A Cognitive Sensor Network based on Binary Spatter Codes

José Ángel Fernández Rodrigues
2013

Master of Science (120 credits)
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
A Cognitive Sensor Network based on Binary Spatter Codes

José Ángel Fernández Rodrigues

Luleå University of Technology
Dept. of Computer Science, Electrical and Space Engineering
Dependable Communication and Computing Group

5th September 2013
Traditionally, packets had been the minimum unity of information that had been used on digital communications to send and receive data between two entities. In them, information is stored using a local representation: a specific binary value placed inside a specific part of the packet. Some basic protocols like TCP/IP for network communications or 3G for radio mobile communications are packet based. However, in areas like Wireless Sensor Networks (WSNs) were complex environments with limited resources are very common, a packet network may not satisfy the needs the project in development requires. Extremely noise environments can produce a great number of packet losses making really difficult to establish a communication channel between nodes.

A new way to represent information was proposed by Pentti Kanerva where the information is encoded using distributed representations. Unlike local representations, information is not tightly defined by a specific value in a specific position if not the meaning is spread along a whole high dimension binary vector. The application of distributed representation inside WSNs for communication purposes started at the Dependable Communication and Computing Group inside the Department of Computer Science, Electrical and Space Engineering at Luleå University of Technology. The initial research developed a Medium Access Protocol (MAC) using distributed data representations as messages between nodes. The results obtained through simulations showed the viability of the protocol to be used on real networks inside noisy and harsh environments.

This thesis continue with the previous research done providing an implementation of an application inside WSN using distributed data representations and a learning algorithm to control the behaviour of the network. The long-term goal of this and future research is to build a zero configuration fault detection system based on distributed representations using binary spatter codes.

**Keywords:** Wireless Sensor Networks, Binary Spatter Codes, TinyOS, NS3, Symphony, HTML5, Websockets
This Master Thesis represents the end of an important stage on my life that begin on 2005 when I started my degree of Telecommunications Engineering at the Technical University of Madrid. They have been eight years with its good and bad moments but now, at the end, I’m able to see the light at the other side of the tunnel.

I would like to start thanking Evgeny Osipov and Laurynas Riliskis for allowing me to participate in their ongoing research projects at the Dependable Communication and Computing Group inside the Department of Computer Science, Electrical and Space Engineering at Luleå University of Technology. It has been a really great time working with them and without their support and comprehension, this project would not have been possible. I also want to thank to Denis Kleyko for his support and help through our Skype and e-mail debugging sessions.

I would like to continue thanking to people at the IEEE Student Branch at the Technical University of Madrid (a.k.a ”El Cubo”). Without them this degree would not have been the same, they made the worst moments better and the good ones much better. And thanks to Javi, for his nightly talks and his help whenever I needed.

I can’t forget also my family, they provided me the tools to arrive till this place. Now, it is time to start building my own ones.

Finally, I would like to thank the most important person in my life, You, for your support since the day we met more than 9 years ago. Without you anything would be the same because with you everything is better.

¡Gracias!

José Ángel
# CONTENTS

## Chapter 1 – Introduction

1. Communications and Architecture ........................................ 1
2. Applications ........................................................................ 3
3. Constraints .......................................................................... 3
4. Testing and Simulation .......................................................... 4
5. Thesis Purpose ..................................................................... 5
6. Delimitation .......................................................................... 6

## Chapter 2 – Theory

2.1 Cognitive Computing ............................................................. 7
   2.1.1 Distributed Representations ........................................... 8
   2.1.2 Encoding information using distributed representations .... 9
2.2 Wireless Sensor Networks .................................................... 13
   2.2.1 Node architecture .......................................................... 13
   2.2.2 Operating Systems .......................................................... 14
   2.2.3 Simulation tools .............................................................. 16

## Chapter 3 – Design Decisions

3.1 System architecture .............................................................. 21
3.2 System implementation ....................................................... 22
   3.2.1 I/O Client .................................................................... 22
   3.2.2 I/O Server .................................................................. 23
   3.2.3 Simulation Server ........................................................ 25
3.3 Binary Spatter Codes for data representation ......................... 27
   3.3.1 Sensor and actuator nodes ............................................. 27
   3.3.2 Base Station ................................................................. 27

## Chapter 4 – Prototype

4.1 Scenario Description ............................................................. 33
   4.1.1 I/O Client .................................................................... 33
   4.1.2 I/O Server .................................................................. 35
   4.1.3 Simulation server .......................................................... 38
   4.1.4 Message protocol .......................................................... 40
4.2 Testing .............................................................................. 40
4.2.1 Behaviour of the network ........................................... 40
4.2.2 Performance ......................................................... 45

CHAPTER 5 – DISCUSSION ............................................. 53
  5.1 Using distributed representations as a communication mechanism .... 53
  5.2 Future Work .......................................................... 55

APPENDIX A – IMPLEMENTATION CODE DETAILS OF THE I/O SERVER 57
  A.1 Endpoints configuration ............................................ 57
  A.2 Protocols implementation ......................................... 58
  A.3 Management of the events from I/O Client ......................... 60
  A.4 Management of the events from Simulation Server ............... 61

APPENDIX B – IMPLEMENTATION CODE DETAILS OF THE I/O PROXY 63
  B.1 Configuration of the I/O Proxy .................................... 63
  B.2 Receiving and parsing information from the I/O Server ........... 66
  B.3 Preparing and sending information to the I/O Server ............... 68

APPENDIX C – NS-3 SIMULATION SCRIPT ................................. 69
  C.1 Script ................................................................. 69

APPENDIX D – BASE STATION LEARNING ALGORITHM .................. 75
  D.1 Learning Algorithm ................................................. 75
The evolution in digital electronics, wireless communications and software tools has provided a great number of new low-cost, low-power and multi-use devices that can be used in a broad range of applications. Computing has evolved from personal computers and mainframes, to a new group of distributed applications powered by a lately generation of electronic devices (e.g. smart phones, all-in-one hardware boards like Raspberry Pi or Arduino, etc.)

A Wireless Sensor Network (WSN) can be defined as a collection of individual nodes that are able to interact with their environment by sensing or controlling physical parameters\[1\]. These nodes have to collaborate between them to fulfill their assigned tasks using wireless communications as collaboration channel. A node is defined as a low power device with limited computational resources. It interacts with the physical surrounding environment through sensors (e.g. temperature, wind speed, air flow, etc.) and actuators (e.g. switches, relays, motors, etc.). A sensor allows the node to get information from its surroundings while an actuator allows the node to modify the status of something nearby.

Thanks to these two basic elements, a WSN is able to monitor the environment, retrieve the data, process it and perform the correct actions based on the decision extracted from the received information.

1.1 Communications and Architecture

Communications in WSNs are based on radio interfaces that use electromagnetic waves to send the information between sender and receiver. As a difference off traditional wired networks, the propagation of the signal is made through the air, an unguided medium.
The main disadvantages of using the air as a transmission medium are related with the choice of the carrier frequency and the problem with interferences between different systems. The spectrum available is regulated by the International Telecommunication Union (ITU). The use of some frequency bands is restricted and special licenses are needed; however, there are some bands available without license used in Industrial, Scientific and Medical (IMS) applications. Common systems that use these bands are IEEE 802.11b, Bluetooth and the IEEE 802.15.4. These protocols work all of them in the 2.4 Ghz band, that implies that the coexistence of different networks in the same area running any of those protocols can produce several interferences between them.

The basic information unit in this kind of networks is the packet. Data inside the packets is stored using a local representation. It implies that the meaning of a bit is tied to its location [2], i.e. when a packet is created, the information is set in a position defined previously and its value is represented by a bit pattern, when the data is received in the other side, it is possible to read that location and extract the information. The problem of using local representations is that data transmissions are not reliable in wireless environments. Phenomena like diffraction, refraction, doppler fadding, variable attenuations, multipath trajectories, delays, etc. can modify the information sent and produce errors in the application.

Due to these problems, new techniques are being studied to create different ways to represent information. Pentti Kanerva proposes a new framework for computing, called Hyperdimensional Computing [3] based in the coding of the information through a Fully Distributed Representation [2]. It implies that the information now is not tightly encoded to specific bit positions if not the meaning is spread along the whole bit pattern.

While using distributed representations is something common in the cognitive and neural computing, its application inside WSNs as a communication framework it is not so common. Reference [32] introduces "to the best of our knowledge [...] the first attempt of using the framework for communication of loss and delay sensitive information."

Using a distributed data representation can be considered as an analogy of a traditional frame sent over a wireless connection; however, it introduces an improvement on the tolerance to losses making possible to recover the information only with the 53% of the original message. That is, it is possible to lose a 43% of the information due to interferences on the channel or noise and the information could be recovered. Moreover, it also adds new possibilities including "extracting new contextual information from parallel transmissions, i.e. collisions, cognitive reasoning and generalization of sensing phenomena".
1.2 Applications

The availability of different types of sensors and actuators has made possible the appearance of a broad number of applications for WSNs. As [4] establishes, it is possible to classify them in five main areas:

- **Military applications**: networks are used in activities related with the command and control, communications, surveillance o reconnaissance.

- **Environmental applications**: from measuring environmental parameters continuously like temperature, wind speed or humidity till complex activities like tracking the migratory movements of different animals.

- **Health applications**: the developments in smart sensor devices applicable to human beings expands the WSNs to this field. They are normally used for monitoring vital signs on patients and detect changes or faults.

- **Home applications**: as technology evolves new systems are integrated in nowadays homes. They allow the users to interact with their houses through new interfaces or from different places thanks to a connection over Internet.

- **Industrial applications**: this is perhaps the broadest area. It is possible to find sensor networks from the high-tech robot-based car factories to an interactive visit system inside a museum.

1.3 Constraints

Normally WSNs are designed to carry out a specific task. That makes one network quite different from the others. However, there are some physicals constraints that affect all wireless sensor networks and should be taken into account when designing these kind of systems.
• **Power Consumption**: there are two factors related with this point. The first one is storing and providing the energy needed by the node and the second, restoring it from a external power source [1]. Energy is usually stored in batteries. For extending the lifetime of the nodes it is possible to harvest energy from external power sources and use it to recharge the batteries: light, temperature changes, vibrations are some of the ones that can be used.

• **Hardware Constraints**: although computing devices are doubling its performance approximately every two years following closely the Moore’s Law[5], hardware used in WSNs doesn’t provided the similar performance than everyday computers due to the physical constraints related with size or power consumption among others. Since 90’s the hardware has evolved, however it still remains very important to optimize the software running over the nodes (e.g. routing algorithms, sensing procedures, transmissions processes) to expand the network lifetime, the maximum number of times a certain data collection function or task can be carried our without any node running out of energy[6].

### 1.4 Testing and Simulation

A WSN is usually composed of a great number of nodes that in some cases can add tens, hundreds or thousands[7] in biggest WSNs. Analytical modeling, simulation, emulation, testbed and real deployment are the most commonly used techniques [8] for testing purposes in WSNs design. However, running a real experiment on a real network is costly, difficulty and in some cases impossible. Simulators, emulators and testbeds provided a good set of tools to analyse the systems before being deployed.

**Simulators**

A simulation is the imitation of the operation of a real-world process or system over time [9]. Simulators strive to accurately model and predict the behaviour of real environment in different scenarios [8]. The model used inside the simulation is really important because simplified ones can produce inappropriate results and conclusions. A good simulator is characterized by its re-usability and availability, its performance and scalability, its support for rich-semantics scripting languages to define experiments and process results and for its graphical, debug and trace support [10].

**Emulators**

Emulation consists in reproducing the function or action of another system. The main difference with simulators is that the code inside the emulator can be run in real hardware also. In an emulator, some of the parts of the system can be implemented
(e.g. memory management, execution time, etc.) while some of them are simulated (e.g. communication links, traffic, etc.). Emulators are commonly based on hardware and software.

**Testbeds**

After using simulators and emulators to analyse and test the design, the last step before a real deployment is the use of physical testbeds. They provided the resources to recreate an environment close to the real one where the application can be deployed and executed.

All these tools have been run usually in on-premise computing resources available locally. However, with the raise of cloud computing services, a new approach has appeared for simulating and testing purposes. There is not a widely accepted definition about what cloud computing means but it can be defined as a set of network enabled services, providing scalable, Quality of Service (QoS) guaranteed, normally personalized, inexpensive computing infrastructure on demand, which could be accessed in a simple and pervasive way[11].

Cloud computing provides high performance computing resources at a reduced price. There are some projects starting to analyse the viability of using this new possibility for simulating and testing. For example, a scientific cloud computing platform for materials simulation[12] or an cloud environment for hosting electrocardiogram data analysis services[13].

### 1.5 Thesis Purpose

The purpose of this work is focused on the following two points:

1. Develop a cognitive application that implements distributed representations as a communication mechanism. The example application used is a small temperature control system with one or two temperature sensors and actuators.

2. Build a demonstrator to interact visually with the previous application from a web browser interface. Make it also extensible to replace the web interface for other different user interface.
1.6 Delimitation

Before continue with next chapter it is important to remark the boundaries set to the project:

- A pure communication based on distributed representation is not possible yet. There is not public available implementations using this kind of representation in the lower layers like the physical or link one. For sending information between nodes we still lie over traditional structures like frames or packets.

