Availability Management of Composable Services for Wireless Sensor Networks
Supported by the Programme Alßan, the European Union Programme of High Level Scholarships for Latin America, scholarship No. E05M056796CO

Verfügbarkeitsmanagement zusammensetzbarer Dienste für drahtlose Sensornetze

Masterarbeit
Darmstadt - Germany
2008

Prüfer: Prof. Dr. Dr. h. c. mult. Manfred Glesner
Betreuer: Dr. -Ing. Leandro Soares Indrusiak
Bearbeiter: Diego Alonso Ahogado Alvarez
Hiermit erkläre ich, die vorliegende Diplomarbeit selbständig und ohne Verwendung unzulässiger Hilfsmittel angefertigt zu haben. Alle benutzten Quellen sind im Literaturverzeichnis aufgeführt.

Darmstadt, 16. Dezember 2008

Diego Ahogado

Diego Alonso Ahogado Alvarez
Acknowledgments

I would like to thank Dr.-Ing. Leandro Soares Indrusiak for his kind assessment and support throughout the orientation and execution of this research, as well as through all the different stages of my master studies in the Darmstadt University of Technology, from the early beginning up to the last day. For being there for application issues, acceptance delay, and scholarship requirements even before knowing me. For finding always solutions to all my concerns and helping to develop myself at personal and academic levels, whenever there were important decisions to be made and actions to be taken.

I would like to thank Prof. Dr. Dr. h.c. mult. Manfred Glesner for his personal support and constant care about my situation as foreign student. For introducing me to the city and activities organized by him to maintain people joined, as well as create new contacts between his appreciated people. Also for supporting me to receive a scholarship that was a key to the success of this project.
ABSTRACT

Resource management is a major concern for system design of battery-powered embedded devices. As some of these devices are deployed in places where they have to work for long time periods without battery replacements, system availability can be maximized by increasing the time that the devices are able to fulfil their objectives (in this research we handle service provision). Given that every service offered by a node implies battery consumption, a permanent availability of all possible services will decrease the average battery life in the system at the maximum rate. Managing the catalogue of offered services at a given point, based on the current available resources and priority of each service, will help to keep available the most important services as long as possible, while extending battery life by avoiding resource consumption in less important or very expensive services. Using an approach of service composition where complex services inherit their functionality from simpler services, it is possible to offer high functionality summarized in a complex service, instead of offering every existent service type. The usefulness of composed services in service availability management is evaluated in this research, additionally introducing a communication protocol for service provision and a codification scheme for hierarchical trees, as tools for service availability management.
# TABLE OF CONTENTS

## INTRODUCTION

### 1. THEORETICAL FRAMEWORK

1.1. State of the Art: Mobile Ad Hoc Networks
1.2. State of the Art: Wireless Sensor Networks (WSN)
    1.2.1. Sensor devices example
    1.2.2. TinyOS
1.3. State of the Art: Wireless Networks simulation
1.4. Ptolemy II
    1.4.1. Simulation scenarios in Ptolemy II/ Visual Sense
    1.4.2. Actor inner-processing inside a simulation
1.5. Tiny Discovery Protocol

## 2. HIERARCHICAL TREE CODIFICATION

2.1. Binary Path Representation of N-ary Hierarchy Trees
2.2. Node representation through Significance Bits and Code Inheritance
2.3. Codification schemes unification
2.4. Relationship evaluation
    2.4.1. Sibling relationship evaluation
    2.4.2. Parentship evaluation

## 3. SERVICE AVAILABILITY MANAGEMENT

3.1. The Availability Problem
3.2. Hierarchical Models for Composable Services
    3.2.1. HSIP: Hierarchical Services of Interacting Processes
    3.2.2. Type inheritance
3.3. Abstract Reachability Graph
3.4. Service Management
    3.4.1. Resource Metrics
    3.4.2. Service Metrics
    3.4.3. Deactivation policies
3.5. Deactivation Procedure
3.6. Conclusion
INTRODUCTION

Service availability management is a helpful mechanism for regulation of resource consumption in nodes providing services to a network. Joint with a good service discovery and advertisement mechanism, it can help saving resources as battery or available memory to improve system performance and life time. In this research, we analyse the usefulness of using an approach of service composition for the service description, using it to manage the service availability by evaluating the affordability of every service based on its priority and the current resource availability, and producing sets of affordable services summarized in a catalogue ready to be sent to client nodes.

The first section describes the Wireless Sensor Networks as the research field for which this research is intended, as well as the simulation tools and protocol developments already existent and used as framework to analyse and simulate the usefulness of service composition in service availability management.

The second section introduces a proposal for binary codification of hierarchical trees, as they are used to describe service composition. This codification scheme allows for the rapid recognition of parentship between two nodes, described also in this section and utilized later in the availability management to select children services out of a hierarchical tree.

