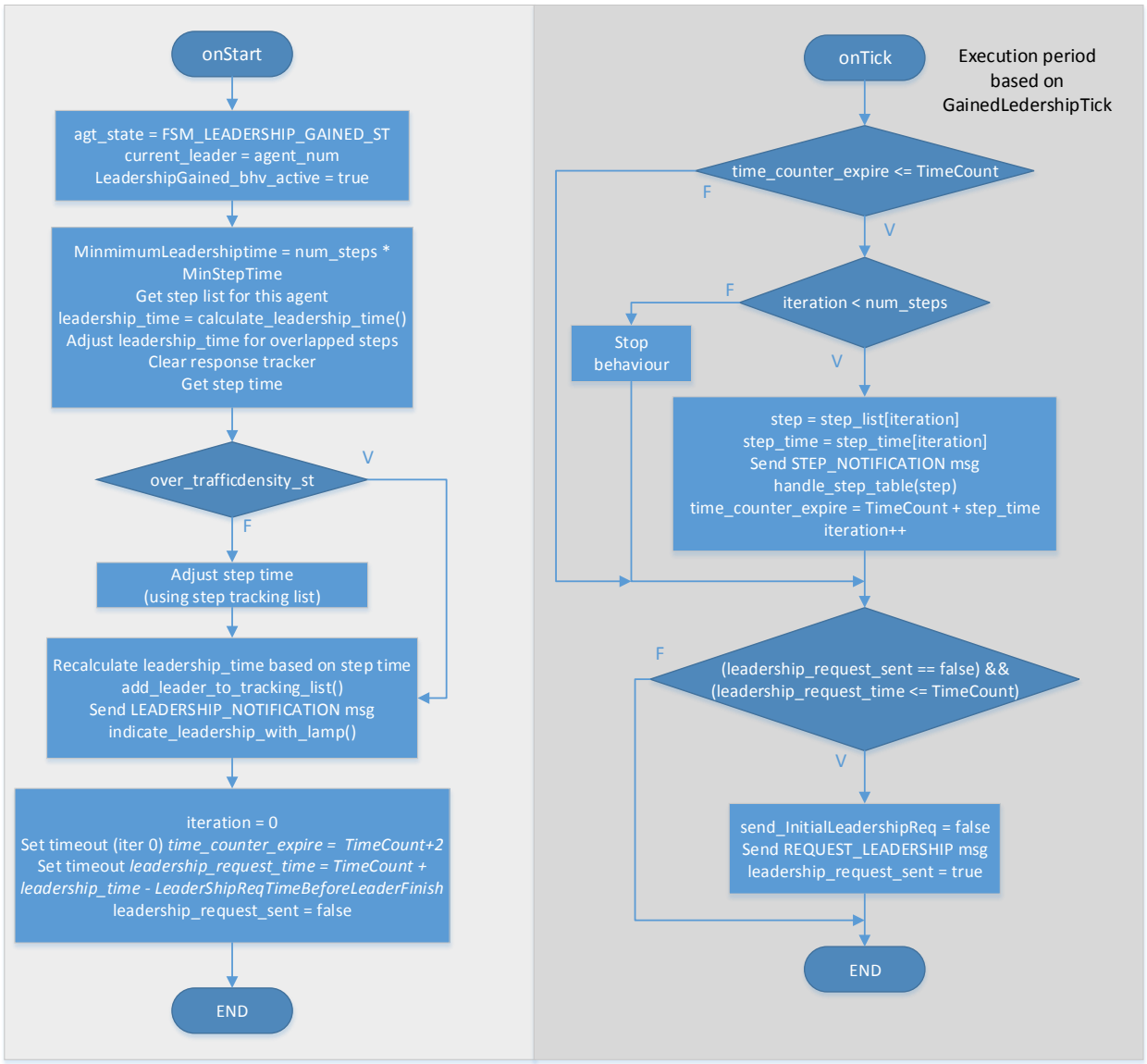


Figure 29. LeadershipGainedFunction Behaviour Flow Diagram (Part 1).

- **onEnd() method**. The agent must execute this method when this already sent all the **STEP_NOTIFICATION** messages for this leadership period and re-sent the new **LEADERSHIP_REQUEST** message. The step that the agent must follow in this method are described next:

  1. Update the **LeadershipGained_bhv_active** status variable to False to indicate that this process is no longer active.
  2. If the **abort_processes** status variable is True, it means that an abnormal scenario occurred. So it needs to check if the **EMERGENCY_CONTROL** message was received (with **emergency_control_st** status variable). If this is the case, this needs to respond with **LEADERSHIPGAINED2EMRGCTRL** indicating that it must transit to

FSM_EMERGENCY_CONTROL_ST. If the **EMERGENCY_CONTROL** message was not received, then it needs to move to **FSM_IDLE_ST** responding **LEADERSHIPGAINED2IDLE**. This is needed as an invalid condition occurred and all the processes needs to be restarted.

3. If the **abort_processes** status variable is False, the agent must check if all the other agents responded to the **LEADERSHIP_REQUEST** message. If this is the case, then the agent must transit to **FSM_SEND_LEADERSHIP_RESIGN_ST** to give the leadership to another agent. To transit to this state this needs to respond with L**EADERSHIPGAINED2LEADERSHIPRESIGN**.

LeadershipGainedFunction Behaviour (TickerBehaviour)

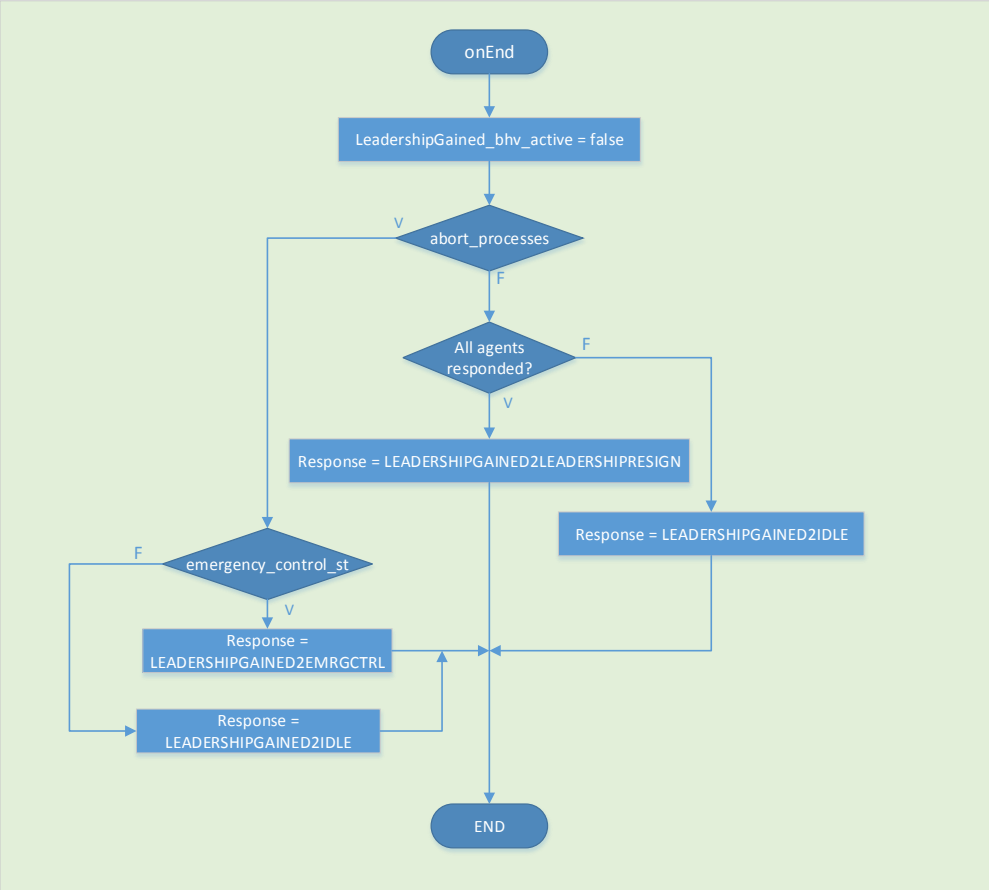


Figure 30. LeadershipGainedFunction Behaviour Flow Diagram (Part 2).

FSM_SEND_LEADERSHIP_RESIGN_ST Behaviour Implementation.

For this behaviour the **LeadershipResignFunction** class must be implemented. This class is extended from the **OneShotBehaviour** Jade class.
The basic functionality of this behaviour is to check if this agent will be the next leader agent or not depending on the responses of the other agents to the **LEADERSHIP_REQUEST** message sent previously. Depending on these responses, the agent must transit either to **FSM_LEADERSHIP_GAINED_ST** or **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state.

In this project, this class must implement the **action()** and **onEnd()** methods. In this class the **action()** method is executed immediately when the behavior is started. Some implementation details of these functions are described and shown next.

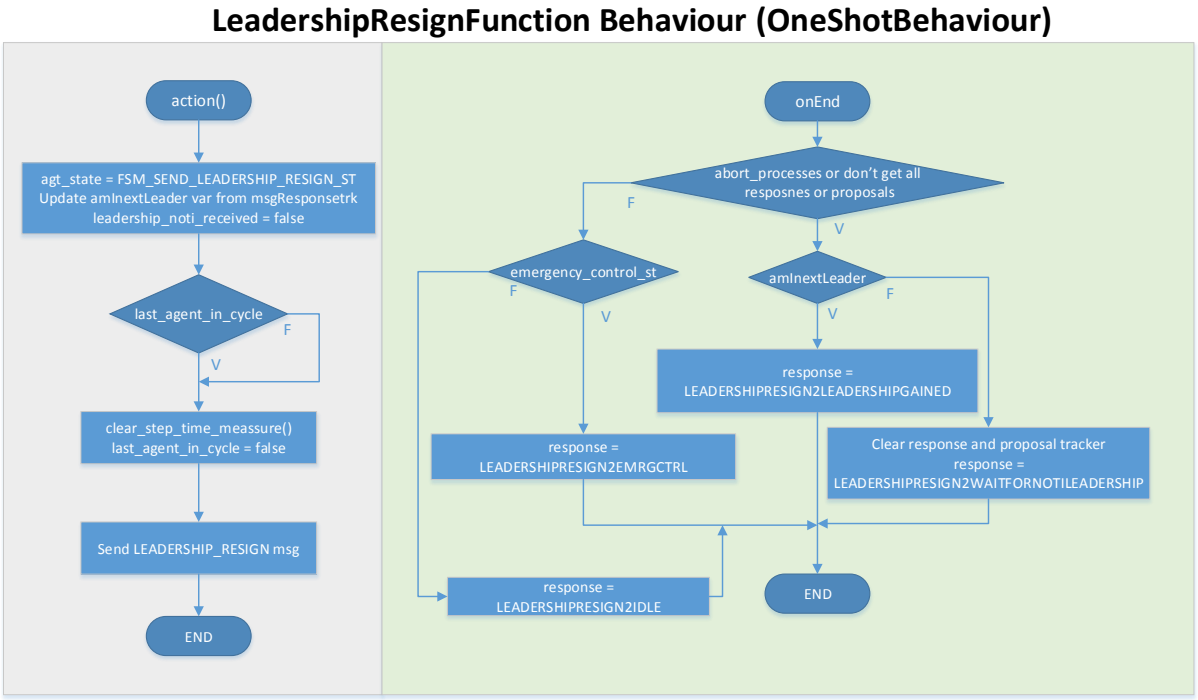


Figure 31. LeadershipResignFunction Behaviour Flow Diagram.

- **action() method**. In these method the following steps must be followed:

  1. Update the **agt_state** status variable with **FSM_SEND_LEADERSHIP_RESIGN_ST**.
  2. Check in the response tracker if all the agents accepted the **LEADERSHIP_REQUEST** message. Store the result in the **amInextLeader** Boolean variable.
  3. Set to false the **leadership_noti_received** status variable, so it makes sure that this is false by the time the next leader agent sends the **LEADERSHIP_NOTIFICATION** message.
  4. If this agent is the last leader in the cycle (check **last_agent_in_cycle** status variable) then it must clear the step time monitor and clear the **last_agent_in_cycle** status variable.
  5. Send the **LEADERSHIP_RESIGN** message, along with the current state of the agent.