- Sensor networks contains usually tens or hundred of devices exchanging information. However, we will not enter into the behaviour of big-scale networks, our prototype will contain a reduced number (i.e 1 to 4 nodes) to show only that is viable using distributed representations on WSNs.
This chapter includes a reference of the theory related with this project. In the first part, section 2.1 introduces the principles of distributed representation and hyperdimensional computing used to build cognitive systems. In the second one, section 2.2 introduce the concepts related with WSNs. It describes the architecture of a node, the use of WSN specific operating systems, and the tools available for simulating applications before its deployment. Subsection 2.2.2 describes TinyOS operating system and subsection 2.2.3 the NS-3 simulator, the specific software used in this project.

2.1 Cognitive Computing

Since the 70’s, with the introduction of the first commercially available microprocessor, conventional computing has evolved based on the Central Processing Unit (CPU) or processor as a core element inside the computers. The hardware architecture was based on von Neumann principles. It consists on the processor plus a memory and the channels for input and output data. All the components are build using transistors as a base element.

However, this technological approach has its limitations on building computers to have a kind of intelligence available in some animals and human beings. The main two limitations are:

- **The size of the components used inside a processor.** As the size is reduced to the scale of nanometres to increase the speed, the physical limitations of the hardware appear limiting the number of operations that can be obtained.

- **Limitations of the architecture.** The operations and code of a system are predefined by programmers and computer engineers. Computers are silly devices that execute the orders inside its memory to perform a task and obtain a result.
However, some problems that need a more intelligent behaviour don’t fit in the conventional computing approach.

In the first case, a possible alternative solution is Quantum Computing, based on qubits instead of bits and ruled by the basics of quantum mechanics. The analysis of this kind of computing is outside the scope of this work. In the second case, engineers have looked for new approaches; for example, creating computing models used to develop computing programs and devices based on how the brain works.

This new type of computing was called Cognitive computing; it integrates biology and technology in order to recreate the insides of the human brain, considered nowadays as the most efficient computer. Its origins are based on artificial intelligence; however, cognitive computing try to teach computers to think like the human mind does, rather than create an artificial system to perform a predefined task. The main problem with artificial intelligence is that most of these intelligent systems cannot learn from their our experiences. That’s way, this knowledge cannot be used in the future to make a decision as a response of the actual situation.

Improvements in the understanding of how the brain works has allowed engineers to create systems based on the working principles of the mind. This systems are able to integrate past experiences and use them to get a result. The next two sections go into details about how the data is represented 2.1.1 and how can be manipulated in cognitive computing to build cognitive systems 2.1.2.

2.1.1 Distributed Representations

Inside a conventional computer, data is represented using the bit as a minimal information unit. That data is stored inside the memory in a specific region called a memory address that has a certain amount of bits (i.e. 32 or 64 bits in modern architectures). The association of a memory address with the chunk of information it contains can be defined as record-field association or local representation.

In local representations, the meaning of a bit is determined by its location. For example, if we want to represent the letter ”A” in ASCII Encoding, the binary value will be: 01000001. Each bit with its value in that specific position represents the desired letter. However, any change in one or more bits, the content is changed and the original information is lost. For example, 01000010 is the letter ”B” and 00110001 is the representation of the number 1. This kind of representation is really weak to errors and ”may actually hinder the development of the kind of computing that makes brain intelligent” as Kanerva says [2]
Brains doesn’t use local representations as the way to store the information. Neural representation can be characterized by the following properties [3]:

**Hyperdimensionality**: the working of the brain is based on the union of really big numbers of neurons that continuously interact between them through synapses. Instead of using words of 32 or 64 bits, Kanerva proposes word of bigger sizes like 10.000 bits. Hyperdimensional vectors of this size have special properties that can leverage a new way of computing based on the same principles of the brain.

**Robustness**: local representations are weak to errors, something completely different to neural architecture. Robustness is obtained using redundant representation of the same concept were different patterns encode the same concept. If any part is modified, the original information can be retrieved. The tolerance to errors increases as the dimensionality does.

**Independence of position**: If the information is really tied to one position, any failures can lead to a loss of information. Encoding the information along all the components increases the robustness in the presence of errors. This way to represent the data can be defined as a distributed representation.

**Randomness**: finally, the connection between the neurons is not predefined. Brain’s plasticity allows the system to generate the connections randomly. However, despite that every brain is completely different, if we look at the connections inside, most of them are able to develop the same tasks: sight, smell, hearing, etc.

### 2.1.2 Encoding information using distributed representations

Before continuing is important to define the following concepts to fully understand how distributed representations are built to create a binary spatter code for coding information:

**Field**: a smaller piece of data from a larger collection. In distributed representations, a variable-value relationship also called role-filler relationship. For example: NodeId = 1.

**Data Record**: a collection of fields combined together to create a single entity. For example: NodeDescription = [NodeId = 1, NodeType = Sensor, NodeState = On].

**Codeword / Hypervector**: N-dimensional binary hypervector with independent and identical distributed components (e.g each bit is generated randomly with a probability of 0.5 for both 1 and 0). A codeword is used to represent any of the previous mentioned elements in a distributed representation. The process is explained below.
The value of \( N \), the dimension of the space, must be big enough in order to clearly see the mathematical properties associated to hyperdimensional representations. Kanerva defines the value of \( N \) equal to 10,000 bits. This size bit vector allows encoding \( 2 \times 10^{3010} \) different entities. Each entity will differ in a number of \( n \) bits from another available inside the hyperspace; this number is called the Hamming distance. Using a normalized version of the Hamming distance, the distances between codewords are highly concentrated around 0.5. However, the most important fact in Kanerva’s analysis, is that almost all the hyper space is contained in a 600-bit wide space around the 0.5 point.

The conclusion extracted is that we can randomly choose two vectors from the hyper space and they will differ around 5,000 bits one from the other. One of the consequences is that we can encode information using a hyperdimensional representation and recover a 47% of the original information after being distorted up by a noisy channel. Thinking in the opposite way is possible to define the concept of similarity between hypervectors: two vectors are similar if the normalized distance between them is smaller than 0.5.

The next step is knowing how to represent a data record using distributed representations instead of the traditional local representation. As an example, if the following data record wants to be encoded:

- **BaseStation**
  - NodeId = 1
  - NodeType = Sensor
  - NodeStatus = ON

First, we encode any of the fields inside the data record. The operation used is called **binding** and is based on the XOR (\( \otimes \)) function between codewords. We should choose randomly an hypervector from the hyperspace and associated it with the role; for example, \( \text{NodeId} \) will be the hypervector associated to the NodeId field. In the same way, we choose another one for the filler, \( 1 \). The field will be defined as \( \text{NodeId} \otimes 1 \).

Next, we encode all the fields inside the data record to build a single entity. The operation used is called **chunking**. Chunking a collection of hypervectors (i.e. fields) is done by adding component wise all of them and applying the majority rule. If the number of hypervectors is even any tie is resolved picking one value with probability 0.5 each. Chunking provides a new vector with the same dimension and with each component being equally probable; that implies that the generated codeword is also in the original hyperspace. The used code is recursive. As a example, the BaseStation record will be encoded as:
BaseStation = (NodeId ⊗ 1 + NodeType ⊗ Sensor + NodeStatus ⊗ ON)

The encoding process is detailed graphically on Fig 2.1.

To extract a filler from a field the XOR function is applied again because it is its own
inverse. The result of the function will be:

NodeId ⊗ (NodeId ⊗ 1) = (NodeId ⊗ NodeId) ⊗ 1 = 0 ⊗ 1 = 1

In the case of a data record, is possible to extract the filler applying the same mecha-
nisms

(NodeId⊗BaseStation) = (NodeId)⊗[NodeId⊗1+NodeType⊗Sensor+NodeStatus⊗
ON]

Applying the distributive property of the XOR function

NodeId⊗NodeId⊗1+NodeId⊗(NodeType⊗Sensor)+NodeId⊗(NodeStatus⊗ON)

And finally

1 + NodeId ⊗ (NodeType ⊗ Sensor) + NodeId ⊗ (NodeStatus ⊗ ON) = 1 + X + Y
A difference of the first case where the filler was extracted from the field, when the filler is extracted from a data record extra hypervectors are added to the expected result. X and Y can be considered random noise added to the original hypervector. To remove the noise it is possible to use a clean-up memory. A clean-up memory stores a copy of all the codewords that are associated with a concept (i.e. NodeId, NodeType, NodeStatus, 1, Sensor, ON) and is able to provide a noise-free version of an hypervector when it is probed with its noisy version. Autoassociative memories are frequently used as clean-up memories, they store each hypervector using the hypervector itself as the address. If an autoassociative memory is probed with a noisy hypervector, it compares the input with all the codewords stored applying a closest-match algorithm and retrieves the result as the noisy-free version of the hypervector.

The decoding process is detailed graphically on Fig 2.2.

![Figure 2.2: Decoding a data field inside a data record using a clean-up memory](image)

Based on [14], employing distributed representations is interesting because they provide an efficient use of representational resources and nuances of meaning can be expressed by small changes in patterns. Moreover, applications using distributed representations can be noise tolerant and the patterns can be superimposed but still individually recognizable.

However, they have two main limitations:

- Only a few components can be included in each chunk at the same time.
- As the number of components present at the same time increases, the probability of interference between chunk increases as well.

These two limitations can be related to the discoveries of the cognitive psychologist George A. Miller related with the working memory in 1956 [15]. Miller proposed that the limit associated with the short-term memory or working memory was the close to
2.2 Wireless Sensor Networks

On Chapter a general view of WSNs and its components was introduced. In this section, a more detailed explanation about the node architecture, the operating systems they run and the simulators available is presented.

2.2.1 Node architecture

Due to the difference between requirements from one application to another, it is not possible to find a standard node architecture used on all WSNs. However, there are five elemental components that can be found in almost all nodes:

- **Microcontroller** or **Microprocessor**: the microcontroller is the heart of the node. It executes the application code, manages the hardware resources available and process the data.

- **Memory**: the memory is used to store temporal or long-term data. There are three types of memory that can be used inside a node:
  - *Read Only Memory (ROM)* used normally for application code storage. When the system starts, the code is read and the execution of the application begins.
  - *Random Access Memory (RAM)* used to store temporal data while the application is executing. When the system is powered off, all the information is lost.
  - *Electrically Erasable Programmable Read-Only Memory (EEPROM)* used to store information, like configuration settings, when the system is powered off.

- **Sensors and Actuators**: the sensor and actuators are the elements that allow the node to interact with its surrounding environment.

- **Communication**: a communication device provides access to a wireless channel allowing WSN nodes to send and receive information from other nodes inside the network.

- **Power**: the supplier of the energy to all node components. Without a power supply the node can’t accomplish its tasks.
There are several commercial hardware platforms (motes) available on the market and choosing a specific one depends on the exact characteristics required by the application. In [16] there is an analysis of 43 different motes. The most popular ones in research are Mica2, MicaZ and Telosb. A summary of its characteristics is include in Table 2.1.

<table>
<thead>
<tr>
<th>Platform</th>
<th>CPU</th>
<th>Clock (Mhz)</th>
<th>RAM/Flash/EEPROM</th>
<th>BW (kbps)</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mica2</td>
<td>Atmel Atmega 128L</td>
<td>8</td>
<td>4K/128K/512K</td>
<td>38.4</td>
<td>TinyOS</td>
</tr>
<tr>
<td>MicaZ</td>
<td>Atmel Atmega 128L</td>
<td>8</td>
<td>4K/128K</td>
<td>250</td>
<td>TinyOS</td>
</tr>
<tr>
<td>Telosb</td>
<td>TI MSP430F1611</td>
<td>8</td>
<td>10K/48K/1M</td>
<td>250</td>
<td>TinyOS</td>
</tr>
</tbody>
</table>

### 2.2.2 Operating Systems

It is possible to use home-made firmware in simple applications. However, as the complexity increases, an Operating System (OS) is needed. An OS can be seen as a manager that controls the hardware resources like memory, radio device, etc. and makes them available to the users in an orderly and controlled manner[17]. The OS role is really important; its performance in the management of the radio interface and its optimization to reduce power consumption are vital for building long-term WSNs. Moreover, using an existing OS reduces the development time of the applications avoiding start writing from scratch each time that a new application is needed.

OSs used in WSNs can be classified in two groups based on its programming model:

- **Multithread Programming**: tasks are organised as threads and executed inside the processor. If there is more than one task, the OS provides a time slot for each task and executes them sequentially. When one time slot finish, the state of the execution is saved and the resources are assigned to a new task. Its main advantage is that avoids OS blocking in the execution of long time-consuming tasks. Its main disadvantages is the extra resource consumption to store the information between task changes.

- **Event-Driven Programming**: task execution is determined by events like interruptions. When a new event is raised, the processor executes the associated task until it finishes as a difference of the multithread model. Its main advantage is the good use of resources in motes with scarce hardware. Its main disadvantage is that the OS stays blocked until time-consuming task executions are finished.

There are several operating systems available: Contiki[18], MANTIS[19], LiteOS[20],etc. However, TinyOS has became de facto standard operating system in WSNs. Most popular platforms support TinyOS as operating system.
TinyOS

TinyOS is an operating system designed for low-power wireless devices used in sensor networks, ubiquitous computing, personal area networks, smart buildings and smart meters[21]. It is based on an event-driven programming model with a reduced memory footprint and characterized by the following three properties:

- **Component-Based Architecture**: TinyOS is developed using component-based development. It provides a collection of system components to build the applications. A component is a software module that encapsulates a set of functions related to each other and each component is loosely coupled with others so it can be reused several times. Communication between components is done through interfaces. Each component exposes the functionality it offers through its public interface. If any other component wants to use that functionality, it only needs to call the methods found in the interface. Using interfaces allows the replacement of one component by an updated version or an alternative one that exposes the same interface. For example, the functionality of a radio hardware device can be exposed as a component inside TinyOS. If a better hardware device appears in the market, it is possible to replace the implementation of the component to use the new one while keeping the same interface than before. Any component using that interface will continue working without any modification. Another advantage of Component-Based Architecture is that unused components don’t need to be included in the application so, it is possible to reduce its memory footprint.