In the third section the concept of availability is analysed, and a cost-driven criteria is recommended as service selection mechanism. Several metrics are proposed for designers to create cost functions and evaluate services, and the service selection mechanism is proposed as a deactivation process, where the managed node have a service composition tree describing the services that he can provide and their inheritance relations, reduces the tree by removing the services which are not affordable given the current resource availability, and then reduces it again by selecting only the most complex services which summarize the functionalities of their parent service types. The result of this “deactivation procedure” will be a catalogue of offered services to be sent to the clients. In this approach, services will be treated as types, which can inherit their functionality to other services, for them to extend and specialize it.
Services are also able to be composed by the functionality of different base types, by means of allowing multiple inheritance.

In the fourth and fifth section, the simulation model used to evaluate the performance of service availability management is introduced, including the improvements made to the protocol used for service discovery and provision. The communication model utilized is explained, together with the packet formats used to allow understanding between the server and client parts, and the constant parameters for the simulation scenario are established. Summaries of several simulations are presented in tables and charts, to analyse the impact of introducing availability management at different stages of the system lifetime, seen as the effect of starting the deactivation of offered services based on priorities at different levels of remaining battery load.

Finally, conclusions are stated based on the simulation results and the experiences through the different stages of analysis, design and implementation of this research, focused on the impact of the discovery mechanism in the benefits of availability management and the behaviour of the simulated network in response to the deactivation of services at different stages.
1. THEORETICAL FRAMEWORK

1.1. State of the Art: Mobile Ad Hoc Networks

Ad Hoc concept in computer networks describes the behaviour of a net having independent nodes as components, making no use of the hierarchy present in classical server-client topologies, avoiding also the dependence on centralization points, as cellular networks depend on static base antenna stations (BS) to communicate with the mobile stations (MS) surrounding them and to interconnect the different cells of a network (See Fig. 1.1) [2].

Due to the independent structure of the Ad Hoc networks it is necessary to take in to account the mobility of the nodes in the environment, where they are moving freely using different mobility patterns and are changing the physical structure of the network dynamically [1]. In this scenario, it is not possible to expect for a permanent communication between two nodes, since they can lose reach ability from its close partners as they move, due to the limited transmission power embedded to the wireless technologies. This problem is solved by using other nodes as routers for the information flowing between two distant nodes, taking into account that as the nodes move, they will have the need for dynamical reconfigurations of the routing network.

Fig. 1.1. Adhoc network (left) and cellular network (right) structures.
1.2. State of the Art: Wireless Sensor Networks (WSN)

Grouping sensor devices, the Wireless Sensor Networks are in general designed for cooperation between battery dependent devices equipped with electronic sensors, to accomplish different control tasks without making any use of physical infrastructures.

1.2.1. Sensor devices example

Wireless sensors have been utilized in different industries, for different kind of uses like temperature control in transport of packed up merchandise and pharmacies, temperature measurement in rooms, nocturnal security (vibration/movement sensors), and ambient intelligence, field in which they are quite important for design of intelligent buildings [18].

One sort of sensor devices was developed in the Berkeley University of California, now under management of Sentilla Inc., ideal for research in sensor-based applications. Tmote Sky sensors, designed and built under the old name of Sentilla: moteiv [9], bring temperature, illumination and humidity measurement functionality, through built-in sensors for this purpose. For communication purposes, they bring communication interfaces working with IEEE 802.15.4\(^1\) [19][20] protocol and USB (Universal Serial Bus) connectors (see Fig. 1.2).

![General Schema and physical appearance of Tmote Sky sensors](image)

1.  IEEE 802.15.4 provides a standard for physical and data link layers, to deliver communication services at low transfer rates in wireless Personal Area Networks.
2.  Tomado de la documentación de Tmote Sky, de moteiv [9].
could be substituted with USB power supply. Tmote Sky provides also capability to turn off and on its network interface in 6 microseconds, with the purpose of saving battery energy without critically affecting the response time from such interface.

This sensors are an example of devices used for control applications where they are installed under dependence on batteries as power supply, requiring good application designs to manage correctly the limited memory resources and low energy consumption by means of minimizing the wireless interface activity (wireless data transfer is one of the most expensive procedures in terms of energy consumption).

1.2.2. TinyOS

TinyOS is an open source operative system, designed for wireless networks of devices with embedded sensors. This system has an architecture based in components allowing for rapid innovation and implementation, minimizing as well the programmed code size, as it is required by memory restrictions inherent to sensor networks [12]. Devices such as Tmote Sky are operated under TinyOS.