  In this behaviour, the **onEnd()** method is executed just after the **action()** method finishes.

- **onEnd() method.** In this method these steps are being executed:

  1. If **abort_processes** status variable is True, it must check if the **EMERGENCY_CONTROL** message was received (with **emergency_control_st** status variable). If this is true the agent must transit to **FSM_EMERGENCY_CONTROL_ST** responding **LEADERSHIPRESIGN2EMRGCTRL**. If the message was not received, then it must transit to **FSM_IDLE_ST** responding **LEADERSHIPRESIGN2IDLE**.

2. If **abort_processes** status variable is False, it must check if this agent will be the leader agent looking at the **amInextLeader** variable updated in the **action()** method. If this is the case, the agent must return to the **FSM_LEADERSHIP_GAINED_ST** state responding **LEADERSHIPRESIGN2LEADERSHIPGAINED**. IF the current agent won't be the next leader it must transit to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** responding **LEADERSHIPRESIGN2WAITFORNOTILEADERSHIP** and also it must clear the message response tracker to let it empty and ready to track new responses for the next **LEADERSHIP_REQUEST** messages. In other to take any of these 2 possible paths the agent must check that it already received all responses and proposal from other agents, so the decision can be done correctly.

FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST Behaviour Implementation.

For this behaviour the **WaitForLeadershipNotiFunction** class must be implemented. This class is extended from the **TickerBehaviour** class. In this type of behaviour the method **onTick()** is executed periodically. This period is defined by the parameter that is provided to the constructor of this class. As indicated before, this is executed every 1 second as defined by the **WaitLeaderNofiTick** constant. In this behaviour the agent will wait for the **LEADER_NOTIFICATION** message from the leader agent. For this, the agent sets a timeout to receive this message, and in case it's not received, then this is considered an invalid condition and the agent must be restarted. In this class also is set a 2nd timeout used to send the new **LEADERSHIP_REQUEST** message few time before the current leadership period ends.
For this project, it is required to implement the **onStart()**, **onTick()** and **onEnd()** methods as described next, and shown in the following figure.

- **onStart() method**. This method is executed just after that the behavior is started. In this method the **agt_state** status variable is update with **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** state. Also a timeout is programmed to wait for the LEADERSHIP_NOTIFICATION message. This timeout will expire after the time indicated by **WaitForLeadershipNotificationTimeout**. The expiration time is stored in the **leader_notification_timeout_limit** variable. In this method it's also indicated that there is a process running for the timeout signaled with the **WaitForLeaderShipNoti_bhv_active** status variable. Finally a flag variable must be created to indicate if the **LEADERSHIP_NOTIFICATION** message was received and the time to re-send the new **LEADERSHIP_REQUEST** message has been calculated (this is obtained using the information that comes with the **LEADERSHIP_NOTIFICATION** message). This variable is called **leadership_time_obtained** and must be set to False.

- **onTick() method**. This method is executed periodically as defined by the **WaitLeaderNofiTick** constant. The steps executed in this method are described next:

  1. It must check if the **LEADERSHIP_NOTIFICATION** timeout has expired. This is done checking the condition if the **leadership_noti_received** variable is false and if the current time has reached the timeout set for this (stored in **leader_notification_timeout_limit**). If the timeout expires under this scenario then the behaviour must be stoped.
  2. It must check if the **LEADERSHIP_NOTIFICATION** message has been received using the **leadership_noti_received** status variable. This is done only while the **leadership_time_obtained** flag is still in false. If this condition is achieved, then the agent must indicate that this doesn't have the leadership anymore using the corresponding indicator lamp. Also it must set to True the **leadership_time_obtained** flag and must

determine the time in which it needs to re-send the **LEADERSHIP_REQUEST** message. This time must be stored in the **leadership_request_time** variable and it's calculated adding to the current time the leadership time (**leadership_time** status variable) and subtracting the value in **LeaderShipReqTimeBeforeLeaderFinish**. Basically the idea of this is to re-send the message few seconds before the leadership expires.

3. If the time to re-send the **LEADERSHIP_REQUEST** message has been calculated (indicated by the **leadership_time_obtained** flag) it must check if the current time reaches the timeout set for this (stored in **leadership_request_time** variable). If this is the case, the agent must clear the **send_InitialLeadershipReq** status variable indicating that the full process related to sending the very first **LEADERSHIP_REQUEST** message has completed. Also the agent must re-send the **LEADERSHIP_REQUEST** message with the current $\boldsymbol{\delta_T}$ value and finally must stop the current behaviour.

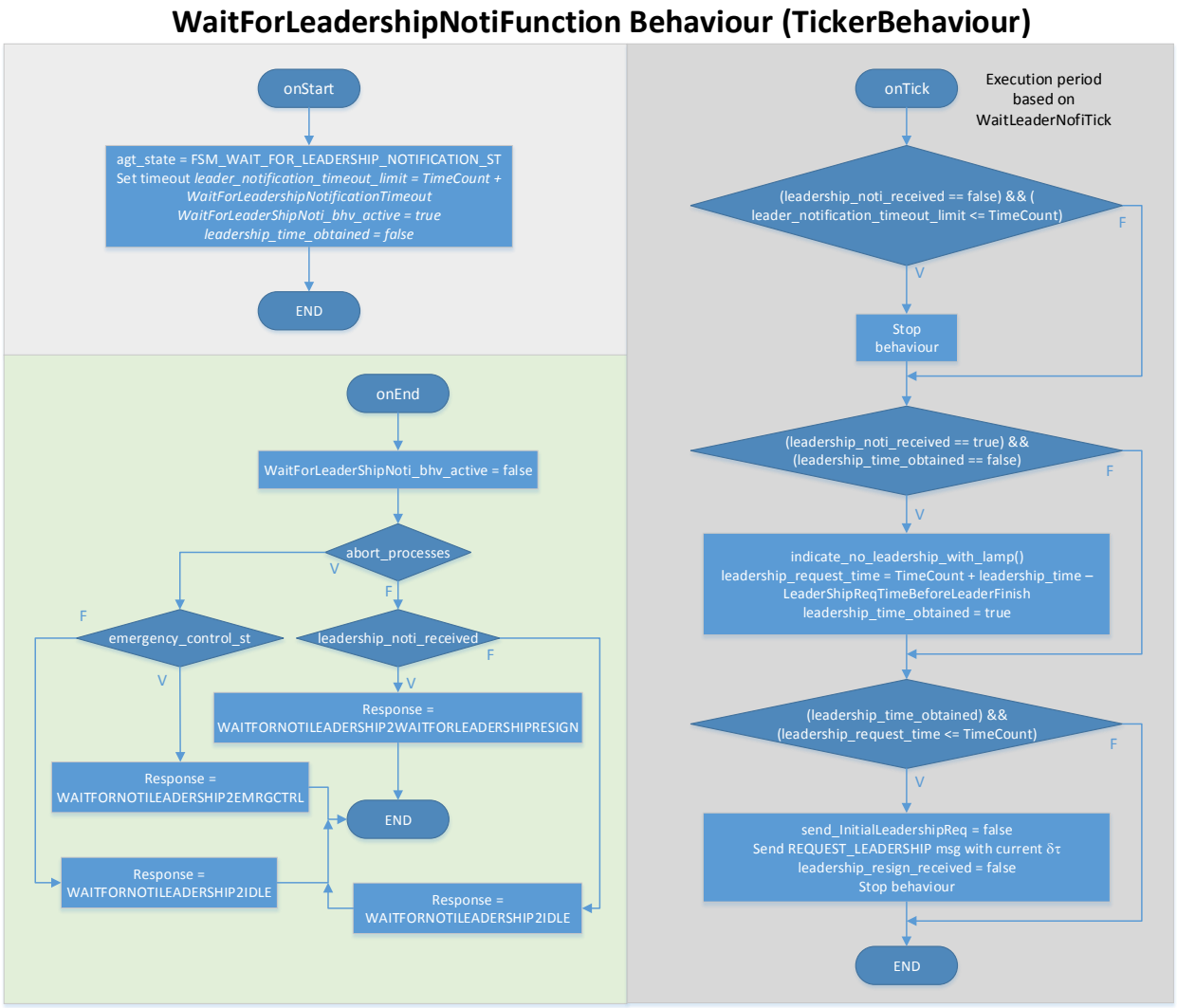


Figure 32. WaitForLeadershipNotiFunction Behaviour Flow Diagram.

- **onEnd() method**. This method is executed once this behaviour is stopped. In this method is decided to which state the agent must transit to. The steps that must be followed are described next:

    1. It must indicate that the process running in this behaviour is over clearing the **WaitForLeaderShipNoti_bhv_active** status variable.
    2. If **abort_processes** status variable is True, it must check if the **EMERGENCY_CONTROL** message was received (with **emergency_control_st** status variable). If this is true the agent must transit to **FSM_EMERGENCY_CONTROL_ST** responding with **WAITFORNOTILEADERSHIP2EMRGCTRL**. If the message was not received, then it must transit to **FSM_IDLE_ST** responding with **WAITFORNOTILEADERSHIP2IDLE**.
    3. If **abort_processes** status variable is False, it must check if the **LEADERSHIP_NOTIFICATION** message has been received (registered in the **leadership_noti_received** status variable). If this is the case, then the agent must transit to the **FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST** state responding with **WAITFORNOTILEADERSHIP2WAITFORLEADERSHIPRESIGN**. If the message was not received it must transit to **FSM_IDLE_ST** responding with **WAITFORNOTILEADERSHIP2IDLE**.

FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST Behaviour Implementation.

For this behaviour the **WaitForLeadershipResignFunction** class must be implemented. This class is extended from the **TickerBehaviour** class. In this type of behaviour the method **onTick()** is executed periodically. This period is defined by the parameter that is provided to the constructor of this class. As indicated before, this is executed every 1 second as defined by the **WaitLeaderResignTick** constant. In this behaviour the agent will wait for the **LEADERSHIP_RESIGN** message from the still leader agent. In this behaviour a timeout is being set receive this message. In this behaviour it's also decided to which the state the agent must transit, either to **FSM_IDLE_ST**, **FSM_LEADERSHIP_GAINED_ST** or **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** depending on the reception or not of the **LEADERSHIP_RESIGN** message and the responses by the other agents for the **LEADERSHIP_REQUEST** message.
For this project, it is required to implement the **onStart()**, **onTick()** and **onEnd()** methods as described next, and shown in the following figure.

- **onStart() method**. In this method the **agt_state** status variable is updated to reflect the current state **FSM_WAIT_FOR_LEADERSHIP_RESIGN_ST**. Also, the timeout for the reception of the **LEADERSHIP_RESIGN** message. This timeout is defined to expire in the time defined by **WaitForLeadershipResignTimeout**. Also, it is being cleared the **leadership_noti_received** status variable because by the time the agent receives the **LEADERSHIP_RESIGN** message the **LEADERSHIP_NOTIFICATION** message should had been already received.

- **onTick() method**. In this method is checked if the timeout set to wait for the **LEADERSHIP_RESIGN** message already expired or if the **LEADERSHIP_RESIGN** message was already received. In any case, the behaviour is stopped and the **onEnd()** method is executed. In the case of checking for **LEADERSHIP_RESIGN** message has been received, also it's needed to check if the agent already received all the leadership proposals. This method is executed periodically based on the **WaitLeaderResignTick** constant.