- **Split-Phase Operations**: Operations that consume a long execution time are implemented as a split-phase functions: operation request and completion are separated. When a call is made to a split-phase function, it returns instantly and an event is raised when the operation has finished. The caller should subscribe itself to the event to know if the operation has been successful or not. For example, if the application wants to send a packet over the network, it should call the send command and subscribe to the sendDone event. Commands and events are explained in the next section.

- **Event-Based concurrency model**: TinyOS has a concurrency model that allows the execution of the code in two different modes: asynchronous and synchronous. When the code is running in synchronous mode it is not interrupted until it finishes; however, using the asynchronous mode, the code can be interrupted by external hardware interruptions. This concurrency model has low overhead, in contrast to a concurrency model based on threads, and allows a high concurrency schema.

TinyOS provides several component to be used directly inside the applications. It is possible to find components for network protocols, sensor drivers, etc. As additional
features, it also includes support for a single file system, small databases with TinyDB, security mechanisms with TinySec and a simple simulation tool called TOSSIM.

Applications in TinyOS are written in NesC[22], a programming language designed for TinyOS as an extension of C. NesC code is translated to a ANSI C in the compiling process and after that, the result is compiled with a platform specific compiler to obtain the final binary image. The image is deployed after that inside the mote and the process has finished.

**Commands, Events and Tasks** Components inside TinyOS have three computational abstractions: *commands*, *events*, and *tasks*. Commands and events are mechanisms for inter-component communication, while tasks are used to express intra-component concurrency[23].

Commands are exposed by the components through its interface. They allow other components to make a request or perform an action. Events are the way to signal that the request has been performed. Both commands and events should not block the execution; if they need to perform any time-consuming operation they should create a task. Tasks are functions scheduled by TinyOS to be executed later depending on the availability of the computational resources. Tasks must be short, very long operations must be divided in smaller tasks, and should be used when time requirements are not very restrictive.

Events can also represent a message from the environment, not only completion notification from a split-phase operation. The execution of TinyOS is ultimately driven by events representing hardware interrupts[22].

### 2.2.3 Simulation tools

It is possible to find two kind of simulators to be used in WSNs: general purpose and WSNs specific ones[8]. General purpose simulators can simulate other kinds of networks but they offer plug-ins or extensions that allow them to be used in the testing of WSNs as NS-3[24] or OMNeT++[25] among others. The specific simulators are normally based on a general purpose one modified and adapted to be used primarily only in WSNs like Castalia[26] built upon OMNeT++ or Prowler[27] upon MATLAB.

For TinyOS, it is possible to use TOSSIM[28]. It has been designed for emulating TinyOS[23] applications running inside a hardware platform called MICAz[29]. TOSSIM emulates the behaviour of raw hardware like the Analog-to-Digital Converter (ADC), the Clock, the transmit strength variable potentiometer, the EEPROM, the boot sequence component, and several of the components in the radio stack[28].
If the simulation is not enough, it is possible to use physical testbeds with remote access like ORBITLAB[30]. It has 400 802.11 and 3G radio nodes in a 20-by-20 grid architecture that allow the experimenter to choose the architecture of the networks that suits his experiment.

**NS-3 and Symphony**

NS-3 is an open source discrete-event network simulator [24]. It can be used to simulate the behaviour of a collection of nodes inside a virtual environment before its deploying to the final place. The level of detail used in the implementation of the simulation models, allows also the interaction between simulated nodes running inside NS-3 and real nodes running in a real environment. A message sent from a simulated node can be routed to a real one and the reply can be received back in the node without any difference with a real node-to-node communication.

The simulator is implemented as a C++ library and includes a collection of simulation model for IP and non-IP networks. Most commonly used are WiFi, WiMax, LTE or CSMA (e.g. Ethernet). A simulation is based on a C++ application that creates instances of different simulation models, configures them and sets up the simulation environment. After that, the simulation loop is started and the application finish when the simulation time finish or there are not more simulation events to execute. A simulation event is a function call scheduled to be executed at a specific time inside the simulation.

NS-3 uses the following abstractions to build the simulations:

**Node:** A node is an abstraction that represents an simulation element inside a network. Doing an analogy with the traditional computing, it is the chase of a computer where different elements can be installed. This abstraction is represented in NS-3 by the class *Node* that provides the base methods to control its representation in simulations.

**Application:** An application is an abstraction of the functionality that our node should execute. It is represented by the class *Application*. Continuing with the analogy, just computers executes applications to perform a task, *Nodes execute Applications*. Extending the *Application* base class it is possible to develop new applications to execute inside a node.

**Channel:** A channel is an abstraction that represents the physical medium where the information is sent. In the NS-3, a Node should be connected to a communication channel to interact with other nodes. Channels are represented by the class *Channel* and it provides methods for managing communication subnetwork objects and connecting nodes to them.
NetDevice: A net device is a component introduced between a Node and a Channel. It adapts the information sent by the node to the physical properties of the channel. In NS-3, this abstraction is represented by the class NetDevice; it provides the methods to manage the connection between a Channel and a Node object. To use a Channel object inside our Node, a NetDevice element should be installed. One node can be connected to several different channels using the associated NetDevice for each Channel.

Figure 2.3 shows how the previous four abstractions are related inside a simple scenario where two nodes want to exchange information.

One of the aims of this work is to execute TinyOS code inside simulations. NS-3, however, doesn’t provide native support to this option. Nevertheless, it is possible to include it thanks to the modular design and the facilities for extension that NS-3 provides.

Symphony is a module extension for NS-3 developed at Luleå Technical University [31]. The objective of Symphony is to build the ideal simulation environment where the models would have a perfect similitude with the real ones. The project is ongoing, the following description reflects its state at the time this work has been done.

The main element in Symphony is the TosNode class. It inherits from the NS-3 base class Node and is an abstract representation of a application developed in TinyOS. TosNodes behave as a virtual machine for the application. The TinyOS image doesn’t know if it is running on a real hardware device or inside the simulator. Symphony allows
the creation of several different nodes and its interconnection building a particular network topology.

The communication between TinyOS and NS-3 is done through bridging proxies (i.e. \texttt{LibToTosProxy} and \texttt{TosToLibProxy}) that create a bidirectional communication channel between the binary image and the simulator. Each \texttt{TosNode} must have a \textit{SymphonyApplication}. This class provides the callback \texttt{ReceiveDataFromApplication} where the information sent inside the node can be collected and processed in NS-3.

Symphony also provides support to simulate sensors connected to the mote. It is possible to set the value and the time that the sensor will receive a new measure from NS-3. This functionality is implemented through the \textit{RawSensor} base class. Each node can have one or more sensors installed.
This chapter describes the design decisions related with the implementation of this project. Section 3.1 presents the architecture of the system with its main components and how are interrelated. In section 3.2 each architectural component is described with its inner details. Finally, section 3.3 shows the details about how binary spatter codes can be use for wireless sensor networks communications describing how nodes encode information and how the base station process it and extract the required knowledge from the network behaviour.

### 3.1 System architecture

There are two different components that should be connected to allow their interaction. In one side, the simulation environment where our application is being tested and in the other side, a frontend interface that interacts with it. The easiest solution is to establish a direct connection between the simulation environment and the frontend interface. Each component exposes an interface with methods that can be called by other component. However, this solution has several limitations; for example, both components are tightly coupled, there isn’t any access control mechanism or it only supports one user and one simulation running at the same time. That’s way, we decided to introduce a middle layer between both components to provide solutions to that limitations. Figure 3.1 shows the architecture proposed.

Communication between the three components is bidirectional, the information can flow forward and backward among them. And the tasks and responsibilities of each component are:

**I/O Client** provides a way to collect the configuration data for the simulation and to show the data and results coming from the simulator. It interacts directly with the I/O Server.
**I/O Server** has two main roles. The first one is as an adaptation layer between the I/O Client and the Simulation Server to allow a transparent communication between them. It provides the mechanisms to adapt the possible differences in data representation, architecture, communication technology, etc. between them. The second one is as a control point implementing support for multiple users and simulations associating each user with a different simulation, error notification between endpoints, security control access, etc.

**Simulation Server** creates the simulation environment to run our application code for the motes inside network simulator. It establishes the connection between the simulator and the I/O Server routing the information coming from I/O Server to the simulation and vice versa.

### 3.2 System implementation

The components in 3.1 represent the high-level architecture of the system. This section introduces a description of each component: what should be inside and how it should interact with the other parts of the system. Figure 3.2 describes a whole view of the implementation details.

#### 3.2.1 I/O Client

The implementation details of the I/O Client depends on the specific client that will interact with the simulation. It should contain at least, three components: one component that collects the configuration information from a source, another component that provides the messages coming from the simulator to the source and the last one, a component that interacts with the I/O Server.

This project, in the showcase scenario in Section 4.1 includes the implementation of a I/O Client for a web based environment.
3.2.2 I/O Server

The I/O Server provides a transparent communication between the I/O Clients and the Simulation Server. It is integrated by three components.

**I/O Client Communication Manager** implements the interfaces to communicate with the clients. It can have one or many different communication interfaces to satisfy different I/O Clients. For example, a web interface connecting through a web socket or another simulator using a tcp socket. Each interface provides support for different I/O Clients.

**Simulation Server Communication Manager** implements the interface to communicate with the Simulation Server. As difference of the I/O Client Communication Manager, there is only one type of simulation server that can connect to the I/O
Server so no multiple interfaces are required. It provides support for connections of different simulations running at the same time on the Simulation Server.

**End-to-end Management** creates the relationship between each I/O Client connected to the I/O Server and its associated simulation. It should adapt the messages to the expected data in the other side of the communication.

The interaction between the I/O Server with the other elements is explained as follow. The initial status of the I/O Server is in execution and the communication managers listening for incoming connections.

**Simulation Server creates a new simulation**
When a new simulation is created inside the simulation server, it should connect automatically to the I/O Server. The End-to-end component should register the new connection to use it when a new I/O Client connects to the server.

**An I/O Client establish a new connection**
When a new client tries to establish a new connection the first step is to check if there is any simulation registered and available, if there is none, the connection should be dropped. However, if a simulation is available the End-to-end component must create a relationship between the client and the simulation and start forwarding the messages between them.

Each connection, for example, can be identified using the IP and its source port (e.g. "127.0.0.1:23032").

**I/O Client or Simulation Server sends a new message**
The End-to-end components gets the related endpoint to the one associated with the message and request the Communication Manager to send the message. If there
is no connection associated, an error message should be sent to the sender and the connection should be drop.

**I/O Client or Simulation Server closes the connection**
The Communication Manager detects that the connection is closed and request the End-to-end component to remove any associated information that it has. The End-to-end component closes the connection with the associated endpoint and release the resources allocated.

This interaction is represented in Fig 3.4 as a flow diagram.

![Flow Diagram](image)

**Figure 3.4: Sequence diagram for the standard use case**

### 3.2.3 Simulation Server

NS-3 has been chosen as the simulation platform for testing the results of our project. It provides a great number of simulation models focused on wireless and IP networks (e.g. Wi-Fi, LTE or WiMAX). However, it lacks of two aspects important for this project.
Support for executing real code inside the simulator. Simulators use models to run simulations, the utility of the results obtained depends on the accuracy of these models. However, another important factor goes into play also, the difference between the code running inside the simulator and the code running in real applications. It is not always possible to have the same one running in both cases. That produces a difference in the results that can lead a problem when the applications are deployed in a real environment. The ability to test the real code inside the simulator can prevent this kind of errors.

Support for interacting with the simulation from the outside. In this case, one of the objectives is to be able to interact with the applications running inside the simulator from the outside. It should be possible to modify the configuration of the simulation, provide values for different parameters and extract the information to be visible in a control panel (i.e. an I/O Client).

In Figure 3.2 is represented the internal implementation of the Simulation Server using NS-3 and two new components added to the simulator that provides the solution to the two previous missing aspects described. The aim of these components is:

I/O Proxy: To provide a external interface of the NS-3 simulator making accessible its configuration mechanism outside the simulator. The interface can be exposed as an API with public methods, as a message protocol to exchange information based on text or with another specific implementation.

Symphony: To provide the environment to execute the TinyOS code inside NS-3.
3.3 Binary Spatter Codes for data representation

On section 2.1 the principles of cognitive codes using hyperdimensional binary vectors also known as binary spatter codes in the literature. Now, this theory will be applied to the case of a cognitive sensor networks using a binary spatter code for communication.

3.3.1 Sensor and actuator nodes

Inside the network, each node needs to be uniquely identifiable to know the source address of the message. In the initialization period, the base station will provide each node two vectors. First one is for randomization, \( P \), it is the same for all the nodes; the second one is for identification and initialization, \( xR_i \) \((x=S\ \text{for sensors, } x=A \text{ for actuators})\), different for each node.