The applications developed for TinyOS are written making use of the nesC programming language, focused on structured applications based on components, and oriented mainly to embedded systems [11]. NesC utilizes a C (programming language) like syntax, but supports the concurrence model from TinyOS, as well as mechanisms to structure, name and interlace software components, constructing robust network systems.

1.3. State of the Art: Wireless Networks simulation

Different tools have been developed to simulate communication networks, some of them capable to simulate mobile wireless networks. Between the most recognized simulators for wireless networks, NS-2 [3], TOSSIM [4] and Visual Sense [5] provide better specific functionalities.
Visual Sense, part of Ptolemy II project [6], is recognized by its clear interface and its capability to simulate concurrence (multiple independent processes running in the same time interval), highly useful to simulate several independent wireless devices with processors, existing in the same scenario and working concurrently. Visual Sense allows for wireless channel simulation taking variables into account such as delay and signal attenuation but, similar to NS-2, it does not provide tools to clearly represent the battery consumption, important factor in simulation of battery-dependent devices such as cell phones and some sensor devices.

To allow for correct presentation of results, several alternatives have been programmed to visualize NS-2 simulation, not being optimal or even unable to visualize energy parameters. In the case of TOSSIM, designed to simulate devices based in TinyOS, the nodes energy is not simulated in its standard version, but has been included in an extension called PowerTOSSIM, able to represent energy consumption in the nodes [7].

In Visual Sense case, being a limited set of the major project Ptolemy II, it disposes of a clear interface to model and simulate scenarios, as well as elements to visualize simulation results (i.e. Cartesian charts). Although simulation of energy consumption in the nodes is not optimally implemented, the elements (actors) in Ptolemy II can be programmed using the Java programming language to achieve an extended functionality and provide a wider control over actor’s behaviour. Additionally and similar to Visual Sense, there exists another subset of Ptolemy II under the name of Viptos, looking for focused simulation of TinyOS-based sensors [8].

1.4. Ptolemy II

The software platform Ptolemy II, developed in the Berkeley University of California, provides tools to simulate devices and electronic components running concurrently [6]. On this platform, several tool sets are grouped in applications to make easier simulation, providing graphical interfaces with access to actors useful for a purpose (microprocessor simulation, wireless networks simulation, etc.). Among those interfaces, we find Vergil, Visual Sense and Viptos, focused on devices and microelectronic components (Vergil),
wireless media and devices (Visual Sense) and wireless sensor networks (Viptos). Visual Sense y Viptos, in the context of protocol development for wireless networks, cover essentially the same set of actors and functions plus some actors in viptos to generate nesC code based on the simulated functionality. This is particularly useful for simulation and design of applications for TinyOS.

1.4.1. Simulation scenarios in Ptolemy II/ Visual Sense

A simulation made in the Ptolemy II software is based on a scenario composed by actors interacting between them. An actor in Ptolemy II could be defined as a single component in a scenario, where each element in a simulation is represented by an actor, being a minimal basic unit, or being a container for other actors and behaving as a black box. Examples of actors are the transmission media (wired and wireless), nodes (composite actors containing other actors), output interfaces for text and graphics (used to allow the user to visualize the data flux or the output data in a simulation). Among the simples actors we can find adders, file readers and logical comparators, useful to assembly a composite actor (which commonly represents a node or device) building a data flux diagram inside of it based on simple actors, where such diagram represents or “simulates” the internal functionality of an actor. The idea behind composite actors is to group a set of actors to perform a task, and let them operate invisibly from the world outside this composite actor. Those actors are capable to communicate between them by sending data tokens (unit of a data type, i.e. double, integer, string) to the simulated wireless media, making use of output ports, and receiving the corresponding ones through their input ports.

A basic simulation scenario constructed on Ptolemy II, using the Visual Sense applications, can have the appearance of Fig. 1.3, where we can appreciate four important actors as example: a Wireless Director, a Communications Channel which takes into account the power loss of a signal, and two composite actors representing a transmitter and a receiver. The wireless media director manages the scenario for it to be executed using the characteristics of a wireless media (represented in Fig. 1.3 by the actor “PowerLossChannel”) for data transmission between two composite actors.
If we take a look inside of the composite actors, we can find algorithms with diverse complexity level, representing the functionality of a certain node. In the example scenario, we have a transmitter sending a signal at random intervals using a clock activated periodically around an average time modified by a Poisson random distribution, producing tokens with alternating values “0” y “1”, and an output port to send the token out of the composite actor (later to the wireless media). It is important to notice that this node is controlled internally by a discrete event director “DEDirector” instead of a wireless director, because there is no internal wireless media in the node, but a wired algorithm executed sequential and iteratively. The receiver node is a little more complex, because it reads the signal power of the received token, and represents it immediately in a time chart over an output interface. This value will depend on the distance between nodes, which can be manipulated by the user by moving any of the nodes during the execution time in the simulation, because the media bounded to power losses diminishes the signal power proportionally to the mathematical value of the square of the distance. The output from the simulation will result in graphs like the one given in Fig.
1.4. Notice that “TimedPlotter” (see Fig. 1.3) is an actor representing the existence of an output interface to visualize information about the simulation.