- **onEnd() method**. In this method the next steps are being executed:

    1. Clear the **WaitForLeaderShipResign_bhv_active** status variable, as indication that this process is over.
    2. Check if the process has been aborted using the **abort_processes** status variable. If this was aborted it must check if the **EMERGENCY_CONTROL** message was received looking at **emergency_control_st** status variable. If the message was received the agent must transit to **FSM_EMERGENCY_CONTROL_ST** state responding with **WAITFORLEADERSHIPRESIGN2EMRGCTRL**. If the message is not received the agent must transit to **FSM_IDLE_ST** state responding with **WAITFORLEADERSHIPRESIGN2IDLE**.
    3. If the process was not aborted the agent must check if the **LEADERSHIP_RESIGN** message was received together with the check of all the agents responded to the **LEADERSHIP_REQUEST** and all the agents already sent its leadership proposal message. If so, it must check if all the agents accepted the **LEADERSHIP_REQUEST** message. If this is the case the agent must transit to **FSM_LEADERSHIP_GAINED_ST** responding with **WAITFORLEADERSHIPRESIGN2LEADERSHIPGAINED**.
    4. If it was found that process was not aborted, the **LEADERSHIP_RESIGN** message was received and any of the agents didn't accept the **LEADERSHIP_REQUEST** message, the agent must clear the response tracker and it must transit to **FSM_WAIT_FOR_LEADERSHIP_NOTIFICATION_ST** responding with **WAITFORLEADERSHIPRESIGN2WAITFORNOTILEADERSHIP**.
    5. If it's found that either the LEADERSHIP_RESIGN message was not received or any of the agents didn't respond to the **LEADERSHIP_REQUEST** message, the agent must clear the response tracker and it must transit to **FSM_IDLE_ST** state responding with **WAITFORLEADERSHIPRESIGN2IDLE**.

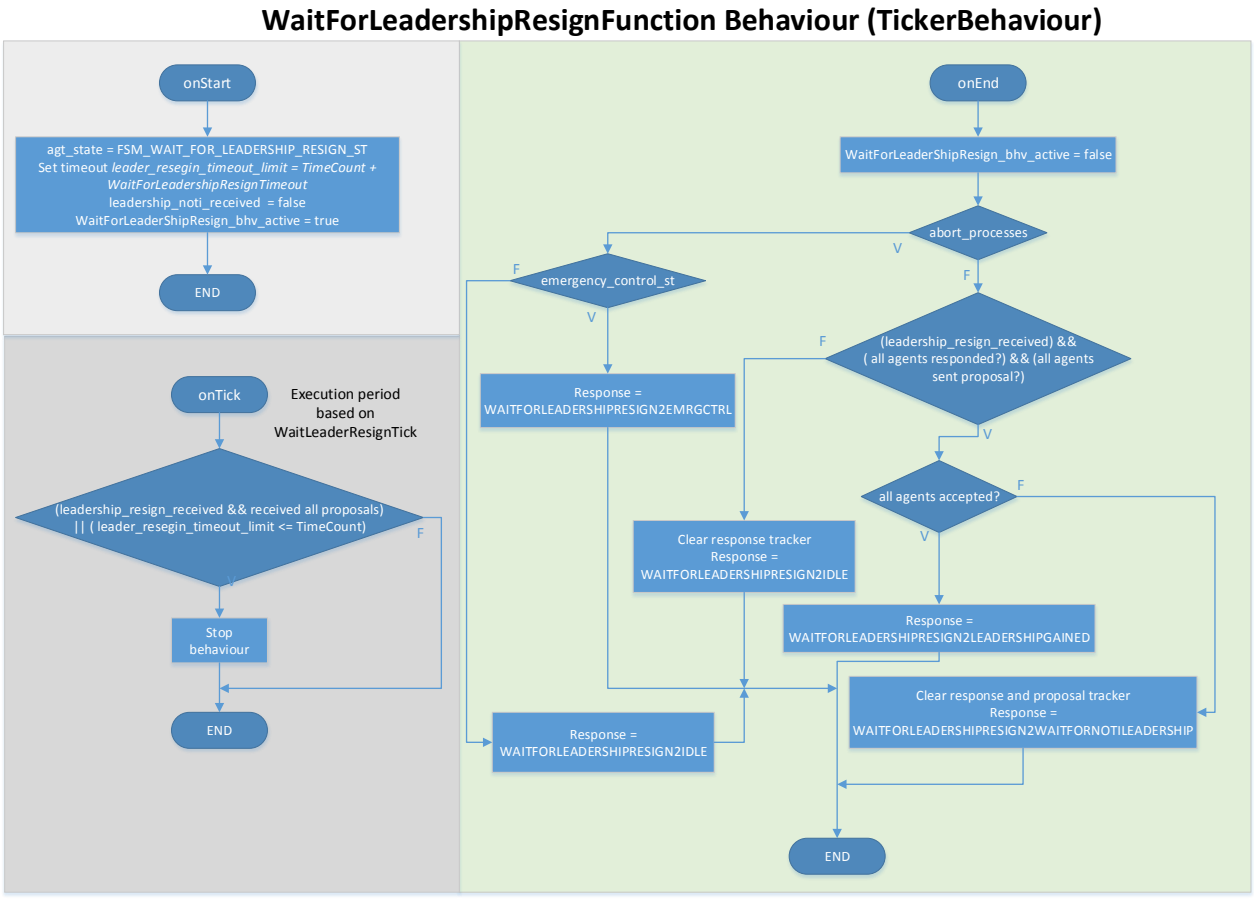


Figure 33. WaitForLeadershipResignFunction Behavior Flow Diagram.

FSM_EMERGENCY_CONTROL_ST Behaviour Implementation.

For this behaviour, the **EmergencyControlFunction** class must be created. This one is extended from the **WakerBehaviour**. This type of behavior is executed after sometime than this is started. This time is provided as a parameter for the constructor, but this can be modified calling the **reset()** method.
This behaviour is executed after receiving the **EMERGENCY_CONTROL** message. When this message is received, any running process is aborted giving priority to execute this behaviour. When this process is started it receives as parameters the lamp step to transit and the duration of this. Once that this time elapses, the agent could transit either to **FSM_IDLE_ST** or remain in the **FSM_EMERGENCY_CONTROL_ST** state.
For this project, it is required to implement the **onStart()**, **onWake()** and **onEnd()** methods as described next, and shown in the following figure.

- **onStart() method**. In this method the next steps are required to be implemented:
  1. The **agt_state** status variable is updated accordingly to the current state (**FSM_EMERGENCY_CONTROL_ST**).
  2. It's indicated that there is an active process related to this state, indicated with the **EmergencyCtrl_bhv_active** status variable.

3. The time to wake is re-stablished or reset based on the parameter provided in the **EMERGENCY_CONTROL** message and stored in the **emergency_time** status variable.
4. It's indicated that the current agent doesn't have the leadership anymore using the leadership indicator lamp. Also the **current_leader** status variable gets updated to indicate that none of the agents in the system has the leadership. Any value different from the valid agents in the system can be assigned.
5. Update the lamp state of the current agent accordingly with the parameter provided with the **EMERGENCY_CONTROL** message and stored in **emergency_step** status variable.

- **onWake() method**. This method is executed once the time set in **onStart()** method (defined by the **emergency_time** status variable) expires. In this method the **abort_processes** variable must be cleared to indicate that any process abortion was finished. Also the **emergency_control_st** status variable must be cleared to indicate that the emergency has been already processed.

- **onEnd() method**. In this method it's checked if it must transit to **FSM_IDLE_ST** or remain in this state. In case that a new **EMERGENCY_CONTROL** state arrives, the current process is aborted and a new one is started. So in this method the **emergency_control_st** status variable must be checked. If this is true, then the agent must remain in **FSM_EMERGENCY_CONTROL_ST** responding with **EMRGCTRL2EMRGCTRL**. If this is not the case, then it must transit to **FSM_IDLE_ST** state, responding **EMRGCTRL2IDLE**.

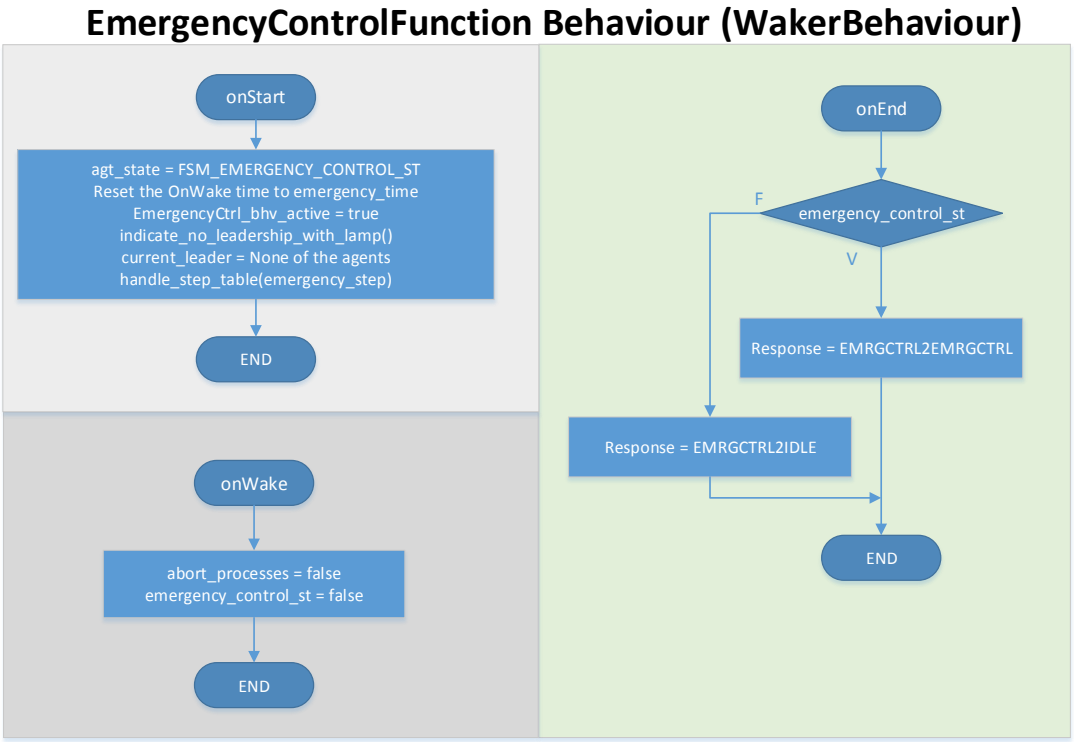


Figure 34. EmergencyControlFunction Behaviour Flow Diagram.

RE-START SUPPORT

In several sections it was mentioned the ability of the system to self-recover from an unexpected failure. Some examples of these failures are:

- Communication loss in one or more agents.
- Timeout expiration when expecting certain condition.
- Reception of an invalid step transition.
- Recovery from an emergency condition.

In order to support all these cases it's required to implement the **restart_system()** method under the **Semaphore_agent** class. This method is mentioned several times in the software specification of the system, but so far it has not described the whole functionality of this. So, next is listed the tasks that are implemented inside this method:

1. Sets the **abort_processes** status flag to indicate to all the running process that a failure has occurred and they need to be aborted.
2. Clear the leader track list and the responses and proposal tracking list.
3. Check if any of the **Behaviours** created in the application are running, using the respective status flag created for each one of them (**<behaviour>_bhv_active**). If it's found that any of these processes are running, then they need to be stopped. When they get stopped, in its **onEnd()** method they must check for the **abort_processes** status flag and take the appropriate action to quit from that behaviour cleanly.

After the completion of this method, it's expected that the **Main Agent FSM** transits to **FSM_IDLE_ST** state where the full initialization process should be executed.

Chapter IV. RESULTS

Having defined the architecture of the proposed traffic controller system and provided the implementation details of this, in this chapter it is exposed some of the results obtained from the system testing, covering the different operation modes and features supported. In this chapter also will be shown the instructions needed to start executing the application in each one of the Galileo board and also to enable the main container. It will also be shown some of the tools that the Jade GUI interface provides for the analysis of the system.

Compiling the Application

In order to compile the full traffic light controller agent (including all the software layers) it is required to use the **javac** command, listing all the java files that integrate the application. As indicated before, in order to achieve this, it's necessary to have already installed the Java virtual machine as specified in the chapter 2. So, the full command needed to compile the application is shown next and need to be executed from the directory where the java files are located in a Linux shell terminal:

javac semaphore_cfg.java semaphore_io.java semaphore_adc.java semaphore_agent.java

Once that the application gets compiled, a .class file will be generated for each one of the classes created in the complete application. These files will be created in the same directory. This process needs to be done in each one of the Galileo boards as each one of these will be a different traffic controller agent.