Each time that a node wants to send information, it should encode it following the next steps:

1. Get the information it wants to send (i.e a new measurement value, a new position, etc.)
   - A new temperature measurement of 5°C

2. Generate a new hypervector shifting the initialization vector \( xR_i \) associated with that information.
   - Filler-HyperVector-5 = Shift(sR0, 5)

3. Store the hypervector to reused it every time the same value needs to be sent.
   - 5°C – Filler-HyperVector-5

4. Encode the hypervector with the initialization vector \( xR_i \) using the bitwise OR (||) function in order to be able to retrieve which node is sending the message.

5. Encode the resulting hypervector with the randomization vector \( P \) using the bitwise XOR (\( \otimes \)) function.
   - \( P \otimes (sR0 || Filler-Hypervector-5) \)

6. Send the resulting hypervector to the base station.

3.3.2 Base Station

The base station is the coordinator inside the network doing at the beginning the initialization of the different nodes and processing later the information coming from them while the network is working.
The aim of the project is to build an autonomous network that is able to configure itself and learn from the behaviour of the nodes to take its own decisions. In our specific case, the network contains sensors that measure the temperature of a room and track how the thermostat (actuator) changes while the measurements on the sensors changes. The idea is to teach the base station that an specific sequence on temperature changes produces a specific change on the actuator. After the base station learns this rule, it should be able to change the actuator status to the desired position when the same temperature change pattern is received.

**Initialization mode**

In the bootstrap time of the network the base station generates the common random vector $P$ and the specific initialization vector $xR_i$ for each node. Vectors are high dimensional with a dimension $D$ bigger than 1,000 bits. They should be generated randomly. Each position inside the vectors should be independent and identically distributed drawn for the normal distribution with mean 0 and variance $1/D$. The resulting vectors will have an equal number of 0’s and 1’s inside.

Base station stores the $P$ and $xR_i$ vectors on its internal memory to be able to identify later which node is sending the incoming vector. Also, an associative array is used to keep the relationship between the $Id$ of the node and its $xR_i$ vector. An example of this array is included on Fig 3.5, it is called the *init memory*.

<table>
<thead>
<tr>
<th>Device Id (i)</th>
<th>Role Vector (xRi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0 0 1 ... 0 0 1 0 0 ... 1 1 0</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 ... 1 0 1 1 0 ... 0 1 0</td>
</tr>
<tr>
<td>3</td>
<td>1 0 1 0 ... 0 1 1 0 0 ... 1 0 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 3.5: Init memory structure

After the initialization is completed, the base station starts working in its normal operation mode.

**Normal operation mode**

In the normal operation mode the base station keeps listening for incoming vectors from the nodes inside the network. Base station contains two buffers for incoming vectors, the integer buffer and the binary buffer. The integer buffer contains the bitwise sum of
the actual and the previous states of the network adding the different vectors received together. Its initial value is 0.

In order to work with binary numbers, the integer buffer is normalized to obtain the binary vector associated. The threshold operation is done obtaining the average value of the positions inside the integer buffer; if the value on a position is bigger than the average the binary value will be one and if its lower, the binary value will be zero. The result of this operation will be stored on the binary buffer.

A graphical representation of the process is available on Fig 3.6 that represents the arrival of three different hypervectors on different times $t$. They are added together inside the integer buffer and the binary buffer is extracted after the previous defined threshold operation.

| $t = t_1$ | 10101...10 |
| $t = t_2$ | 01110...10 |
| $t = t_3$ | 11101...00 |
| Majority Rule | $> 1.5$ |
| Binary Buffer | 11101...10 |

Figure 3.6: Threshold operation of the binary buffer

Inside the base station is possible to have different running states depending if the networks has just been initialized or it has been running for a period of time. The following points describe the behaviour of the base station in these cases:

**Item memory is empty**

The item memory is the data structure that stores the learned patterns from the network behaviour. This situation happens when the base station has finish the initialization scenario and starts running on normal mode. In that case, no vector has been received yet so everything inside the base station doesn’t contain any information. The process that happens inside the base station when a new vector is received are:
1. Check if the input vector is new or it has been received before.
   - This verification is done applying the $\otimes$ operation between the binary buffer and the input vector. If the Hamming distance between both is lower than 0.20, it means that this vector has been received before and it is contained in the buffer. The threshold is set based on the properties of the similarity between codewords in binary spatter codes. Due to in this case the received vector is the first one that the base station processes the execution continue in the following step. In the other scenarios, if the vector has been received before the information is discarded.

2. Remove the $P$ vector from the input vector.
   - result = $P \otimes (P \otimes (xR_i \parallel Filler-Hypervector)) = xR_i \parallel Filler-Hypervector$

3. Look for the node that has sent the message.
   - Due to the properties of the OR operation is possible to extract the id of the node that sent the message calculating the hamming distance between $(xR_j \otimes (xR_i \parallel Filler-Hypervector))$. If the base station cycles along the $xR_i$ vectors stored in the item memory only when $sR_j = sR_i$ the value of the Hamming distance will be $\approx 0.20$. In the other cases, the value will be $\approx 0.50$. In the worst case scenario, the base station should make $(s + a + 1)$ checks to find the node that is sending the message where $s$ is the number of sensors and $a$ the number of actuators.

4. Look for the information contained in the filler.
   - Nodes encode the information shifting $x$ positions its initialization vector $xR_i$. In the previous step the id of the sender was extracted so it is possible to know its vector $xR_i$. If the vector $xR_i$ is shifted $x$ positions and compared with the $(xR_i \parallel Filler-Hypervector)$ vector it is possible to find the value of $x$ using again the result of the Hamming distance lower than $\approx 0.20$

5. Check the type of node that sent the message
   - If the id of the node is associated with a sensor, the received vector contains new information about the changes on the network so it should be stored. The first step is update the content of the integer buffer adding the input vector and performing the threshold operation to update the binary buffer. The next one, is related with the learning process.

The learning process implemented uses a two steps-back memory that makes possible to track the changes from one situation to another and not relying only on the actual state. In other words, the change of the temperature between $0^\circ C$ to $5^\circ C$ will be treated different than the change from $4^\circ C$ to $5^\circ C$. There are two data structures used in this process, the sensor state associative array and the item pending register. The first one keeps track of the actual state of each sensor available on the network with its id and the last value of $x$ received (Fig. 3.7).
3.3. **Binary Spatter Codes for data representation**

The second one, build a temporal register with the data that will be introduced in the *item memory* when a change on the actuator node is received. The register contains the following information: the id of the node associated with the current value of the shift, the id of the node associated with the previous value of the shift, and a copy of the input vector of the sensor (Fig. 3.8). The other field is explained in the next point.

![Table](sensor_data_representation.png)

**Figure 3.7: Sensor state structure**

Both data structures are filled and updated with the correct values before the process finish for this case.

- If the id of the node is associated with an actuator the next step is checking if the *item pending* register is empty or not. If is empty because no sensor information has been received yet, the process finish. If it contains information, the empty field actuator vector will be filled with the received vector and the *item pending* register will be stored as a new pattern inside the *item memory*.

**Item memory doesn’t contain the input pattern**

In this case, the item memory contains at least one pattern that associates a sequence of changes on the sensors with a change on the actuator. The process is the following:

1. Check the incoming vector with all the patterns stored in the item memory to find a match.

2. Due to the pattern is new, no match will be found. Continue with the first step of
previous scenario on 3.3.2

**Item memory contains the input pattern**

In this case, the item memory contains at least one pattern as before that matches the input pattern. The process is the following:

1. Check the incoming vector with all the patterns stored in the item memory to find a match.

2. A match with a stored pattern is found when the Hamming distance between the input vector $\otimes$ with the stored pattern is $\approx 0.20$.

3. If the values of the previous shift and the current shift of the sensor stored as explained on 3.3.2 the match is correct.

4. Retrieve the associated actuator vector stored on the *item memory* register and update the values of the integer and binary buffer.

5. Send the actuator vector to the actuator node. The node will change its position extracting the information from the received hypervector in the same way we did on the third step in section 3.3.2.

This final step represents the learning process that happened inside the base station. It has been able to reproduce the action learned processing only the random information received from the nodes in the structure of a hypervector.

The base station doesn’t need to take care about the types of the sensors or the type of actuators, it only uses the random hypervectors as an information source for learning.

More details about the code implementation of the different can be found in the Appendix D
4.1 Scenario Description

This section explains a real implementation of a showcase scenario based on the design explained on 3 using web technologies. Users can configure the simulation and receive information from the simulator using a web browser as I/O Client. The I/O Server is built as a python web server that communicates with the I/O Client using websockets and with the Simulation Server through a TCP Socket. Simulation Server runs over a Linux machine with the ns-3 network simulator installed.

4.1.1 I/O Client

The I/O Client is built using web technologies; it should be executed from a web browser that supports HTML5, CSS3, JavaScript and Websockets (In this case, Google Chrome v.26 has been used as a testing browser). Mobile device browsers, like iOS Safari or Chrome for Android, are also a viable platform.

To simplify the development of the component, the following libraries has been used:

**jQuery (v.1.9.1):** A small Javascript library that simplifies HTML manipulation, the event handling system and the creation of animations among others. It is a cross-browser library that provides compatibility with the latest versions of the five main web browsers in the market: Internet Explorer, Chrome, Firefox, Safari and Opera [33].

**jQueryUI (v.1.10.2):** An extension of jQuery that provides a collection of user interface elements, interactions, widgets, effects and themes [34].

**Twitter Bootstrap (v.2.3.1):** A frontend framework that provides resources like a grid system, layouts, responsive design, CSS styles and other components for faster
and easier web development [35].

The communication with the I/O Server is done by the Websocket API [36] exposed by the web browser through Javascript. Listing 4.1 includes the full definition of the interface. In our case, we are interested in the events related with networking and messaging and the option to send data as a String.

```javascript
enum BinaryType { "blob", "arraybuffer" }
    [Constructor(DOMString url, optional (DOMString or DOMString[]) protocols )]
interface WebSocket : EventTarget {
    readonly attribute DOMString url;

    // ready state
    const unsigned short CONNECTING = 0;
    const unsigned short OPEN = 1;
    const unsigned short CLOSING = 2;
    const unsigned short CLOSED = 3;
    readonly attribute unsigned short readyState;
    readonly attribute unsigned long bufferedAmount;

    // networking
    attribute EventHandler onopen;
    attribute EventHandler onerror;
    attribute EventHandler onclose;
    readonly attribute DOMString extensions;
    readonly attribute DOMString protocol;
    void close([Clamp] optional unsigned short code, optional DOMString reason);

    // messaging
    attribute EventHandler onmessage;
    attribute BinaryType binaryType;
    void send(DOMString data);
    void send(Blob data);
    void send(ArrayBuffer data);
    void send(ArrayBufferView data);
}
```

Listing 4.1: Websocket API Definition

When the user interface has finished its rendering inside the web browser, the web socket is initialized. The frontend reacts to the Websocket events in the following way:

**onopen (line 15):** the socket is ready to start sending and receiving information. It makes available the options of the user interface to start configuring the environment and set the connection status indicator as "Connection Established" as 4.1 shows.
onerror (line 16): an error has happened in the connection between the I/O Client and the I/O Server. The I/O Client sets a timer and tries to connect again each 5 seconds. It updates the status indicator to notify that the connection is not available and disables the options of the user interface.

onclose (line 17): the connection has been closed by the I/O Server. It happens when the simulation has finished and no more interaction is available.

onmessage (line 23): a new message is received from the I/O Server. It is processed and the user interface updated to reflect the new incoming information.

If a message needs to be sent, the code calls the method `Send(DOMString data)` to pass it through the web socket. Encoding the information as strings allow us to use plain text configuration files making them easier to modify and debug. It also allows to create more complex definitions using structured languages like JSON or XML.

### 4.1.2 I/O Server

As explained in 3.1, one of the main roles of the I/O Server is an adaptation interface between the other two components. The technology used for its development is Python (v.2.7.4) [37]. To simplify the management of sockets, connections and protocols, two open source python libraries are used:

**Twisted (v.13.0):** an event-driven networking engine that allow writing servers and clients for common network protocols like HTTP, SMTP, POP3 pr SSH. It helps to create scalable, small-footprint and easy to configure web servers to serve from simple to high-traffic websites [38].
**AutobahnPython (v.0.5.6):** is a library that extends Twisted and provides support for The Websocket Protocol and The WebSocket Application Messaging Protocol (WAMP). [39].

Before continuing with the description the I/O Server, three concepts from Twisted should be introduced to understand better how it works. A detailed explanation can be found in Twisted’s documentation [40].

**Reactor:** is the main component inside Twisted. It is the responsible of providing the basic interfaces related with threading or network communications over different platforms. It interacts directly with the event loop, the programming construct responsible of waiting and dispatching events (e.g. a new connection has been established or a new message has arrived). The I/O Server runs inside that event loop and interacts with the events that the reactor generates.

**Factory:** a factory class is associated with a socket although it does not listen for incoming connections. When a new connection is received in the socket, the reactor calls the `buildProtocol` method inside its associated factory and creates a new `Protocol` object that will be the responsible of interact with the new client.

**Protocol:** a new instance of this class is instantiated each time a new connection is established. The life time of the object is until the connection is finished. It responds to the events that came from the reactor through calls to methods on the protocol interface. It doesn’t keep state between different connections so any information should be stored in the `Factory`.

After this introduction, Figure 4.2 represents the architecture of the I/O Server.

![Figure 4.2: I/O Server server architecture](image)

The server has two endpoints. The first one listen for web socket connections from the I/O Client in port 9000, the second one listen for TCP socket connections from the
Simulation Server in port 9999. The default values can be modified when the server is launched. Each time the reactor receives a new connection in any of the endpoints it calls the associated factory to create the protocol that will manage the connection.

**Factories**

There are two factories: `Ns3TcpServerFactory` and `BrowserWebSocketServerFactory`. The first one inherits from the base class `Factory` from Twisted and it is the responsible of creating the protocol objects that will interact with the simulations running on the Simulation Server. The second one inherits from `WebSocketServerFactory` base class from AutobahnPython, it is responsible of creating the protocols that interact with the I/O Clients.

Both factories are dummy classes; they don’t modify the behaviour of the base classes. However, they have been introduced to make easier to modified the way that protocol objects are created or to persist configuration information in case if it is needed in future updates.

**Protocols**

As in previous section, there are also two protocols: `Ns3TcpProtocol` and `BrowserWebSocketProtocol`. Each one is created by its associated factory. The main role of the protocols is to subscribe to the events related with a new connection has been established, a new message has been received and the connection has been lost.

`BrowserWebSocketProtocol` inherits from the base class `WebSocketServerProtocol` from AutobahnPython. It overrides the `onConnect`, `onMessage` and `onConnectionLost` events and notify the application when any of them are called.

In the same way, `Ns3TcpProtocol` inherits from the class `LineReceiver` from Twisted. It is a protocol that allow the manipulation of text-based protocols that receive the information as a line and/or raw data. Our implementation overrides the `connectionMade`, `rawDataReceived` and `connectionLost` events and notify also the application when any of them are called.

**End-to-end management**

The End-to-end management application is the responsible of configuring the endpoints and initializing the server to start listening in the ports. It also tracks the relationship and passes the messages back and forward between each I/O Client with its associated simulation inside the Simulation Server. When the system starts, the application subscribes itself to the events raised by the protocol factories and waits for incoming connections.

More details about the code implementation of the different parts of I/O Server can be found in the Appendix A.
4.1.3 Simulation server

NS-3 is extended using the following two components to provide the support needed to this work.

I/O Proxy

In order to allow a node inside a simulation in NS-3 interact with other nodes in a real network, NS-3 includes a specific net device called *Emu*. The computer where the simulator is running should have a network interface that supports promiscuous mode (i.e. it can be configured to listen all the traffic inside the network, not only the packets addressed to itself). Using MAC spoofing, NS-3 is able to create a virtual host inside the network to receive and send packets. MAC spoofing consists in the generation of a virtual MAC address that is assigned to the *Emu* net device making it visible in the network as an independent device.

The communication is based on the implementation that NS-3 provides of the Socket Application Programming Interface (API) based on the Berkeley sockets API \(^1\). The main difference with the Berkeley API is that the one provided by NS-3 is asynchronous. The calls are not blocking and the interaction is done by the callbacks provided by the *Socket* class.

These two native components inside NS-3 are use to build the I/O Proxy that will interact between the code running inside our simulation and the I/O Server.

List 4.5 enumerates the callbacks available in NS-3 *Socket* class.

```c
1 void SetConnectCallback (Callback<void, Ptr<Socket>> >
   connectionSucceeded,
   Callback<void, Ptr<Socket>> > connectionFailed);
3 void SetCloseCallbacks (Callback<void, Ptr<Socket>> > normalClose,
   Callback<void, Ptr<Socket>> > errorClose);
5 void SetAcceptCallback (Callback<bool, Ptr<Socket>, const Address &>
   connectionRequest,
   Callback<void, Ptr<Socket>, const Address&>
   newConnectionCreated);
7 void SetRecvCallback (Callback<void, Ptr<Socket>, uint32_t> dataSent);
9 void SetSendCallback (Callback<void, Ptr<Socket>, uint32_t> sendCb);
```

Listing 4.2: Callbacks exposed by Socket NS-3 Class

They are used in the following way:

**Connect Callbacks (line 1):** when the *Connect* method is called, the socket tries to initialize the connection with the remote server. There are two callbacks available

\(^1\)The details about this API are out of the scope, for a deeper understanding there are several books available to the reader [41][42].
that notify if the connection has been made or it has failed. In our I/O Proxy, if the connection succeed the simulation continues but if it fails, the I/O Proxy retries the connection until it succeeds.

**Close Callbacks (line 3):** there are two callbacks related with closing a connection. The first one is called when the other end closes, and the second one is called when an error has happened and the connection closes abnormally. In both cases, the I/O Proxy stops sending information back to the I/O Server and ends the simulation.

**Accept Callbacks (line 5):** in the actual implementation, the I/O Proxy acts as a client connecting to the I/O Server. However, it is possible to change the behaviour and act as a server waiting for the connection of the other component. Support to this option has been included in case if it is needed in future work.

**Receive Callback (line 9):** it is used to notify the I/O Proxy that new information is ready in the socket to be read. It is used to extract the information coming from the I/O Server to be used inside the simulator.

**Data Sent Callback (line 10):** notifies the I/O Proxy that the packet has been sent. It is not used in the actual implementation.

**Set Send Callback (line 11):** notifies when new space in the transmit buffer is added. It is not used in the actual implementation.

**Symphony**

The basic element is the *TosNode* class. It represents a node of the network running the TinyOS code of the application; it acts as a virtual machine. TinyOS code doesn’t know if it is running on a hardware device or in the simulator. For each node in the simulation, a new *TosNode* should be created and stored in a *TosNodeContainer*. This class allows a simpler manipulation of all the nodes.

The communication between TinyOS and NS-3 is done after installing a *SymphonyApplication* inside the node. This class provides a callback *ReceiveDataFromApplication* where the information sent inside the node can be collected and processed in NS-3.

Finally, it is possible to emulate sensors measuring some parameters like temperature and passing that information to the TinyOS code through the *RawSensor* base class. Each node can have one or more sensors stored inside a *SymphonySensorContainer*.

To make it easier accessing the information of the nodes inside different parts of NS-3, the *Names* class is used. It creates a dictionary relationship that if the name of the objects is used as a key, a pointer to the object is returned.

For more details about the code implementation of the I/O Proxy and its Socket and Symphony components, it is included in Appendix B
4.1.4 Message protocol

The communication between the different components of the system is done with a plain text based protocol. There are two types of messages:

- **A change of the status due to an interaction of the user with the I/O Client.** The message contains the object type identification name, the identification number that makes the object unique on the system and a the new value of the object. The text string is built in the following way:
  
  \[ \text{identificationName-identificationNumber:newValue} \]

  For example, `temperatureSensor-0:1` tells the Simulation Server that the new value of the `temperatureSensor` with identification number 0 is 1.

- **An action messages sent from the Simulation Server to the I/O Clients to update the status of a specific object.** The message contains the target object type identification name, the identification action name, the identification number of the object and the new value assigned. The text string is built in the following way:
  
  \[ (\text{identificationName+idActionName})-\text{identificationNumber:newValue} \]

  For example, the message `actuatorChange-0:1` tells the I/O Client to `change` the value of the `actuator` with identification number 0 to 1.

4.2 Testing

The tests made on our showcase scenario have been focused on demonstrate the correct behaviour of the cognitive sensor network and get a summary of two key performance indicators: execution time and memory consumption.

4.2.1 Behaviour of the network

The core of the sensor network is the base station and is learning algorithm. At the starting point, the base station distributes the initialization information among all the nodes inside the network. After that, it receives the hypervectors coming from the nodes and uses the cognitive algorithm implemented to perform different actions.

On Section 3.3, the theoretical analysis of the learning algorithm has been developed. The goal of these tests was to show its correct behaviour in a real implementation on code through the interaction with our prototype.

**A sensor and an actuator**

On this scenario, two nodes are deployed on the network. One of them acts as a sensing node and the other acts as an actuator node. From the I/O Client the values of the
sensor and actuator are modified and they are sent to the simulator. When a result is available the I/O Client refresh the current status to make the changes visible to the user.

The steps followed are:

1. Connect the I/O Client to a running simulation on the Simulation Server. It is possible to see the *Connection Established* message on the top similar to Fig 4.3.

![Figure 4.3: I/O Client interface initial screen](image)

2. Add a temperature sensor node and an actuator node using the options available on the left side bar. Two elements will appear on the dashboard similar to Fig 4.4.

3. Introduce the changes on the simulation moving the slider of each object. The values introduced were:

   (a) Change temperature sensor value from 0 to 5.
   (b) Change actuator position from 0 to 5.
   (c) Change temperature sensor value from 5 to 0.
   (d) Change actuator position from 5 to 0.
This sequence records two patterns inside the base station: the change from 0 to 5 and from 5 to 0. The next time the base station receives any of the previous patterns it orders the actuator to change its position to the associated with the pattern. So, if the slider of the sensor is moved again from 0 to 5, the actuator should change its position to 5.

4. Change the value of the temperature sensor from 0 to 5. The position of the actuator will be updated and the background of the object will blink to notify the change to the user similar to Fig 4.5.
Two sensors and two actuators

This scenario is an advanced version of the previous one where four nodes are deployed on the network. Two of them act as sensing nodes and the other two act as actuator nodes. As before, the values of the sensors and actuators are modified from the I/O Client and when a result is available the I/O Client refresh the current status to make the changes visible to the user.

The steps followed this time are:

1. Connect the I/O Client to a running simulation on the Simulation Server. It is possible to see the *Connection Established* message on the top similar to Fig 4.3.

2. Add two temperature sensor nodes and two actuator nodes. The components inside the dashboard can me moved around its surface using the technique of *drag and drop*. In this case, they have been arranged in a better position to manage them (Fig 4.6).

![SkyNet Web UI](image)

Figure 4.6: I/O Client interface with two sensors and two actuator elements

3. Introduce the changes on the simulation moving the slider of each object. The values introduced were:

   (a) Change the first temperature sensor value from 0 to 5.
   (b) Change the first actuator position from 0 to 5.
   (c) Change the second temperature sensor value from 0 to 5.
   (d) Change the second actuator position from 0 to 5.
(e) Change the first temperature sensor value from 5 to 0.
(f) Change the first actuator position from 5 to 0.
(g) Change the second temperature sensor value from 5 to 0.
(h) Change the second actuator position from 5 to 0.

This sequence records four patterns inside the base station: the changes from 0 to 5 and from 5 to 0 of the two sensors and actuators. The next time the base station receives any of this patterns, the user interface will be refreshed as in the previous case. Any change on the sensors from 0 to 5 or vice versa will produce a refresh on the user interface with the new actuator position.

Other scenarios

It is possible to see the correct behaviour of the algorithm in more complex scenarios. For example, two sensors nodes interacting with one actuator node, one sensor interacting with two actuators and more that two sensors and actuators interacting together. The steps to reproduce these scenarios are similar to the previous two cases explained in detail. It is left to the reader to test them if he is interested on visualize how the interface is updated with the changes.

Testing environment

These test has been deployed in two environments: a local environment based on a personal laptop and a cloud environment on Amazon Web Services

- **Local environment**: the laptop has a 2.2 Ghz Intel Core 2 Duo with 4GB of RAM shared between the host operating system and a virtual machine running Ubuntu LTS 12.04. The virtual machine acts as the Simulation Server running the NS-3 simulator. The host operating system runs the I/O Client and the I/O Server.

- **Cloud environment**: two Amazon EC2 instances has been used to recreate the local environment on the cloud. The first instance is a c1.medium with 2vCPU and 1.7GB of RAM running a Ubuntu LTS 12.04, it contains the the Simulation Server running the NS-3 simulator. The other instance is also a c1.medium with similar configuration that runs the I/O Server and the web server to provide remote access to the user. He can connect to the simulation through his web browser using the IP of the machine and the port number 80.
4.2. Testing

4.2.2 Performance

The two factors that will affect the viability of the cognitive networks in a real scenario are:

- The time that it is needed to perform the hypervector encoding and send it to the base station.
- The amount of memory that is used in these two processes.

In the following sections it is explained how the data has been collected.

Time Performance

The time cost of an operation executed on a TinyOS node can be obtained theoretically knowing the details of the hardware where the program is going to be executed. In this case, due to the availability of a Mulle\[43\] mote using TinyOS as its operating system, real measures were performed to extract the information.

Results has been extracted using a power consumption measurement board developed for the Mulle mote. The board tracks the current consumption of the node. The data is sampled using a Matlab application that provides the values of the current and the voltage consumed and plots them as figures. For each of the measures we were interested on, a TinyOS application performing only the specific task has been deployed to the mote and after that, its power consumption has been measured. The details of each case are in the following subsections.

Operations  When a sensor node receives a new measure or when an actuator node receives a change in its position, the node needs to encode that information and generate the associated hypervector to be transmitted. The operations performed in the encoding process are the elementary binary operations.