Launching Main Container

The main container could be any system or board in which **Jade** could be installed. This means that this could be any of the Galileo boards where the agent lives, another independent Galileo, Edison or other type of board or a computer or server connected to the same network. In this case, as is desired to monitor the full traffic controller system it was decided to use a PC. The reason for this it that in a PC it is possible to use the Jade graphical interface (GUI) which offers some helpful tools to help with the debug.

In order to launch the Jade GUI it is necessary to run the next command using the command prompt (either for a Windows or Lunix/Unix system) at the directory where Jade was installed:

RunJade.bat –gui

When this command is entered, the Jade GUI will be launched as shown in the next figure. Notice that the main container it's associated to the IP address of the PC where this lives. It's important to pay attention to this address, because each one of the agents requires to know this IP to get associated with this container. In the figure shown, the IP of the main container was **192.168.2.4**.

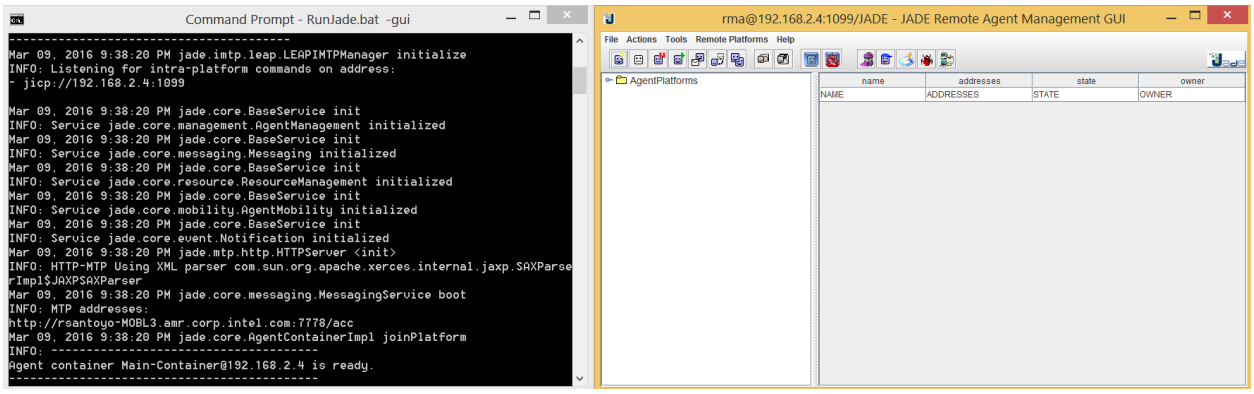


Figure 35. Jade GUI for the main container.

As mentioned above, the GUI interface provides some tools that may be useful for the analysis of the system. Some of the tools that were used to analyze the results captured in this project are described next:

- Allow to know that agents that are associated with this main container including the IP address of each one of them.
- Allow to monitor the messages shared among the agents using the Sniffer tool. This basically shows graphically the messages that go out from each one of the agents showing also the destination of this. Also it shows the performative of the message and the information that comes along with the message. In the case of this work, it's very useful because the agent that sends the messages with the INFORM performative indicates that this is the leader agent.
- Allow to send messages to any or all the agents in the system. This facilitates exercises any possible corner case that perhaps it's difficult to cover under normal conditions.

Launching Traffic Controller Agent

Once that the traffic controller agent is compiled and the main container has been already launched, each one of the agents must be started and added to the main container. For this, it is required to know this information:

- The IP of the main container
- The number of the agent in the traffic light controller system
- The number of the node assigned to the system,
- The operation mode of the agent, leadership selected based on the follow transition table (true) or based on the traffic density (false).

The template of the command use to launch the command is this one:

**java jade.Boot -host <main container IP> **
**-services "jade.core.messaging.TopicManagementService; jade.core.event.NotificationService; jade.core.mobility.AgentMobilityService" **
-container -agents 'AgentName:semaphore_agent(Agent_number,Node_number,OperationMode)'

In the case of the agent 0 in the node 1, this is the required command to operate in the mode of finding the leadership following the transition table:

**java jade.Boot -host 192.168.2.4 **
**-services "jade.core.messaging.TopicManagementService; jade.core.event.NotificationService; jade.core.mobility.AgentMobilityService" **
-container -agents 'Sem0:semaphore_agent(0,1,true)'

Once that all the agents gets started, these will appear in the main container GUI as shown in the next figure:

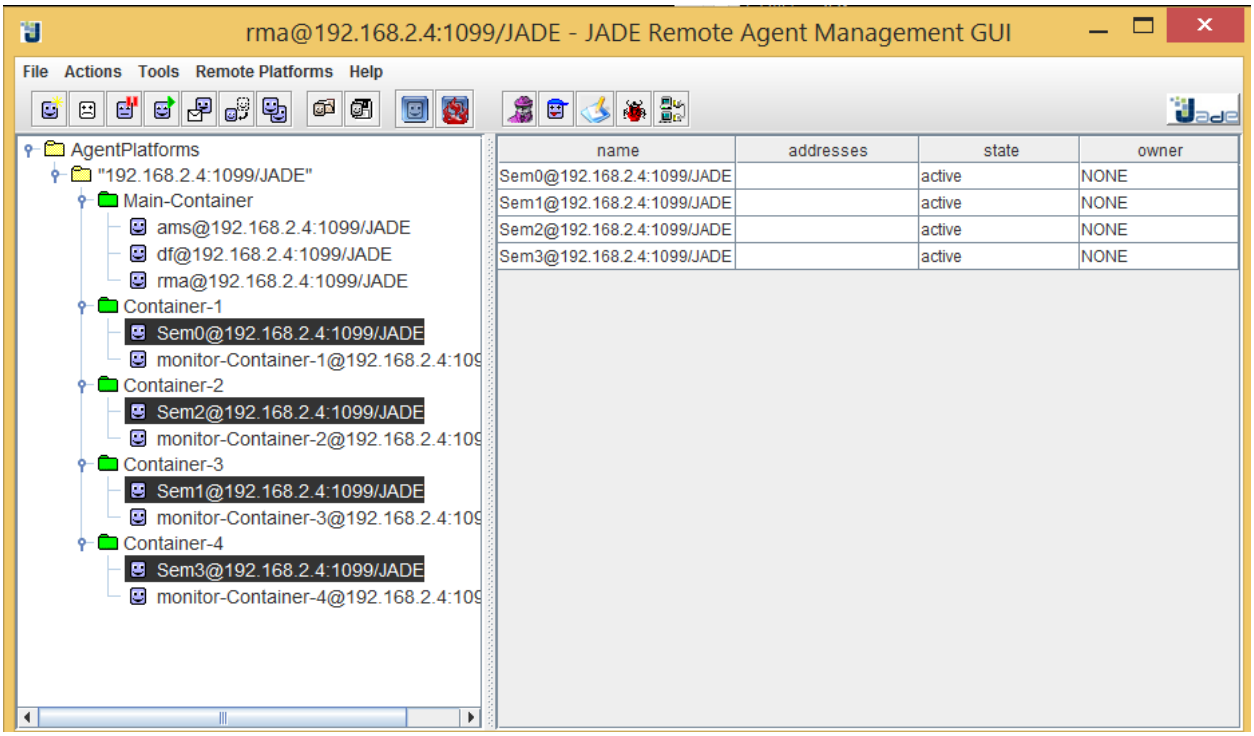


Figure 36. Jade GUI showing the agents in the Main Container.

Testing Scenarios

At this point it has been explained how to configure and start the full system (agents + main container). Based on this it is possible to start exercising the system to obtain some results under some specific conditions. So, in the next sections it will be presented the behavior observed in the agents under several scenarios playing with:

- Operation mode of the agents
- δ_T in the agents below threshold limit
- δ_T in the agents above threshold limit (traffic overload)
- Emergency control state

The behavior presented in the agents will include the leadership transition, the leadership times and the steps transition under each leadership period. The results will include a description and explanation of the behavior observed in each one of the scenarios to exercise.

Scenario 1.

Leadership Following Transition Table Mode, δ_T similar in agents under normal conditions

In this scenario the traffic light controller system is configured to operate finding the leadership using the transition table. Based on the transition table defined the expected agent leadership should be Agent0 -> Agent 1 -> Agent2 -> Agent 3 -> Agent0 and so on. Also, in this case the traffic density is set be approximately equal for each one of the agents. So, under this scenario, it's expected that the leadership time for all the agents be equal.

The experiment was done setting the $\delta_T \approx 2000$. Based on the traffic time equation defined in the previous chapter, the expected raw leadership time should be:

$$CycleTime = \frac{300*1000*\mathbf{2000}}{4096} = 146484 \text{ ms}$$

$$leadershipTime = \frac{2000 * 146484}{8000 * 1000} = 36.6\ sec$$

Following with the experiment, the next table shows the results obtained from this test. The table shows the order in which the agents were selected as leader.
Also can be seen the δ_T reported by each one of the agents by the time they requested the leadership. As it can be observed all of them are around 2000.
Also, it shows what is the raw leadership time calculated by each one of the agents, which matches with the expected 36 seconds, and it can be seen the final adjusted leadership time after considering any overlapped and repeated step in the cycle.

Leader Agent Ordering	Traffic Density Agents				Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3			Step	Time(s)
0	2004	2009	2007	2001	36	36	1	7
							2	3
							3	23
							4	3
1	2005	2009	2006	2003	36	10	5	7
							6	3
2	2004	2008	2007	2003	36	36	7	7
							8	3
							9	23
							10	3
3	2005	2009	2006	2002	36	10	11	7
							12	3

Related to this leadership time adjustment, if the original transition table is observed, it can be checked the steps in which each one of the agents is in green step and measure the time in which this lamp is on. The expectation from this is that all of the green times be equivalent as the traffic load is also equivalent. For example, check this points:

- For agent 0, the green lamp is on during S1, S2 and S3. Meaning that the lamp is ON during 33 seconds. Similarly, the left green lamp is ON during S1 and S2, meaning that this lamp is ON or Blinking during 10 seconds.
- For agent 1, the green lamp is on during S3, S4 and S5. Meaning that the lamp is ON during 33 seconds. Similarly, the left green lamp is ON during S5 and S6, meaning that this lamp is ON or Blinking during 10 seconds.
- For agent 2, the green lamp is on during S7, S8 and S9. Meaning that the lamp is ON during 33 seconds. Similarly, the left green lamp is ON during S7 and S8, meaning that this lamp is ON or Blinking during 10 seconds.
- Finally for agent 3, the green lamp is on during S9, S10 and S11. Meaning that the lamp is ON during 33 seconds. Similarly, the left green lamp is on during S1 and S2, meaning that this lamp is ON or Blinking during 10 seconds.

The results obtained from this experiment demonstrates that the behavior of the traffic light controller system works correct as the time measurements obtained from the experiment matches with the expected ones.

Finally, the next figure shows the message flow diagram delivered by the Jade Sniffer. The figure was arranged to show in each one of the columns the messages during each one of the leadership periods. It's difficult to appreciate all the messages that are being shown in the figure, but something that it's very clear is that the messages with **INFORM** performative are always generated by the leader agent. These **INFORM** messages are initially the **LEADERSHIP_NOTIFICATION**, then during all the leadership period several **STEP_NOTIFICATION** messages and finally the **LEADERSHIP_RESIGN** message. Also it can be seen several messages with **PROPOSAL** performative. These messages are basically the **LEADERSHIP_REQUEST** messages which may receive an **ACCEPT** or **REJECT** response. The agent which receives only **ACCEPT** responses is the one that will become the next leader.

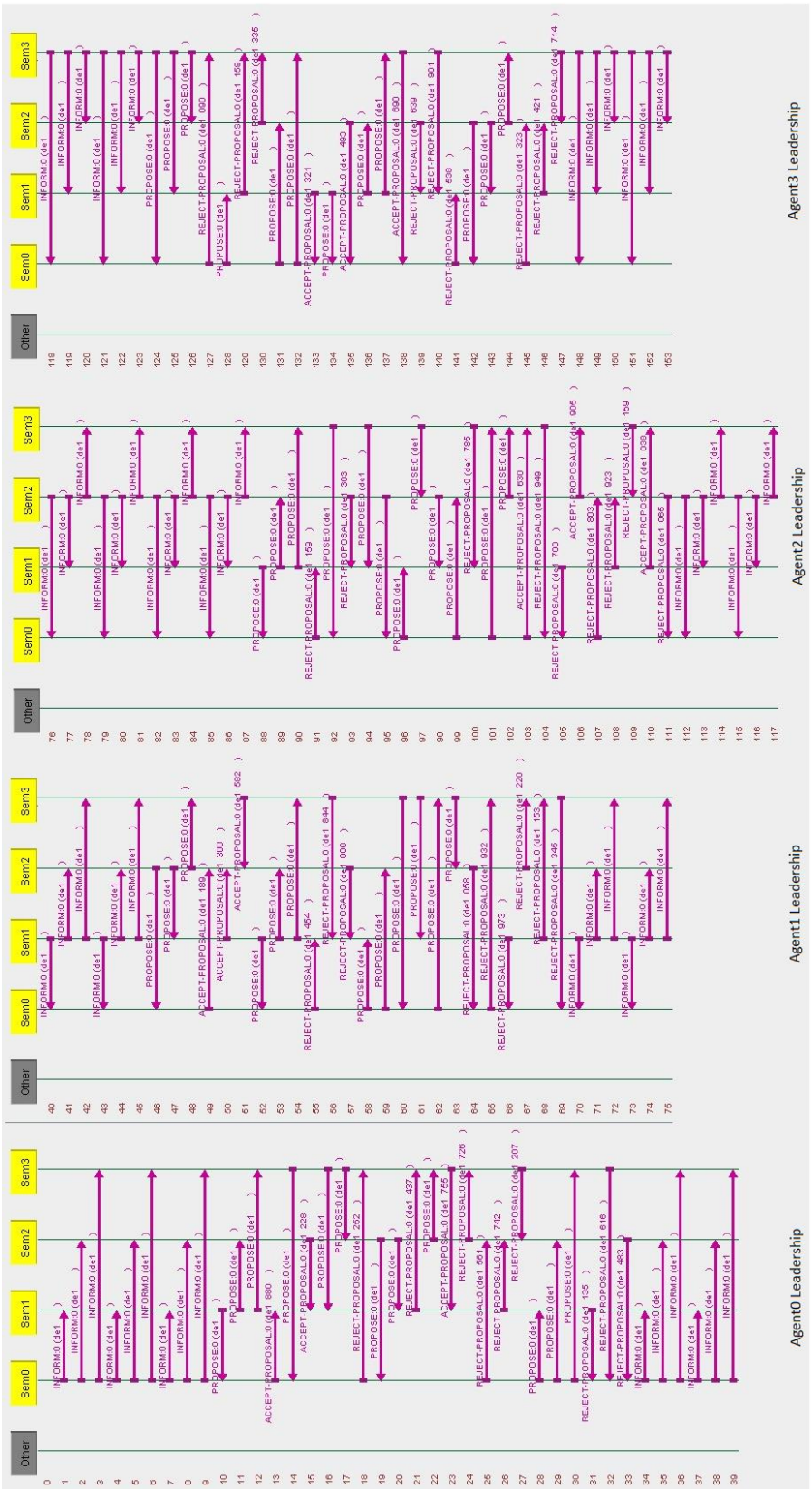


Figure 37. Message flow diagram for a full cycle when agents operates in Follow Transition Mode table.

Scenario 2.

Leadership Following Transition Table Mode, having different δ_T in agents under normal conditions

In this experiment the agents also are configured to find the next leader following the transition table, but this time they have different traffic density (δ_T) but all of them are still inside the valid range condition. The expected behavior is that the leader agent is selected following the scheme Agent0 -> Agent 1 -> Agent2 -> Agent 3 as defined in the transition table. In this case, the leadership time is expected to be different for each one of the leader agents because now the δ_T is different for all of them.

The results of this scenario can be seen in the next table. Here can be seen that the leader agent is being selected in the expected order. Also, notice that the δ_T for each one of the agents is different, but this doesn't affect the leadership selection. Although δ_T doesn't affect the leadership selection it actually affects the leadership time. For this example, the **CycleTime** is calculated to be around 259 seconds. Based on this, the expected raw leadership time (before any adjustment for step overlap or step repeated) can be calculated. This expected leadership time is shown in the table. The real raw leadership time calculated by the system is also shown. Notice that this raw leadership time is proportional to the δ_T observed by the respective agent. This raw leadership time is adjusted to discard the overlapped steps in each one of the leadership periods.
One interesting case is observed in the leadership period of the agent 3. As shown in the table, the raw leadership time for this period is 48 seconds, but the steps 9 and 10 are overlapped with the leadership period for agent 2. So because of this overlap, the leadership time for agent 3 must be adjusted. The interesting case here is that the overlap time for these 2 periods is 57 seconds (time for step 9 and 10), but this time is greater than the raw leadership time calculated for agent 3. So, in this case the leadership time for agent 3 is adjusted to the minimum leadership time for a 2 step leadership period, which is 6 seconds (3 per step).

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
0	485	3540	2521	1519	15.59	15	15	1	3
								2	3
								3	6
								4	3
1	486	3539	2522	1522	113.68	113	104	5	101
								6	3
2	485	3540	2523	1519	81.09	81	81	7	21
								8	3
								9	54
								10	3
3	485	3541	2523	1518	48.8	48	6	11	3
								12	3

The next figure shows the message flow for this experiment captured with the Jade Sniffer. In this case, the figure was arranged to show each leadership period in each column. In the figure can be seen that the agent that sends the INFORM performative message is the current leader.

Figure 38. Message diagram flow for a full cycle for the Scenario 2.

Scenario 3.

Leadership Following Transition Table Mode, having different δ_T in agents exercising traffic overload

In this experiment the agents in the traffic controller system are configured to operate in the mode of finding the next leader agent following the transition table, starting with traffic density inside of a normal range but at some point the traffic density is increased in one of the agent so this traffic overload is being detected in this agent. The expected behavior is that the agent with the traffic overload will get a higher priority and will gain the leadership until this condition is not observed anymore.

The next table captures the behavior observed in this experiment. In this case, during the 1st cycle the traffic density for all the agents is below the threshold specified in the **EmergencytrafficDensity** constant (value of 3900). In this first cycle the selection of the leader agent follows Agent0 -> Agent1 -> Agent2 -> Agent 3. After this cycle, the agent 1 reports a δ_T above the normal limit, so it's observed that it has precedence over the Agent0 which would be next leader agent if the transition table were used to select it. So in this case a new cycle starts with the agent 1. After this leadership period, this agent reports a δ_T which is still above the normal threshold, so this gains the leadership again. By the end of this leadership period the agent 1 now reports a δ_T below the threshold, so at this point, the next leader is selected using the transition table which is the agent 2. So in this case, the new cycle is like this: Agent1 -> Agent2 -> Agent3 -> Agent0. So the behavior observed and reported in the table matches with the expected result.

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
0	795	1366	1186	2311	23.78	23	23	1	3
								2	3
								3	14
								4	3
1	793	1367	1186	2312	40.9	40	22	5	19
								6	3
2	793	1366	1186	2311	35.49	35	35	7	7
								8	3
								9	22
								10	3
3	793	1367	1185	2311	69.15	69	43	11	40
								12	3
1	793	4057	1186	2313	144.11	144	144	3	98
								4	3
								5	40
								6	3
1	796	3902	1185	2317	135.99	135	135	3	92
								4	3
								5	37
								6	3
2	793	3538	1186	2311	39.26	39	39	7	8

								8	3
								9	25
								10	3
3	793	3539	1185	2311	76.52	76	47	11	44
								12	3
0	792	3540	1185	2311	26.23	26	13	1	4
								2	3
								3	3
								4	3
1	793	3543	1185	2311	117.38	117	117	5	114
								6	3

Also, in the next message flow diagram can be seen the leadership transition during the overload condition. Here the leader agent is the one that sends the message with the INFORM performative. So, in the figure it can be seen that after Agent3, the next leader agent is Agent1 during 2 periods and after this the Agent2 is the new leader.

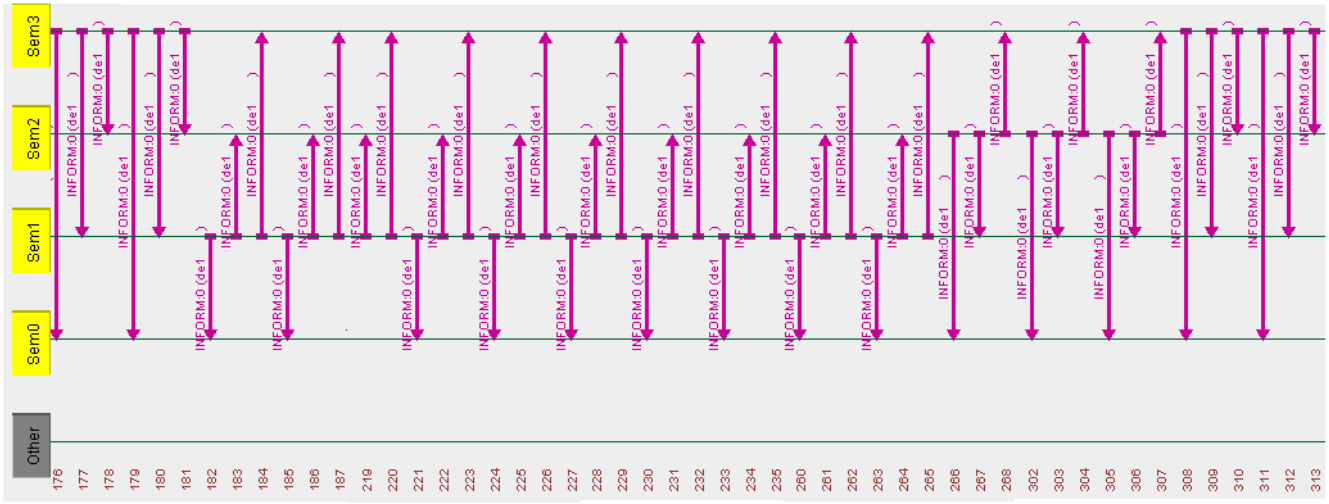


Figure 39. Message flow diagram for the Scenario3 (Overload condition).

Scenario 4.

Leadership Following Transition Table Mode exercising EMERGENCY_CONTROL messages.

In this scenario, the EMERGENCY_CONTROL message is sent to all the agents by the agent0 (in a real scenario this message is sent by an external agent such as ambulance or from a control central office). This message indicates that the agents must transit to Step 5 and the duration is for 50 seconds. The message is sent at 200 seconds since the beginning of the test, so in the log file is expected that around this time the agents receive this message and transit to the step 5. Also, around 250 seconds of execution time, the Emergency Control state should finish and in this time all the agents transit to IDLE state. So because the agents are configured to find the new leader following the transition table, after this point the new leader should be Agent0 as well.

This expected behavior was confirmed when testing this scenario, and this is shown in the next figure. Here it's shown that at 200 seconds the Agent0 sends the EMERGENCY_CONTROL message and this is received by all the agents inclusive this agent. By the time the message is received, the agents transits to the EMERGENCY_CONTROL_ST and sets the lamps according to the Step5 defined in the transition table.

Agent	A0				A1				A2				A3			
State	R	Y	G	LG	R	Y	G	LG	R	Y	G	LG	R	Y	G	LG
S5	1	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0

Also, it can be seen that after 50 seconds the agent transits to IDLE state (the figures only shows this for agent 0).