The following TinyOS application makes the binary XOR operation over two hypervectors of 1024 bytes five seconds after all the system has been initialized to avoid errors in the measurement.

```c
#include "Timer.h"
#define DIMENSION 64 // 16 bits number * 24 = 1024 bits

module BlinkC @safe()
{
    uses {
        interface Boot;
        interface Random;
        interface Timer<TMilli> as MilliTimer;
        interface Leds;
    }
}
```
implementation
{
    uint16_t hv1[DIMENSION];   // Hypervector 1
    uint16_t hv2[DIMENSION];   // Hypervector 2
    uint16_t hvres[DIMENSION]; // Hypervector result

    /**
     * Executes the XOR operation between the hypervectors
     */
    void task xor()
    {
        uint16_t i = 0;
        for (i = 0; i < DIMENSION; i++)
        {
            hvres[i] = hv1[i] ^ hv2[i];
        }
    }

    /**
     * Configures the time to execute the task when is fired
     */
    event void MilliTimer::fired()
    {
        post xor();
    }

    /**
     * Initializes the hypervectors to random values and configures
     * the timer to be fired on 5 seconds
     */
    event void Boot::booted()
    {
        // Random values
        for (i = 0; i < DIMENSION; i++)
        {
            hv1[i] = call Random::rand16();
            hv2[i] = call Random::rand16();
        }
        call MilliTimer::startOneShot(5000);
    }
}

Listing 4.3: TinyOS binary operation test application

On Fig 4.7, the voltage consumed by the mote versus the time is plotted. It contains a
time slot of 10 seconds that allows to see the behaviour of the mote along all the execu-
tion of the program. At the beginning, a peak near 1V is visible when the mote performs
the generation of the hypervectors. After that, it stays on a low power state until the
timer is fired around the 5th second and the XOR operation is performed.

Zooming the previous picture to the range around 5 seconds (Fig. 4.8) is possible to
4.2. Testing

Figure 4.7: Time expended on XOR operation

effect the time expended by the mote. The measurements over the graph provided an estimation of \( \approx 1 ms \)

Figure 4.8: Time expended on XOR operation
Transmission After the encoding has finished, the next step is sending the information to the base station. The actual TinyOS implementation provides a message structure for radio interfaces that it is defined as:

```c
typedef struct TOS_Msg {
    // The following fields are transmitted/received on the radio.
    uint16_t addr;
    uint8_t type;
    uint8_t group;
    uint8_t length;
    int8_t data[TOS_DATA_LENGTH];
    uint16_t crc;

    // The following fields are not actually transmitted or received
    // on the radio! They are used for internal accounting only.
    // The reason they are in this structure is that the AM interface
    // requires them to be part of the TOS_Msg that is passed to
    // send/receive operations.
    uint16_t strength;
    uint8_t ack;
    uint16_t time;
    uint8_t sendSecurityMode;
    uint8_t receiveSecurityMode;
} TOS_Msg;
```

Listing 4.4: TinyOS message structure

Taking into account only the information is sent there are 7 bytes used by the addr, type, group, length and crc fields and 121 bytes available as payload to transmit data (e.g the maximum size of the packet is 128 bytes). Using this information, the following applications was deployed to the Mulle mote to extract the time needed to send the packet over the radio interface.

```c
#include "Timer.h"

#define TOS_DATA_LENGTH 121

typedef nx_struct radio_msg {
    nx_uint8_t data[121];
} radio_msg_t;

enum {
    AM_RADIO_MSG = 6,
};

module RadioToLedsC @safe () {
```
uses {
    interface Leds;
    interface Boot;
    interface Receive;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface SplitControl as AMControl;
    interface Packet;
}

implementation {

    message_t packet;
    bool locked;

    /**
     * Initialize the radio interface when the node is booted
     */
    event void Boot.booted()
    {
        call AMControl.start();
    }

    /**
     * Configures the timer to send the packet after the
     * radio interface is ready
     */
    event void AMControl.startDone(error_t err)
    {
        if (err == SUCCESS) {
            call MilliTimer.startPeriodic(10000);
        } else {
            call AMControl.start();
        }
    }

    event void AMControl.stopDone(error_t err)
    {
        // do nothing
    }

    /**
     * Sends the packets when the timer is fired
     */
    event void MilliTimer.fired()
    {
        int i = 0;
        if (locked) {
            return;
        }
        else {

radio_msg_t* rcm = (radio_msg_t*)call Packet.getPayload(&packet, sizeof(radio_msg_t));

if (rcm == NULL) {
    return;
}

if (call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(radio_msg_t)) == SUCCESS) {
    dbg("dbg", "dbg: packet sent.\n");
    locked = TRUE;
}
}
}

/**
 * Receives the messages from the network
 */

event message_t* Receive.receive(message_t* bufPtr, void* payload, uint8_t len) {
    if (len != sizeof(radio_msg_t)) {
        return bufPtr;
    }
    else {
        int i = 0;
        radio_t* rcm = (radio_msg_t*)payload;
        return bufPtr;
    }
}

/**
 * Event raised when the packets is sent
 */

event void AMSend.sendDone(message_t* bufPtr, error_t error) {
    if (&packet == bufPtr) {
        locked = FALSE;
    }
}
}

Listing 4.5: TinyOS radio test application

On Fig 4.9, the voltage consumed by the mote versus the time is plotted. It contains again a time slot of 10 seconds to see the behaviour of the mote along all the execution of the program. At the beginning, the radio interface is started increasing the power consumption of the mote to near 1.6V. After that, it stays on that state until the timer is fired around the 5th second and the packet starts to be sent.
4.2. Testing

Figure 4.9: Time expended sending the packet

Zooming the previous picture to the range around 5 seconds (Fig. 4.10) is possible to extract the time expended by the mote. The measurements over the graph provided an estimation of $\approx 50ms$.

Figure 4.10: Time expended sending the packet
CHAPTER 5
Discussion

At the beginning of the thesis we introduce the two goals of our project that we defined when this work started:

1. Develop a cognitive application that implements distributed representations as a communication mechanism. The example application used is a small temperature control system with one or two temperature sensors and actuators.

2. Build a demonstrator to interact visually with the previous application from a web browser interface. Make it also extensible to replace the web interface for other different user interface.

Now, it is the time to pick them up and provide the answers extracted from the work done. On section 5.1, the results of using a distributed representation on WSNs for communication purposes are analysed. We summarize its performance, viability and limitations. Finally, section 5.2 shows the future steps to follow in these research areas.

5.1 Using distributed representations as a communication mechanism

Performance The two more demanding tasks using a distributed representation is the generation and processing of the information as hypervectors inside the nodes, and its transmission over the wireless channel to the base station. All the work has been based on high dimensional vectors of dimension 1000 bits.

The total time of generating and sending one hypervector can be approximated by \( \approx 60ms \) including the results obtained on real measures plus a little overhead of other minor operations not included in those test. However, using a common channel produces interferences and collisions between different nodes trying to send at the same time. As example, using a data payload of 121 bits as in 4.4, the previous time for sending a
packet, and the simplest Medium Access Protocol (MAC) called ALOHA the probability of collision depending on the number of nodes is plotted for three different cases on Fig 5.1.

![Figure 5.1: Probability of collision for a specific number of nodes](image)

In traditional packet-based WSNs the nodes will wait and resend the packets until the message has arrived; however, using distributed local representation and its tolerance to losses it is possible to miss part of the information and still recover the data on the base station. The details about how the nodes access the channel and its performance is out of the scope of this work.

**Viability and limitations**  The prototype developed shows the viability of using distributed representations inside WSNs to communicate information. Although the scenarios addressed may being seen simple, they can represent a control of a real process like an air conditions system that regulates the temperature inside a bedroom at home or a cold room inside a fabric, or a light system that measures the amount of light that is available and turns on or off the lights. The most important thing is that the network doesn’t need to know what it is the system it is controlling, it only takes care about the sequence of random data sent as hypervectors from sensors and actuators. Autonomously is capable of creating the relationship between them and reproduce the actions the next time it is observed.

This project is an overview pilot following the previous research on the department [32] more focused on analysing the viability of the system rather that finding the possible limitations that will be postponed for future work.
5.2 Future Work

In general, the prototype developed can be described as a high level implementations where several elements have been simplified because they were not directly related with the problem addressed. After checking that the prototype is viable several questions have risen that should be addressed in the future.

- How the network will perform using a radio device model instead the capabilities of Symphony for sending and receiving data directly between TinyOS and NS-3?
- How the network will perform in a real deployment using a TinyOS node like Mulle?
- How the increment of the size of hypervectors from 1,000 bit to bigger dimensions like 10,000 or 20,000 bits can improve the tolerance in noisy and harsh environments?
- Will appear hardware devices capable of using the principles of distributed representations instead of building the theory over packet networks as it has been done in this work?
- How a visual interface as a web page inside a web browser can improve the performance of working with simulators?

Open questions that I’m sure we will have an answer sooner than later.
A.1 Endpoints configuration

```python
def __init__(self, wsHost="localhost", wsPort=9000, tcpSockPort=9999):
    """ Initialize the web socket server and the TCP socket. It is possible to define
    the url and port of the web socket and the url and port of the TCP socket
    """
    self.wsHost = wsHost
    self.wsPort = wsPort
    self.tcpSockPort = tcpSockPort

    try:
        # Initialize the TCP socket between server and NS3 simulator
        self.ns3Factory = Ns3TcpServerFactory()
        self.ns3Factory.protocol = Ns3TcpProtocol

        # Set the callbacks
        self.ns3Factory.onNs3Connection = self.onNs3Connection
        self.ns3Factory.onNs3Disconnection = self.onNs3Disconnection
        self.ns3Factory.onNs3Msg = self.onNs3Msg

        self.ns3Socket = serverFromString(reactor, "tcp:port=" + str(self.tcpSockPort))
        self.ns3Socket.listen(self.ns3Factory)

        # Initialize the Web Socket server to communicate with the web browser
```
self.wbSocket = BrowserWebSocketServerFactory("ws://" + self.wsHost + ":" + str(self.wsPort), debug=False)
self.wbSocket.protocol = BrowserWebSocketProtocol

# Set the callbacks
self.wbSocket.onWebBrowserConnection = self.onWebBrowserConnection
self.wbSocket.onWebBrowserDisconnection = self.onWebBrowserDisconnection
self.wbSocket.onWebBrowserMsg = self.onWebBrowserMsg

listenWS(self.wbSocket)

except socket.error, msg:
    print "Failed to create socket. Error code: " + str(msg[0]) + " ,
    Error message : " + msg[1]
sys.exit()
except CannotListenError, msg:
    print "Failed to create web socket server . Error code: " + str(msg.
    socketError[0]) + " , Error message:" + str(msg.socketError[1])
sys.exit()

Listing A.1: TCP Socket and Web Socket initialization

A.2 Protocols implementation

from twisted.python import log
from autobahn.websocket import WebSocketServerFactory, WebSocketServerProtocol

class BrowserWebSocketProtocol(WebSocketServerProtocol):
    """ Defines the interaction through a web Socket with the web browser
    client.
    Each time a new connection made, connection lost or data received event
    is
    raised, it is forwarded to the factory to notify the Backend Server.
    """

def onConnect(self, connectionRequest):
    self.factory.onWebBrowserConnection(self)
    log.msg(self.__class__.__name__ + ": Connection made")

def connectionLost(self, reason):
    self.factory.onWebBrowserDisconnection(self)
    log.msg(self.__class__.__name__ + ": Connection lost")

def onMessage(self, msg, binary):
    self.factory.onWebBrowserMsg(self, msg)
    log.msg(self.__class__.__name__ + ": Msg received: " + msg)
Listing A.2: Browser websocket protocol implementation

```python
from twisted.python import log
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver

from autobahn.websocket import WebSocketServerFactory, WebSocketServerProtocol

class BrowserWebSocketServerFactory(WebSocketServerFactory):
    """ Dummy class inherited from base class in case the base behaviour
    should be modified """
    pass

class Ns3TcpProtocol(LineReceiver):
    """ Defines the interaction through a TCP Socket with the NS3 Simulator.
    Each time a new connection made, connection lost or data received event
    is
    raised, it is forwarded to the factory to notify the Backend Server.
    """

def connectionMade(self):
    # If Ns3 will send lines ending in \n\r the following line should
    # be removed
    self.setRawMode()

    self.factory.onNs3Connection(self)
    log.msg(self.__class__.__name__ + " : Connection Made")

def connectionLost(self, reason):
    self.factory.onNs3Disconnection(self)
    log.msg(self.__class__.__name__ + " : Connection Lost")

    # If Ns3 will send lines ending in \n\r the following method should
    # be replaced for this one:
    #
    # def lineReceived(self, msg):
    #    self.factory.onNs3Msg(self, msg)
    #    log.msg(self.__class__.__name__ + " : Line received")

    def rawDataReceived(self, data):
        self.factory.onNs3Msg(self, data)
        log.msg(self.__class__.__name__ + " : Data received: " + data)

class Ns3TcpServerFactory(Factory):
```