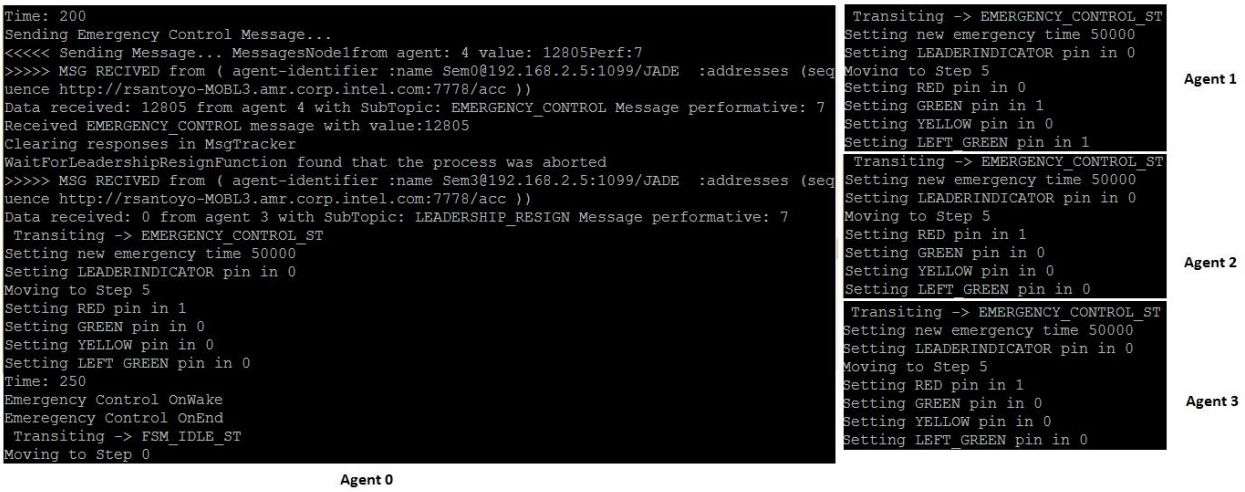


Figure 40. Emergency Control message received and processed by the agents. Step 5 for 50 seconds.

Once the system gets recovered from the EMERGENCY_CONTROL_ST, it will continue finding the leader using the transition table starting from Agent0. This is confirmed in the next capture taken in the test run.

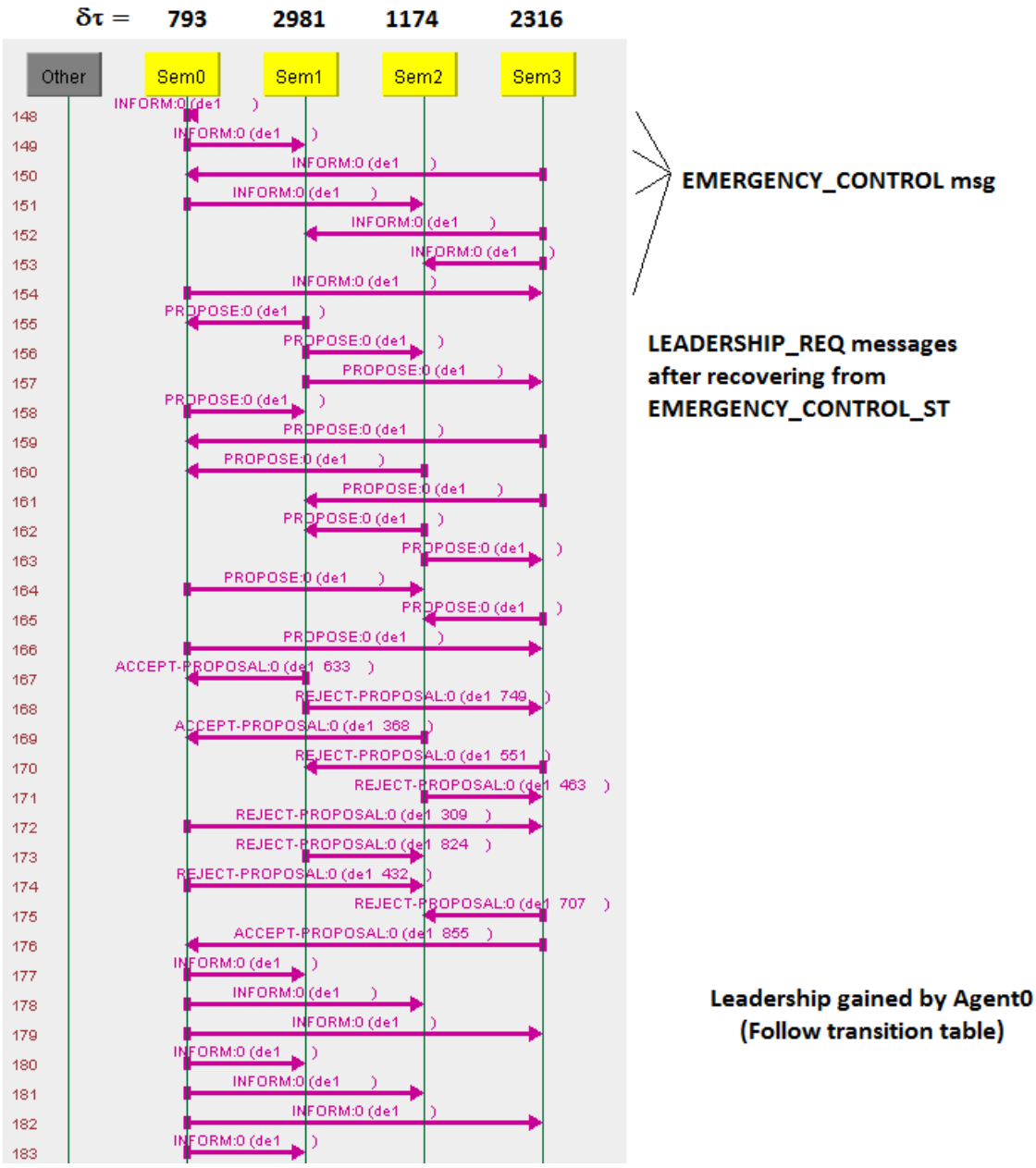


Figure 41. Message flow diagram after recovering from EMERGENCY_CONTROL_ST.

Scenario 5.

Leadership based on Traffic Density under normal traffic conditions (no traffic overload).

In this scenario the traffic controller system is configured to find the next leader based on the traffic density (δ_T), where this is inside of normal conditions for all the agents. The δ_T for each one of the agents is fixed having these values for agent 0 to agent 3: 325, 3626, 1388 and 2976. Under these conditions it's expected that the leadership selection is done in this way:

Agent1 -> Agent3 -> Agent2->Agent0

Also, the leadership time for each one of the agents should be proportional to the δ_T in the road the each agent controls.

Based on this, the result observed in the experiment is described in the next table:

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
1	326	3627	1388	2976	115.56	115	115	3	78
								4	3
								5	31
								6	3
3	326	3628	1388	2975	94.7	95	95	9	64
								10	3
								11	25
								12	3
2	326	3627	1388	2976	44.22	44	12	7	3
								8	3
								9	3
								10	3
0	326	3629	1387	2975	10.38	12	12	1	3
								2	3
								3	3
								4	3

In the table it can be seen that the leadership selection ordering matches with the expected one. Also it can be seen that the leadership time for all the agents matches with the expected values.
Notice that in the case of the Agent2 and Agent0 the leadership time is adjusted. In the case of Agent2, the time adjustment is done because of the step overlapping observed during the same cycle. In this case notice that step 9 and 10 are repeated, so it's necessary to discard the time that has been already covered in the previous leadership periods. In the case of the leadership time for Agent0, it's observed that the expected time is about 10 seconds. If this value is divided among the 4 steps, the time for each one of them would be less than the minimum step time. So in this case, the time adjustment is done based on the minimum step time which is defined as 3 seconds.

In the next figure it's also shown a section of the messages flow captured with the Sniffer. Here it can be verified that the agent leader is selected in the correct order (notice that the agent leader is the one who sends the messages with INFORM performative).

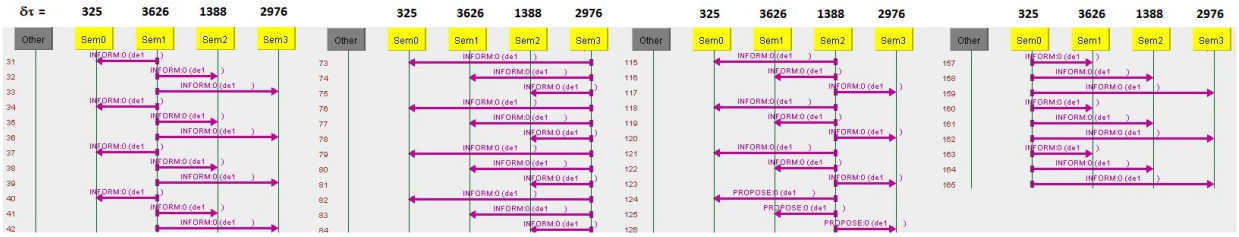


Figure 42. Message flow diagram for leadership based on traffic density.

Scenario 6.

Leadership based on Traffic Density under traffic overload condition for a single agent.

In this scenario the traffic controller system is configured to find the next leader based on the traffic density (δ_T). Initially all the agents starts with δ_T under normal conditions, and then one of the agents reports traffic overload during a couple of periods and then returns to normal conditions. The expected behavior is that the system must assign the leadership to the agent that reports traffic overload during all the periods in which this condition remains (regardless the same agent gains the leadership more than once in a cycle), and after the overload condition disappears, the system must continue finding the next leader in the same way using as decision element the δ_T of each agent.

The result observed with the experiment can be seen in the next table. Here it can be seen that initially, the δ_T for all the agents is inside of normal condition (< 3900) and remains in this way during one complete cycle. Here it's seen that the agent that initially gains the leadership is the Agent2, because it has the maximum δ_T (~3499). After this we can see that the next agents to be selected as leader are also based in the next maximum δ_T. Then, when the full cycle is completed the Agent2 is selected again as leader.

In the next leadership period it's observed that the δ_T for Agent0 is above the traffic overload threshold (in yellow). So, at this point the support for traffic overload is activated and the Agent0 is selected as the agent leader. In this case, notice that the leadership time is not adjusted, which is a requirement under this condition. In the next leadership period it's also observed that the δ_T for Agent0 still is above the threshold. So this agent is selected as leader again. When any of the agents is repeated during the same cycle, only the 1st one is inserted in the leadership track list. So, at the end of this period only Agent2 and Agent0 are in this list.

For the next leadership period it's seen that the Agent3 is selected as leader as it's seen that this has the next maximum δ_T. After, this, the Agent1 is the next leader agent based on the value of δ_T. With this agent, the current leadership cycle is closed. So, after this, the next agent to be selected as leader is Agent0 which is now the agent with maximum δ_T. Notice that because this is the 1st element in the table there is not any kind of leadership time adjustment.

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
2	1619	2571	3499	3194	82.39	82	82	7	21
								8	3
								9	55

								10	33
3	1614	2562	3497	3194	75.28	75	16	11	13
								12	3
1	1614	2561	3499	3193	60.39	60	60	3	39
								4	3
								5	15
								6	3
0	1615	2561	3497	3196	38.05	38	12	1	3
								2	3
								3	3
								4	3
2	1618	2561	3497	3194	82.39	82	82	7	21
								8	3
								9	55
								10	3
0	4082	2559	3498	3195	91.52	91	91	1	24
								2	3
								3	61
								4	3
0	3990	2561	3498	3195	88.04	88	88	1	23
								2	3
								3	59
								4	3
3	3656	2561	3500	3195	66.25	66	25	9	3
								10	3
								11	16
								12	3
1	3661	2561	3499	3194	53.17	53	21	3	3
								4	3
								5	12
								6	3
0	3652	2561	3498	3194	75.69	75	75	1	19
								2	3
								3	50
								4	3

In the next message flow diagram, it also can be seen the leadership ordering described in the table above. Notice that the agent that send the message with INFORM performative is the current leader.

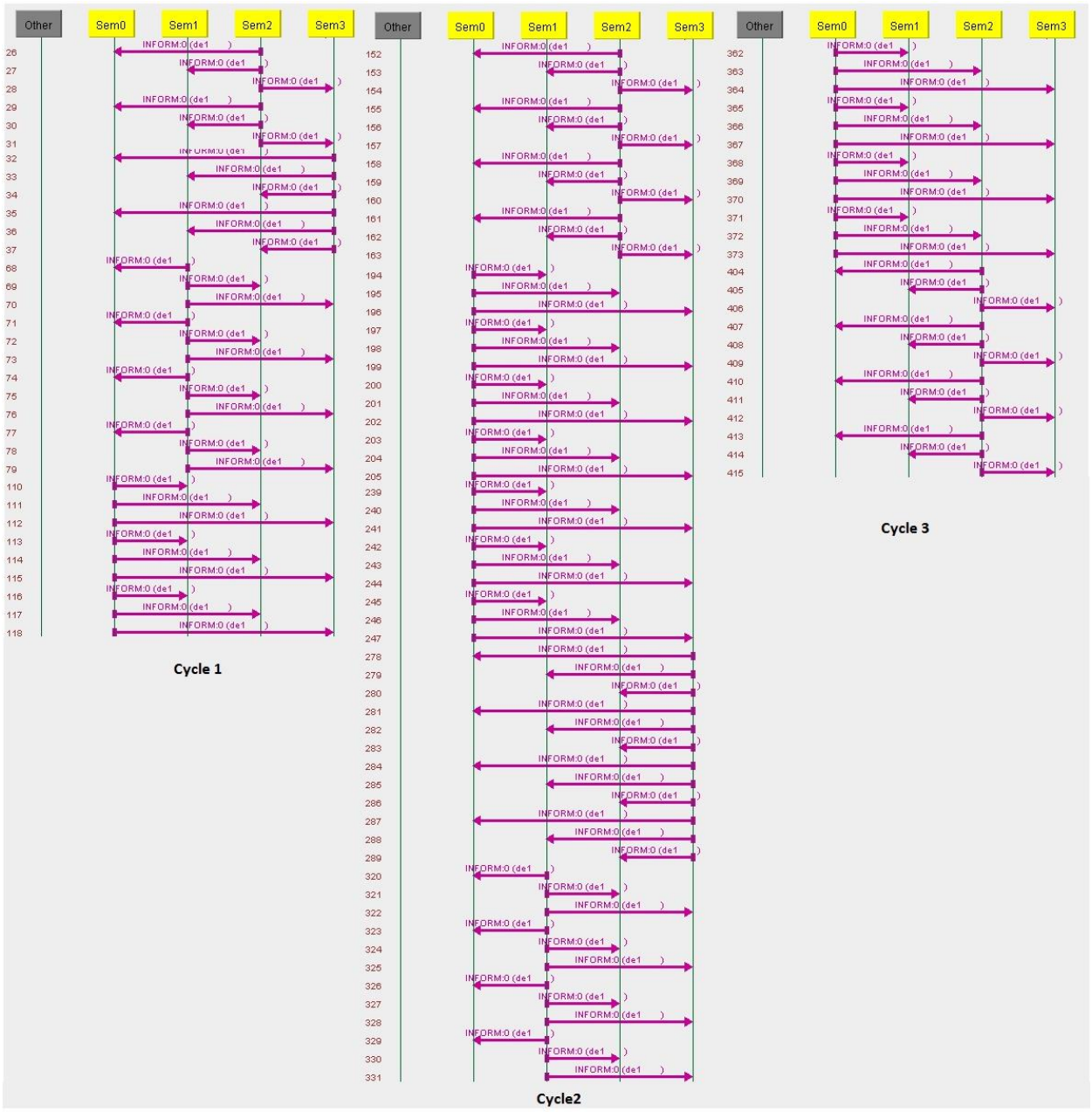


Figure 43. Message flow diagram for leadership based on traffic density with overload condition.

Scenario 7.

Leadership based on Traffic Density under traffic overload conditions for multiple agents.

In this scenario all the agents in the traffic controller will operate in the mode to find the next leadership agent based on the traffic density (δ_T) that each one reports. All the agents will start with a δ_T inside of the normal range, but at some time 2 of the agents reports a δ_T in the overload range. In this case, it is

expected that at this point the system assigns the leadership to the agent that reports the highest overload δ_T. The process should be repeated as long as there is any agent that still reports a traffic overload (regardless of the agent leader is repeated in the same cycle). Once that none agent reports overload, the system should continue working normally.

The next table describes the results obtained from the experiment implemented. During the 1st cycle, all the agents start with δ_T under normal condition range. In this cycle, the agent leader is selected based on the maximum δ_T. By the time the agent is selected as leader, this agent is inserted into a leader agent tracking list, so this agent is no longer selected again during this 1st cycle. In this cycle the leader agent selection ordering is like this:

Agent1 -> Agent0 -> Agent2 -> Agent3

At the beginning of the 2nd cycle, all the agents still reports normal δ_T levels. So the Agent1 is again selected as the 1st leader agent. During the 2nd leadership period of this cycle, the Agent1 and Agent3 reports a δ_T with an overload level. In this case the agent with highest δ_T is selected. So Agent3 is selected. In the next period Agent3 has also the highest δ_T value, so this is selected again regardless it has been already selected as leader in the current cycle. In the next period, Agent1 is the one that has the highest δ_T and this is still in the overload range, so this is selected as leader. In the next period Agent3 has the highest δ_T and this is the only agent that reports overload condition, so this is selected as leader again. In the next period none of the agents report overload condition, so the next agent with highest δ_T (discarding the agents already selected in the current cycle) is Agent0. Finally, the last agent to be selected as leader in the current cycle is Agent2. This completes the 2nd leadership cycle. Notice that in the case that overload condition is reported there is no leadership time adjustment. In the second cycle the leadership selection is like this:

Agent1 -> Agent3 -> Agent3 -> Agent1 -> Agent3 -> Agent0 -> Agent2

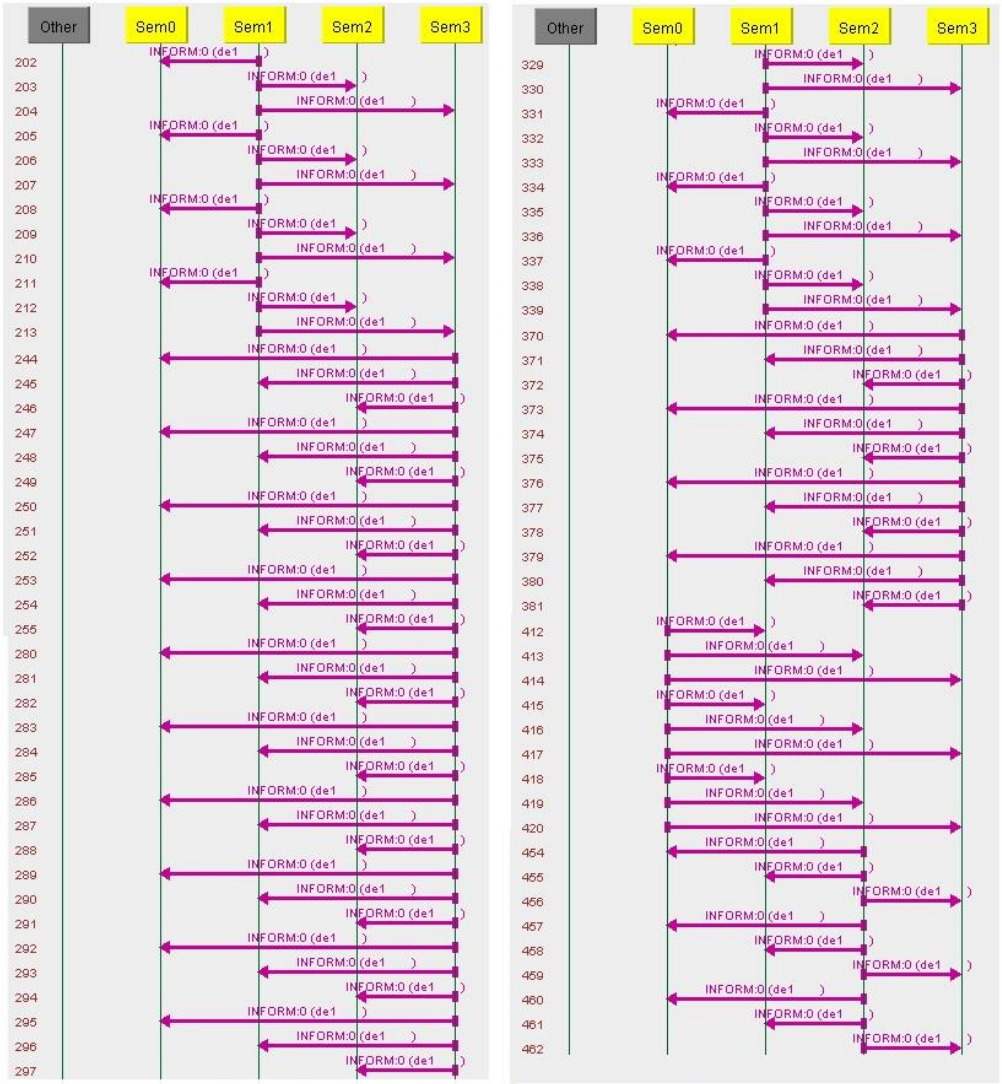
In the 3rd cycle, all the agents reports δ_T under normal range conditions. So, the leadership selection is done normally based on the δ_T reported by each one of the agents without repeating any of the agents already selected in this cycle. The leadership selection in this cycle is like this:

Agent1 -> Agent3 -> Agent0 -> Agent2

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
1	2898	3539	2180	1398	91.59	91	91	3	61
								4	3
								5	24
								6	3
0	2901	3530	2180	1398	74.93	74	12	1	3
								2	3
								3	3
								4	3
2	2895	3530	2183	1399	56.4	56	56	7	13
								8	3
								9	37
								10	3

3	2894	3530	2181	1403	36.24	36	6	11	3
								12	3
1	2898	3530	2181	1400	91.18	91	91	3	61
								4	3
								5	24
								6	3
3	2897	4053	2183	4065	91.7	91	91	9	61
								10	3
								11	24
								12	3
3	2896	4053	2179	4065	91.73	91	91	9	61
								10	3
								11	24
								12	3
1	2896	4052	2181	3965	91.83	91	91	3	61
								4	3
								5	24
								6	3
3	2895	3715	2181	3965	90.26	90	90	9	60
								10	3
								11	24
								12	3
0	2895	3714	2182	3547	63.82	63	24	1	15
								2	3
								3	42
								4	3
2	2904	3714	2183	3548	48.08	48	20	7	11
								8	3
								9	3
								10	3
1	2896	3712	2181	3548	81.80	81	81	3	54
								4	3
								5	21
								6	3
3	2906	3714	2181	3549	78.17	78	78	9	52
								10	3
								11	20
								12	3
0	2897	3715	2181	3549	63.86	63	24	1	15
								2	3
								3	3
								4	3
2	2897	3717	2181	3550	48.09	48	20	7	11
								8	3
								9	3
								10	3

The leadership selection for the 2nd cycle also can be seen in the next message flow diagram where only the messages with Inform performative are shown. This is helpful to identify the agent leader as the leader is the only agent that is able of send this kind of messages.



Sem1 -> Sem3 -> Sem3 -> Sem1 -> Sem3 -> Sem0 -> Sem2

Figure 44. Message flow diagram during the 2nd cycle where 2 agents reports traffic overload.

Scenario 8.

Leadership based on Traffic Density exercising EMERGENCY_CONTROL messages.

In this scenario the traffic controller system is configured to select the leader based on the traffic load (δ_T). The δ_T for each one of the agents is inside of normal condition range. At some point an **EMERGENCY_CONTROL** message is received by all the agents, make them to transit to any specific step

during some time. After this, the system must return to IDLE state and the system must operate normally again.

The next table shows the leadership selection ordering before and after the **EMERGENCY_CONTROL** message is being sent. In this case all the agent reports a δ_T inside the normal range. The **EMERGENCY_CONTROL** message is sent during the 3rd leadership period while Agent3 has the leadership. As can be seen the message is received while the step 11 in the transition table is being executed so step 12 is not executed as the message aborts the current leadership period. The **EMERGENCY_CONTROL** message indicates that all the agents needs to transit to step 3 during 100 seconds. In the test it was verified that the transition to step 3 is correct and the system remained in this step during the 100 expected seconds.
After this all the agents transit to IDLE state and the regular operation is restarted. This was confirmed as the next agent to gain the leadership was Agent1, which is the one that has the highest δ_T. After this, the leadership selection goes on with the next agent that has the next maximum δ_T which in this case was Agent0. After this Agent3 is selected as leader and finally Agent2 is selected completing this leadership cycle.