Listing A.3: Ns3 TCP socket protocol implementation

A.3 Management of the events from I/O Client

```python
# Management of the connections received from the web browser

def onWebBrowserConnection(self, browserClient):
    # If there is not a simulation that it is not being used by any client
    # from a browser, the connection is closed.
    if len(self.simulationClientAvailable) == 0:
        browserClient.transport.loseConnection()
        return

    # If there is one available, we Save the relationship between browser
    # client and simulation
    # to be able to forward messages between both ends of the communication.

    # Save browser - simulation relationship
    key = browserClient.transport.getPeer().host + "\n" + str(browserClient.
       transport.getPeer().port)
    value = self.simulationClientAvailable[0]
    self.connections[key] = value

    # Save simulation - browser relationship
    key = value.transport.getPeer().host + "\n" + str(value.transport.
       getPeer().port)
    value = browserClient
    self.connections[key] = value

    # Now the simulation is in use and is not available for other clients
    del(self.simulationClientAvailable[0])

def onWebBrowserMsg(self, browserClient, msg):
    # We get the Ns3 connection associated with this browser client
    key = browserClient.transport.getPeer().host + "\n" + str(browserClient.
       transport.getPeer().port)
```
A.4. Management of the events from Simulation Server

```python
# Check if the connection is still alive
if key not in self.connections:
    browserClient.sendMessage("Connection with the simulation not available")
    browserClient.transport.loseConnection()

# If it is alive, the message is forwarded
simulation = self.connections[key]
simulation.transport.write(msg)

def onWebBrowserDisconnection(self, browserClient):
    # Get the browser client key
    browserClientKey = browserClient.transport.getPeer().host + ":" + str(
        browserClient.transport.getPeer().port)

    # If it has been closed before by the simulation when it was shut down
    # we need to do nothing
    if browserClientKey not in self.connections:
        return

    # Get the simulation client key
    simulationClient = self.connections[browserClientKey]
simulationClientKey = simulationClient.transport.getPeer().host + ":" +
    str(simulationClient.transport.getPeer().port)

    # Shutdown connection with the simulation and remove from our connections
    simulationClient.transport.loseConnection()

    del(self.connections[browserClientKey])
    del(self.connections[simulationClientKey])
```

Listing A.4: Callbacks exposed by Socket ns-3 Class

**A.4 Management of the events from Simulation Server**
# We get the Ns3 connection associated with this browser client
key = simulationClient.transport.getPeer().host + ";" + str(
    simulationClient.transport.getPeer().port)

# Check if the connection is still alive
if key not in self.connections:
    simulationClient.transport.write("Connection with the browser not available")
    simulationClient.transport.loseConnection()
    return

# If it is alive, the message is forwarded
client = self.connections[key]
client.sendMessage(msg)

def onNs3Disconnection(self, simulationClient):
    # Remove if the simulation was not used and simulation time has finished
    if simulationClient in self.simulationClientAvailable:
        self.simulationClientAvailable.remove(simulationClient)
        return

    # Get the simulation client key
    simulationClientKey = simulationClient.transport.getPeer().host + ";" + str(simulationClient.transport.getPeer().port)

    # It has been closed before by the client when he disconnected
    if simulationClientKey not in self.connections:
        return

    # Get the simulation client key
    browserClient = self.connections[simulationClientKey]
    browserClientKey = browserClient.transport.getPeer().host + ";" + str(
        browserClient.transport.getPeer().port)

    # Shutdown connection with the client and remove from our connections
    browserClient.transport.loseConnection()
    del(self.connections[browserClientKey])
    del(self.connections[simulationClientKey])

Listing A.5: Callbacks exposed by Socket ns-3 Class


APPENDIX B

Implementation code details of the I/O Proxy

B.1 Configuration of the I/O Proxy

```cpp
#include "ns3/icmpv4.h"
#include "ns3/assert.h"
#include "ns3/log.h"
#include "ns3/ipv4-address.h"
#include "ns3/socket.h"
#include "ns3/integer.h"
#include "ns3/boolean.h"
#include "ns3/inet-socket-address.h"
#include "ns3/packet.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/config.h"
#include "ns3/tos-device.h"
#include "ns3/names.h"
#include "ns3/string.h"
#include "ns3/object.h"
#include "io-proxy-server-application.h"
#include <stdio.h>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("IOPProxyServer");
NS_OBJECT_ENSURE_REGISTERED (IOPProxyServer);

IOPProxyServer::IOPProxyServer ()
{
```
m_socket = 0;
}

TypeId IOProxyServer::GetTypeId (void)
{
    static TypeId tid = TypeId ("ns3::IoProxyServer")
        .SetParent<Application>()
        .AddConstructor<IOProxyServer>()
        .AddAttribute("RemotePortNumber",
            "Remote port listening for connections",
            IntegerValue(9999),
            MakeIntegerAccessor(&IOProxyServer::m_remotePortNumber),
            MakeIntegerChecker<int64_t>())
        .AddAttribute("RemoteIp",
            "Remote IP listening for connections",
           StringValue("127.0.0.1"),
            MakeStringAccessor(&IOProxyServer::m_remoteIp),
            MakeStringChecker())
        .AddAttribute("LocalPortNumber",
            "Local port for incoming connections",
            IntegerValue(3333),
            MakeIntegerAccessor(&IOProxyServer::m_localPortNumber),
            MakeIntegerChecker<int64_t>())
        .AddAttribute("LocalIp",
            "Local IP for incoming connections",
           StringValue("127.0.0.1"),
            MakeStringAccessor(&IOProxyServer::m_localIp),
            MakeStringChecker())
        .AddAttribute("NodePairsNumber",
            "Number of node pairs used on the simulation (e.g 1 = one sensors & one actuator)",
            IntegerValue(2),
            MakeIntegerAccessor(&IOProxyServer::m_nodePairsNumber),
            MakeIntegerChecker<int64_t>())
    return tid;
}

void IOProxyServer::StartApplication (void)
{
    NS_LOG_FUNCTION (this);
    m_socket = Socket::CreateSocket(GetNode(), TypeId::LookupByName("ns3::TcpSocketFactory"));
    NS_ASSERT_MSG (m_socket != 0, "An error has happened when trying to create the socket");
    InetSocketAddress src = InetSocketAddress(Ipv4Address::GetAny(), m_localPortNumber);
    InetSocketAddress dest = InetSocketAddress(Ipv4Address(m_remoteIp.c_str()), m_remotePortNumber);
int status;
status = m_socket->Bind(src);
NS_ASSERT_MSG(status != -1, "An error has happened when trying to bind to local end point");

status = m_socket->Connect(dest);
NS_ASSERT_MSG(status != -1, "An error has happened when trying to connect to remote end point");

// Configures the callbacks for the different events related with the connection

// m_socket->SetConnectCallback
m_socket->SetAcceptCallback(
    MakeNullCallback<bool, Ptr<Socket>, const Address&>(),
    MakeCallback(&IOProxyServer::HandleAccept, this));

m_socket->SetRecvCallback(
    MakeCallback(&IOProxyServer::HandleRead, this));

m_socket->SetDataSentCallback(
    MakeCallback(&IOProxyServer::HandleSend, this));

// m_socket->SetSendCallback
m_socket->SetCloseCallbacks(
    MakeCallback(&IOProxyServer::HandlePeerClose, this),
    MakeCallback(&IOProxyServer::HandlePeerError, this));

// If we need to configure a reception only socket or a sending only socket
// we need to call one of the following methods:
// m_socket->ShutdownSend();
// m_socket->ShutdownRecv();

void IOProxyServer::StopApplication(void)
{
    NS_LOG_FUNCTION(this);
    m_socket->Close();
}

void IOProxyServer::HandleAccept(Ptr<Socket> s, const Address& from)
{
    NS_LOG_FUNCTION(this << s << from);
    s->SetRecvCallback(MakeCallback(&IOProxyServer::HandleRead, this));
}
void IOProxyServer::HandleRead (Ptr<Socket> socket)
{
    [...] \ Appendix B.2
}

void IOProxyServer::HandlePeerClose (Ptr<Socket> socket)
{
    NS_LOG_FUNCTION (this << socket);
}

void IOProxyServer::HandlePeerError (Ptr<Socket> socket)
{
    NS_LOG_FUNCTION (this << socket);
}

void IOProxyServer::HandleSend (Ptr<Socket> socket, uint32_t dataSent)
{
    NS_LOG_FUNCTION (this << socket);
}

IOProxyServer::~IOProxyServer ()
{
}

void IOProxyServer::DoDispose (void)
{
    NS_LOG_FUNCTION (this);
    m_socket = 0;
    Application::DoDispose ();
}

} // namespace ns3

Listing B.1: IOProxyServer

B.2 Receiving and parsing information from the I/O Server

void IOProxyServer::HandleRead (Ptr<Socket> socket)
{
    NS_LOG_FUNCTION (this << socket);
    Ptr<Packet> packet;
    while ((packet = socket->RecvFrom (from)))
    {
        [...] \ Appendix B.3
    }
if (packet->GetSize() == 0)
{
    // EOF
    break;
}

if (InetSocketAddress::IsMatchingType(from))
{
    NS_LOG_INFO("\n[" << Simulator::Now().GetSeconds() << "s] - Rx: " << packet->GetSize() << " bytes from 
" << InetSocketAddress::ConvertFrom(from).GetIpv4() << ":" << InetSocketAddress::ConvertFrom(from).GetPort();
    std::cout << "\n\n";
    // Extract the string from the packet
    unsigned char *value = (unsigned char *) malloc(sizeof(uint8_t) * packet->GetSize());
    packet->CopyData(value, packet->GetSize());
    std::string data((char *)value);
    // Select if its sensor or actuator checking the received string
    if (data.find("Sensor") != std::string::npos)
    {
        // Received pattern matches: *Sensor-sensorId:sensorValue
        int sensorId = atoi(data.substr(data.find("-")) + 1, data).c_str());
        std::cout << "Sensor Id: " << sensorId;

        int sensorValue = atoi(data.substr(data.find(":")+1).c_str())
        ;
        std::cout << " - Value: " << sensorValue << "\n\n";

        std::string name = "/Names/TemperatureSensor";
        // Identify the number of the sensors
        char numstr[21]; // enough to hold all numbers up to 64-bits
        sprintf(numstr, "%d", sensorId);
        std::string result = name + numstr;
        Ptr<TosDevice> sens = Names::Find<TosDevice>(result);

        // Send the measurement to the correct sensor
        sens->SendRawData(sizeof(sensorValue), &sensorValue);
        // Creates an echo of the message received
        socket->SendTo(packet, 0, from);
    }
    else if (data.find("actuator") != std::string::npos)
    {
        // Received pattern matches: *Actuator-actuatorId:
        actuatorValue
    }
}
int actuatorId = atoi(data.substr(data.find("-") + 1, data.find(";")).c_str());
std::cout << "Actuator Id: " << actuatorId;

int actuatorValue = atoi(data.substr(data.find(";")+1).c_str());
std::cout << "Value: " << actuatorValue << " \n\n";

std::string name = "/Names/Actuator";
// Identify the number of the sensors
char numstr[21]; // enough to hold all numbers up to 64-bits
sprintf(numstr, "%d", actuatorId);
std::string result = name + numstr;
Ptr<TosDevice> act = Names::Find<TosDevice>(result);

// Send the measurement
act->SendRawData(sizeof(actuatorValue), &actuatorValue);
// Creates and echo of the message
socket->SendTo(packet, 0, from);
}
}

Listing B.2: IOProxyServer::HandleRead

B.3 Preparing and sending information to the I/O Server

void IOProxyServer::SendData(std::string str)
{
    char *cstr = new char [str.length()+1];
    strcpy(cstr, str.c_str());

    Ptr<Packet> packet = Create<Packet>(((uint8_t*)cstr, str.length()+1);
    m_socket->Send(packet, 0);
    delete [] cstr;
}

Listing B.3: IOProxyServer::SendData
NS-3 Simulation Script

C.1 Script

```cpp
#include "ns3/abort.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/network-module.h"
#include "ns3/emu-module.h"
#include "ns3/applications-module.h"
#include "ns3/ipv4-static-routing-helper.h"
#include "ns3/ipv4-list-routing-helper.h"
#include "ns3/io-proxy-server-application.h"
#include "ns3/hv-base-station-application.h"

#include "ns3/symphony-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("SkynetNs3ProxyScript");

int main (int argc, char *argv [])
{
    NS_LOG_INFO ("Loading Skynet NS3 Proxy Script");
    LogComponentEnable("SkynetNs3ProxyScript", LOG_INFO);
    LogComponentEnable("IOProxyServer", LOG_INFO);
    LogComponentEnable("SymphonyApplication", LOG_INFO);

    // IO Proxy default network parameters
    std::string deviceName("eth0");
```
std::string localIp("172.16.107.142");
std::string localMask("255.255.255.0");
std::string localGateway("172.16.107.2");
int localPortNumber(3333);

// Backend Server default network parameters
std::string remoteIp("172.16.107.1");
int remotePortNumber(9999);

// Default Symphony parameters
std::string nodeModel("/home/onir/dev/skynet/ns-3.14/build/symphony.xml");
std::string nodeImage("/home/onir/dev/skynet/ns-3.14/build/libSkynetTos.so");
int nodePairsNumber(2);

// Default simulation time
std::string simulationTime("60");

// Allow modification of default network parameters
CommandLine cmd;
cmd.AddValue("deviceName", "Device Name (e.g eth0)", deviceName);

cmd.AddValue("localIp", "Local IP address (e.g 192.168.1.100) – It should match with the IP range inside the network to work", localIp);

cmd.AddValue("localMask", "Local Network Mask (e.g. 255.255.255.0)", localMask);

cmd.AddValue("localGateway", "Local Gateway (e.g. 192.168.1.1)", localGateway);

cmd.AddValue("localPortNumer", "Local Port Number (e.g 3333)", localPortNumber);

cmd.AddValue("remoteIp", "Remote IP address (e.g 172.16.107.1)", remoteIp);

cmd.AddValue("remotePortNumber", "Listening port for the server (e.g 9999)", remotePortNumber);

cmd.AddValue("simulationTime", "Simulation time (s) (e.g. 60)", simulationTime);

cmd.AddValue("nodeModel", "Full path to XML description of the node (s) (e.g. /home/user/symphony.xml)", simulationTime);

cmd.AddValue("nodeImage", "Full path to .so image of TinyOS code (e.g /home/user/libTos.so)", nodeImage);

cmd.AddValue("nodePairsNumber", "Number of sensor&actuator pairs to configure the simulation. (e.g. 1 = one sensors & one actuator") , nodePairsNumber);

cmd.Parse(argc, argv);
// Configure the simulation as a real time and checksum enabled
// simulation to allow the communication with nodes outside.
GlobalValue::Bind("SimulatorImplementationType", StringValue("ns3::
    RealtimeSimulatorImpl"));
GlobalValue::Bind("ChecksumEnabled", BooleanValue(true));