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
1	2896	3716	1682	1970	98.53	98	98	3	66
								4	3
								5	26
								6	3
0	2896	3717	1681	1968	76.82	76	12	1	3
								2	3
								3	3
								4	3
3	2905	3720	1681	1974	52.31	52	52	9	34
								10	3
								11	12
								12	3
NONE	EMERGENCY_CONTROL_ST				100	100	100	3	100
1	2894	3717	1681	1968	98.62	98	98	3	66
								4	3
								5	26
								6	3
0	2898	3718	1681	1971	76.85	76	12	1	3
								2	3
								3	3
								4	3
3	2903	3719	1682	1967	52.16	52	52	9	34
								10	3
								11	12
								12	3
2	2987	3717	1682	1969	44.22	44	12	7	3
								8	3

								9	3
								10	3

Scenario 9.

Leadership based on Traffic Density exercising EMERGENCY_CONTROL messages under overload conditions.

In this scenario, the traffic controller system works in the mode to find the leader agent based on the traffic load (δ_T) that each agent reports, but one of the agents reports traffic overload. When this agent has the leadership the **EMERGENCY_CONTROL** message is being sent asking to transit to some specific step. As the **EMERGENCY_CONTROL** message has the highest priority, the system must transit to the EMERGENCY step during the specified time. After processing the **EMERGENCY_CONTROL** step, the full system is restarted going to IDLE state and after this the leadership selection must be done normally based on the δ_T.

The next table shows the behavior of the leadership selection under this scenario for one single leadership cycle. Initially all the agents have a δ_T with value inside of the normal range. So the leader selected is Agent1 which is the one with highest δ_T. At this point, the Agent3 shows a δ_T in the traffic overload range, so this agent is selected as the new leader. During this leadership period, the **EMERGENCY_CONTROL** message is received asking to transit to step 3 during 100 seconds. So all the agents must transit to the **EMERGENCY_CONTROL_ST** and transit to step3 during 100 seconds. After that, the system is restarted and it's found that the Agent3 still reports traffic overload, so this agent is selected as leader again and the leadership time is not adjusted. After this, there is no longer traffic overload, so the next agent with the highest δ_T is Agent1, so this is selected as leader. After that, it is found that Agent3 reports again traffic overload, so this is selected again as leader, regardless it has been selected several times as leader in the same cycle. After that, none of the agents report traffic overload, so the system will choose as leader agent with highest δ_T that has not been added to the leader tracking list. In this case Agent0 is the next leader agent. Finally Agent2 is selected as leader which is the one that has the lowest δ_T.

The Leadership ordering basically can be summarize like this (in yellow traffic overload):

Agent1 -> Agent3 -> Emergency -> Agent3 -> Agent1 -> Agent3 -> Agent0 -> Agent2

Leader Agent Ordering	Traffic Density Agents				Expected Leadership time	Leadership Time	Leadership Adjusted Time	Step Transition	
	0	1	2	3				Step	Time(s)
1	2901	3713	1683	1933	98.70	98	98	3	66
								4	3
								5	26
								6	3
3	2904	3721	1680	4067	97.91	97	97	9	65
								10	3
								11	26

								12	3
None	Emergency Control				100	100	100	3	100
3	2905	3714	1681	4080	98.48	98	98	9	66
								10	3
								11	26
								12	3
1	2902	3713	1682	3529	85.38	85	85	3	57
								4	3
								5	22
								6	3
3	2903	3713	1681	3979	94.46	94	94	9	63
								10	3
								11	25
								12	3
0	2898	3713	1681	3122	69.04	69	26	1	17
								2	3
								3	3
								4	3
2	2901	3714	1681	3121	40.05	40	18	7	9
								8	3
								9	3
								10	3

Model of the proposed traffic light system topology

In order to demonstrate and verify the correct functionality of the system, it was created a scale model of the street intersection with the traffic light controller topology proposed. This model includes an intersection of 2 2-way streets. In this intersection there a 4 traffic lights controlling the traffic flow in each one of the directions. Each one of the traffic light controller includes 4 lamps (Green, Red, Yellow and GreenLeft) and a potentiometer to vary the traffic load emulated. Each one of these traffic lights is connected to a Galileo board. All the Galileo boards are connected to a router using the Ethernet interface and the PC where the main container runs is also connected to the same router using WiFi. So, this router allows to stablish the communication among the semaphores in the traffic light controller system.
This model was the prototype where all the scenarios proposed in this chapter were exercised and validated. A picture of the model created is shown in the next figure.

Figure 45. Model of the Traffic Light Controller System.

Chapter V. CONCLUSIONS AND FUTURE WORK

Conclusions

In this work it was presented the traffic congestion as one of the most important problems that are being faced in the big metropolis and the current governments are interested in find a solution to this problem. In order to do this, they are following different approaches that includes but are not limited to public transportation improvements, controlling the access to congested highways, car usage restrictions, carpooling and optimization of the traffic flow in main avenues using technology to synchronize traffic lights.

The work presented in this thesis provides a solution to mitigate the congestion problem based on the use of multi-agent technology applied to traffic light systems to make more efficient the control that these systems offer to the traffic in the street intersection where the system is located. It was mentioned that the efficiency improvement that the proposed system provides comes from the dynamic calculation of the time sequences in the traffic light controller system, providing certain level of intelligence to the system. It was indicated that calculation is done in real time and is adaptive based on the current traffic density (δ_T) in each one of the streets controlled by the system. This means that the time assigned to the green light in one of the streets is proportional to the δ_T measured in that street. The algorithm to calculate these times were presented in detail in the Project Specification chapter and it was shown that the main variables in the calculation were the δ_T of each one of the streets under the control of the system.

In the work presented it was described the main characteristics and properties of the proposed system:

- The system was implemented as an embedded system over the Intel Galileo board (one for each one of the traffic lights) in which it was mounted a software stack conformed by an Linux distribution (**Yocto**) as OS, a Java Virtual Machine, a middleware specific for multi-agents systems called **Jade** and on top of this, the application which is also made up of several software layers using Java as development language.
- It's a multi-agent system in which each one of the traffic lights is an agent.
- As a multi-agent system, each agent is communicated with each other sending and receiving messages to share information.
- Each agent becomes the agent leader for some period of time and this one controls the state of the lamps and their transition states for all the agents.
- The leader agent is selected in agreement with the rest of the agents based on the individual δ_T that is shared among them.
- The proposed system can be adapted easily to any topology whose lamp states is defined in a transition table. Also it can support different number of lamps in the agents. The system is also parametrized so it's possible to modify them these parameters to get adapted to the needs of each street intersection.
- The system can operate in 2 different modes: selecting the leader based on a predefined way in the transition table or select the leader based on the δ_T measured by each agent.
- The system is able to handle traffic overload so it can give additional priority to the street that reports it.
- The system is able of get recovered itself from any possible communication failure.

- The system supports attending emergency request from external agents (such as ambulance or main control office) which ask the complete system to transit to a specific state during certain time.

The great advantage of the system proposed over the traditional systems is basically the time optimization for the lamp state transitions which is adjusted in real time based on the δ_T, switching the priority of the traffic lights dynamically. Also, the handling of traffic overload and emergency conditions are attractive features for the system.

Finally it was presented some real scenarios in which the system was tested showing the transitions and lamps states for the topology selected and also some of diagrams that show the message flow that were exchange among the agents during the scenario exercised. These results demonstrated that the selection of the leader agent and the leadership and step time calculation are done correctly under many different conditions resulting as expected based on the system specification and algorithms defined.

Future Work

In this project, the only variable that is taking in to account to determine the green time assigned to each semaphore is the traffic density (δ_T). Modern traffic controller systems also include additional features that could be added to the current system. These features are for example:

- As this is project that fits in to the IoT world and the safety of people can be affected if the system is used is incorrectly, the system requires some kind of protection against vulnerabilities in the network in which the system operates. This means that the messages shared within the system require some level of security (like encryption) to avoid that invalid users can access the system and generate any kind of damage.
- Detection of humans or bicycles that pretend to cross the street.
- Audible signals for blind people who want to cross the street.
- In this project the traffic load is being emulated, but in order to determine this variable it is needed much more investigation to determine a feasible way to do it.
- Include circulation speed as an additional variable to determine the lamp state times.
- Synchronization with other traffic controller systems to have a better circulation in several streets and not only in individual street intersection.
- In terms of security, the inclusion of a “Panic button” and camera would be a feature with great value to prevent crime.

In order to solve the traffic congestion problem in cities there are other possible multi-agent solutions besides of the traffic light controller systems presented in this work. Some examples of these are mentioned next:

- Automatic speed controllers in roads based on the traffic load. If the speed limit is adjusted dynamically, the congestion could be reduced.
- Displays with real time traffic information that could notify drivers about any possible accident or closed lane ahead to start taking caution of this.
- Adaptive lane separators to increase the number of lanes in the direction with more traffic load and reduce the number of lanes in the other direction.

Although the above possible solutions could help out to reduce the traffic congestion in cities, the best investment that governments can do should be focus on public transportation, because this is how more people gets benefiting by this.
Related to this it's important to remember this:

"A developed country is not the one in which poor people can afford a car, instead it is the one in which rich people use the public transportation".

Bibliography

[1] **THE GEOGRAPHY OF TRANSPORT SYSTEMS**, 3rd Edition
Jean-Paul Rodrigue (2013)
https://people.hofstra.edu/geotrans/eng/ch6en/conc6en/ch6c4en.html

[2] **JADE**
http://jade.tilab.com/

[3] **La congestión del Transito Urbano, Causas y Consecuencias económicas y sociales**
Ian Thompson, Alberto Bull, Abril 2002, CEPAL
http://www.cepal.org/publicaciones/xml/6/19336/lcg2175e_bull.pdf

[4] **Intelligent Systems**
http://whatis.techtarget.com/definition/intelligent-system

[5] **Transport Problems in Cities**
K. Zabitsas, I.Kaparias, M.G.H. Bell. Imperial College London
7th Framework Programme, Theme 7 Transport
http://www.transport-research.info/sites/default/files/project/

[6] **Traffic Light Control and Coordination**
https://en.wikipedia.org/wiki/Traffic_light_control_and_coordination

[7] **Smart Traffic Light Systems**
https://en.wikipedia.org/wiki/Smart_traffic_light

[8] **MULTIAGENT SYSTEMS Algorithmic, Game-Theoretic, and Logical Foundations**
Yoav Shoham, Kevin Leyton-Brown
Cambridge, 2008

[9] https://www.cs.cmu.edu/~softagents/multi.html

[10] **Multiagent Systems, 1st Edition**
Gerhard Weiss
MIT Press
http://www.the-mas-book.info/index-first-edition.html

[11] **Mutiagent Barger Example**
http://www.planningaparttogether.nl/?page_id=150

[12] **Internet of the things Agenda**
http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[13] **IPv6**
http://searchenterprisewan.techtarget.com/definition/IPv6

[14] **What is the Internet of Things?**
http://internetofthingsagenda.techtarget.com/feature/Explained-What-is-the-Internet-of-Things

[15] **IoT examples**
http://systemdesign.altera.com/the-internet-of-things-can-drive-innovation-if-you-understand-sensors/

[16] **Galileo DataSheet**
https://www.arduino.cc/en/ArduinoCertified/IntelGalileo

[17] **Linux over Galileo**
http://blog.dimitridiakopoulos.com/2014/03/18/navigating-linux-on-intel-galileo/

[18] **JADE**
http://jade.tilab.com/

[19] **JADE TUTORIAL. JADE Programming For Beginners**.
Giovanni Caire, TILAB, 2009

[20] **FIPA**
http://www.fipa.org/

[21] **JADE Documentation**
http://www.iro.umontreal.ca/~vaucher/Agents/Jade/

[22] **Making a bootable micro SD Card with Linux**
https://software.intel.com/en-us/programming-blank-sd-card-with-yocto-linux-image-linux

[23] **MRAA**
http://iotdk.intel.com/docs/master/mraa/

[24] **Java Development Kit definition**
http://searchsoa.techtarget.com/definition/Java-Development-Kit

[25] **Java Development Kit vs Java Run Environment**
http://www.javabeat.net/what-is-the-difference-between-jrejvm-and-jdk/