// Configuration of the Skynet proxy node
NS_LOG_INFO("- Create IO Server Node");

Ptr<Node> node = CreateObject<Node>();

// Create an emu device, allocate a MAC address and point the device to
// Linux device name. The device needs a transmit queueing discipline
// so
// create a droptail queue and give it to the device. Finally, "install"
// the device into the node.
NS_LOG_INFO("- Create Emulation Device");

Ptr<EmuNetDevice> device = CreateObject<EmuNetDevice>();
    device->SetAttribute("Address", Mac48AddressValue(Mac48Address::
        Allocate()));
    device->SetAttribute("DeviceName", StringValue(deviceName));

Ptr<Queue> queue = CreateObject<DropTailQueue>();
device->SetQueue(queue);

node->AddDevice(device);

// Add a default internet stack to the node.
NS_LOG_INFO("- Add Internet Stack");

InternetStackHelper internetStackHelper;
internetStackHelper.Install(node);

NS_LOG_INFO("- Add IPv4 Interface");

Ipv4Address ip = localIp.c_str();
Ipv4Mask mask = localMask.c_str();

Ptr<Ipv4> ipv4 = node->GetObject<Ipv4>();
    uint32_t interface = ipv4->AddInterface(device);
    Ipv4InterfaceAddress address = Ipv4InterfaceAddress(ip, mask);
    ipv4->AddAddress(interface, address);
    ipv4->SetMetric(interface, 1);
    ipv4->SetUp(interface);

// Configure the default path to route the packets
NS_LOG_INFO("- Add Default Route");

Ipv4Address gateway (localGateway.c_str());

Ipv4StaticRoutingHelper ipv4RoutingHelper;
Ptr<Ipv4StaticRouting> staticRouting = ipv4RoutingHelper;
GetStaticRouting (ipv4);
staticRouting->setDefaultRoute (gateway, interface);

// Configure the application running on the node
NS_LOG_INFO("- Adding IO Proxy Server");

Ptr<IOPProxyServer> ioApp = CreateObject<IOPProxyServer>();
ioApp->setStartTime (Seconds (1.0));
ioApp->setStopTime (Seconds (atoi(simulationTime.c_str()) - 1));
ioApp->setAttribute ("RemotePortNumber", IntegerValue (remotePortNumber));
ioApp->setAttribute ("RemoteIp", StringValue (remoteIp));
ioApp->setAttribute ("LocalPortNumber", IntegerValue (localPortNumber));
ioApp->setAttribute ("LocalIp", StringValue (localIp));
ioApp->setAttribute ("NodePairsNumber", IntegerValue (nodePairsNumber));
node->addApplication (ioApp);

Names::Add ("IOServer", ioApp);

// Configure the TinyOS node
NS_LOG_INFO("- Creating TinyOS node");

TosNodeContainer tosNode;
tosNode.Create (nodePairsNumber * 2, nodeModel.c_str());

// Configure the Symphony Application
NS_LOG_INFO("- Adding bridge between TinyOS and NS3");

Ptr<SymphonyApplication> symphonyApp = CreateObject<SymphonyApplication>();
symphonyApp->setNode (tosNode.Get (0));
symphonyApp->setStartTime (Seconds (0.0));
symphonyApp->setStopTime (Seconds (20.0));
symphonyApp->startApplication ();

// Configuring the sensors available in the node
NS_LOG_INFO("- Adding sensors");

TosHelper sens;
sens.SetNodeModel (nodeModel);
SymphonySensorContainer sc = sens.InstallSensors (1, tosNode, "/home/onir/dev/symphony/ns-3.14/bin_pkt/" );

// Set properties of the nodes
for (int i = 0; i < nodePairsNumber; i++)
{
    char numstr[21]; // enough to hold all numbers up to 64-bits

    // Set sensor properties
tosNode.Get(i)->SetAttribute("TosId", UintegerValue(i));
tosNode.Get(i)->AddApplication(symphonyApp);

    // Set sensor identification string
    sprintf(numstr, "%d", i);
    std::string name = "TemperatureSensor";
    std::string result = name + numstr;
    Names::Add(result, sc.Get(i));

    // Set actuator properties
tosNode.Get(i + nodePairsNumber)->SetAttribute("TosId", UintegerValue(i + nodePairsNumber));
tosNode.Get(i + nodePairsNumber)->AddApplication(symphonyApp);

    // Set actuator identification string
    sprintf(numstr, "%d", i);
    name = "Actuator";
    result = name + numstr;
    Names::Add(result, sc.Get(i + nodePairsNumber));
}

NS_LOG_INFO("- Adding HV Base Station");
Ptr<HvBaseStation> bsApp = CreateObject<HvBaseStation>(tosNode);
bsApp->SetStartTime(Seconds(1.0));
bsApp->SetStopTime(Seconds(atoi(simulationTime.c_str()) - 1));
node->AddApplication(bsApp);
Names::Add("BaseStation", bsApp);

// Start the simulation
NS_LOG_INFO("- Emulation starting");
NS_LOG_INFO("=");
Simulator::Stop(Seconds(atoi(simulationTime.c_str())));
Simulator::Run();
Simulator::Destroy();
NS_LOG_INFO("=");
NS_LOG_INFO("- Emulation finished");
}
D.1 Learning Algorithm

```cpp
void HvBaseStation::ReceiveHyperVector(uint16_t size, void *buffer) {
    NS_LOG_FUNCTION(this);

    // Extract the message from the sensor
    NodePacket* packet = (NodePacket*) buffer;
    // Store the vector locally
    memcpy(input_vector, packet->vector, sizeof(int[a]));
    std::cout << "\tInput vector sum is: " << Sum(input_vector, a) << " and value is: \n";

    // Check if the vector is on binary buffer
    Xor(check_vec1, input_vector, buff_bin, a);
    check_sum = Sum(check_vec1, a);
    std::cout << "\tCheck sum of input_vector xor buff_bin is " <<
               check_sum << "\n";

    if (check_sum < 0.20) { // Discard input vector, it is already in the buffer
        for (int i=0; i<a; i++) input_vector[i] = 0;
        return;
    } else { // Process new vector
        if (item_len != 0) { // if item memory is not empty
            std::cout << "\tItem memory has elements... \n";
            for (int i=0; i<item_len; i++) {
```
Xor(check_vec1, input_vector, item_mem[i].hpSen, a); //
check for vector presence in item memory
item_check[i] = Sum(check_vec1, a);
std::cout << "\tCheck entry " << i << " of item memory => Sum " << item_check[i] << "\n";

if (item_check[i] < 0.2) {
    item_pres[i] = 1; // if yes put 1 in presence variable
    vector
    std::cout << "\tPresence on position " << i << "\n";
} else {
    item_pres[i] = 0;
}

// Sum of presence vector
pres_sum = Sum(item_pres, item_len);
std::cout << "\tSum of presence vector is " << pres_sum << " (Item memory len: " << item_len << " )\n";

// if presence sum non zero check for prev_shift and sensor
// cur_shift coincidence
if (pres_sum != 0) {
    for (int i = 0; i < item_len; i++) {
        std::cout << "\tChecking entity " << i << "\n";

        if (item_pres[i] == 1) {
            std::cout << "\tFlag set to 1 on item presence \n";

            if (item_mem[i].prev_shift == sen_state[item_mem[i].prev_role].cur_shift) {
                for (int j = 0; j < a; j++) {
                    output_vec[j] = item_mem[i].hp_act[j]; // send corresponding hypervector to actuator
                }

                Shift(subs_vec, init_mem[item_mem[i].prev_role].role_hv, a, sen_state[item_mem[i].prev_role].cur_shift);
                Or(subs_vec, subs_vec, init_mem[item_mem[i].prev_role].role_hv, a);
                Xor(subs_vec, subs_vec, mP.role_hv, a);
            }
        }
    }

    for (int h = 0; h < a; h++) {

D.1. Learning Algorithm

```cpp
buff_int[h] = buff_int[h] + input_vector[h]; //
// add new vector to the integer buffer
buff_int[h] = buff_int[h] - subs_vec[h]; //
// substitute old vector from the integer buffer
}
Buff_int_to_bin(buff_int, buff_bin, a); // update
// binary buffer
sen_state[item_mem[i].prev_role].cur_shift =
item_mem[i].cur_shift; // change sensor state
// in the table
std::cout << "\t\t\t\t\t\t\t\n"; Return actuator
status << "n";
SendOutputVector(output_vec);
return;
```

```cpp
std::cout << "Looking for input vector... \n";
// release input vector from randomization vector (init_mem[0].
// role_hv) by xoring
Xor(check_vec1, input_vector, mP.role_hv, a);
// check in init memory till we find correspond role vector
std::cout << "\t\t\tInit memory... \n";
while (sum_adres > 0.3) {
  Xor(check_vec2, check_vec1, init_mem[count_adr].role_hv, a);
  sum_adres = Sum(check_vec2, a);
  input_id = init_mem[count_adr].id;
  count_adr++;
}
std::cout << "\t\t\t[i, input_id, sum_adres] = " \t" \t<< count_adr <<
  " " << input_id << " " << sum_adres << "\n";
// check for shift of correspond role vector
std::cout << "\t\t\tShifting vector... \n";
while (sum_shift > 0.3) {
  Shift(shift_vec1, init_mem[input_id].role_hv, a, k_shift);
  Xor(check_vec2, check_vec1, shift_vec1, a);
  sum_shift = Sum(check_vec2, a);
  input_shift = k_shift;
  count_shift++;
  k_shift++;
}
std::cout << "\t\t\t[i, input_shift, sum_shift] = " \t" \t<< count_shift <<
  " " << input_shift << " " << sum_shift << "\n";
```
102
104
106
108
110
112
114
116
118
120
122
124
126
128
130
132
134
136
138

// If it is sensor
if (input_id < node_pairs_number/2) {  // sensors id less than number of nodes / 2
    // find sensor in sensors state table
    std::cout << "\t\tMessage from sensor " << input_id << " detected \n";

    for (int j=0; j<n;j++) {
        if (sens_state[j].sen_id == input_id) {
            cur_state_num = j; // fix position of sensor in the table
        }
    }
    std::cout << "\t\t\tSensor position in table " << cur_state_num << "\n";

    // Form previous measurement vector for current sensor
    Shift(subs_vec, init_mem[cur_state_num].role_hv, a, sens_state[cur_state_num].cur_shift);
    Or(subs_vec, subs_vec, init_mem[cur_state_num].role_hv, a);
    Xor(subs_vec, subs_vec, mP.role_hv, a);

    for (int i=0; i<a;i++) {
        buff_int[i] = buff_int[i] + input_vector[i]; // add new vector to the integer buffer
        if (sens_state[cur_state_num].init_flag == 1){
            buff_int[i] = buff_int[i] - subs_vec[i]; // substitute old vector from the integer buffer
        } else {
            if (i == a-1) { sens_state[cur_state_num].init_flag = 1;}
        }
    }
    Buff_int_to_bin(buff_int, buff_bin, a); // update binary buffer

    // Open pending record for item memory
    item_pending[0].cur_role = input_id;
    item_pending[0].cur_shift = input_shift;
    item_pending[0].prev_role = input_id;
    item_pending[0].prev_shift = sens_state[cur_state_num].cur_shift;
    sens_state[cur_state_num].cur_shift = input_shift; // change sensor state in the table
    for (int i=0; i<a;i++) {
        item_pending[0].hp_sen[i] = input_vector[i];
    }
std::cout << "\tInteger and binary buffer updated. Item pending configured \n";
std::cout << "\tCurrent Role: " << item_pending[0].cur_role << "\n";
std::cout << "\tCurrent Shift: " << item_pending[0].cur_shift << "\n";
std::cout << "\tPrevious Role: " << item_pending[0].prev_role << "\n";
std::cout << "\tPrevious Shift: " << item_pending[0].prev_shift << "\n";

input_shift = 0; input_id = 0; count_adr = 0; count_shift = 0;
sum_adres = 1; sum_shift = 1; k_shift = 1;

// if vector from actuator
else if (input_id >= node_pairs_number/2) { 
  std::cout << "\tMessage from actuator detected \n";

  if (item_pending[0].cur_role != -1) { // if pending record exists
    for (int i = 0; i < a; i++) {
      item_pending[0].hp_act[i] = input_vector[i];
    }
    item_mem[item_len] = item_pending[0]; // add record to the item memory
    item_pending[0].cur_role = -1;
    item_pending[0].cur_shift = 0;
    item_pending[0].prev_role = 0;
    item_pending[0].prev_shift = 0;
    item_len++; // increase item memory length
    std::cout << "\tNew entry included in the item memory \n";
  }

  input_shift = 0; input_id = 0; count_adr = 0; count_shift = 0;
  sum_adres = 1; sum_shift = 1; k_shift = 1;
}

Listing D.1: Learning Algorithm in Base Station
REFERENCES