![Fig. 1.4. Example of data visualization in Ptolemy II](image)

Viptos, the subset of Ptolemy II for Wireless Sensor Networks, provides additional actors to translate simulation models to NesC programming language [11], base programming language for applications running under TinyOS operative system. Example of sensors based on TinyOS and thus NesC are moteiv’s Tmote Sky [9].

1.4.2. Actor inner-processing inside a simulation

Actors in Ptolemy II follow an execution process composed by six “action methods” [16], programmed in Java language and executed in the following order:

- **preinitialize()**, invoked exactly once at the beginning of the execution, generally used to establish data type restrictions,
- **initialize()**, invoked once during execution, typically to initialize state variables; after its execution, the actor experiments some iterations, composed by the three following action methods,
- **prefire()**, invoked once per iteration, returning a Boolean value (false or true), indicating if the actor is ready to be fired\(^3\),

---

\(^3\) To fire an actor refers to the fire() function calling, where the main purpose of the actor is executed.
• **fire()**, is the main execution point, and is generally responsible for input reading and output production, being able to be called once or more per actor execution,

• **postfire()**, returns a “false” Boolean value to indicate if the actor execution has been completed, avoiding to iterate it again during the current model execution. It can be called at most once per iteration,

• **wrapup()**, used typically to visualize final results and to unlock resources; it is invoked exactly once at the end of execution, including the case in which the execution is stopped by an exception.

Through the execution of this sequence in each actor in the model, giving to each one the opportunity to be executed through steps (iterations), Ptolemy II simulates concurrency between actors execution. The already existent actors can be expanded through creation of new actors, adopting their functionality through inheritance and method overload for the six methods mentioned above.

### 1.5. Tiny Discovery Protocol

Tiny Discovery Protocol (TDP) describes a simplified mechanism for non related nodes to start a communication with each other, with the purpose of advertising and discovering available services in a wireless sensor network.

TDP follows three main steps to achieve its purpose, using a client-service approach for the communication parts:

- **Discovery/Advertisement**: The server will send a broadcast message announcing its presence in the scenario, and the server “type” that it constitutes, with the purpose of advertising servers by its general purpose instead of sending a list of all offered service to every client. The type of a service should represent the collection of functionality for a determined purpose (i.e. weather services, global positioning system, etc.). In case that a node gets into the network in need for a service, the client itself could sent also a broadcast message to find out if there is a present server of the type which fulfils its needs.

---

4 Broadcast is understood as the action of sending data intended to reach all nodes present in the scenario. This term can refer to sending a data packet, or to transmit in a media where all nodes have access (i.e. air)
∇ **Description:** When a client is interested in a server, it will ask for a server description, where the server informs all the available services corresponding to the advertised server type. The server will send an XML (eXtended Markup Language) structured message to present the services as independent items, and be able to characterize them in case it is needed to indicate details about the offered service.

∇ **Service Utilization:** Once a client requires a service, it will request it using a simple packet, directed to the selected server, indicating name and input parameters for the service. In a similar way, the server will answer to this request by sending a packet with the output values (response) of the service. Some example of output results from services can be URLs (Universal Resource Locator), fail/success acknowledgements for the service execution, crypt seeds, and data to start the usage of other protocols for usage of more complex services.

A first set of packets was proposed in the early version of TDP protocol, which can be appreciated in Fig. 1.5, corresponding the described three-steps procedure.

---

**Fig. 1.5. TDP packets for a service discovery and usage case**

---

A service can be composed by one or more actions, which are similar to function calls in structured programming languages, using input parameters and producing output values after its execution. An event is defined as one execution of the action. Those services are
described in the early version of TDP using an XML syntax described in the Document Type Definition (DTD) for Tiny Discovery Protocol (see Fig. 1.6) [16].

```
<!-- TDP DTD Version 0.2, National University of Colombia -->
<!-- Id: TDP.dtd,v 0.2 03.12.2007 16:26:46 eal Exp $ -->
<!-- Experimental DTD for Tiny Discovery Protocol - Service Advertisement. -->

<!-- Identifies the SERVER we are describing through its ID -->
<!ELEMENT server (act)+>
<!ATTLIST server
  id CDATA #REQUIRED
  type CDATA #IMPLIED>

<!-- Defines an ACTION (equivalent to function) that can be invoked -->
<!ELEMENT act ( sub | in | out)*>
<!ATTLIST act
  name CDATA #REQUIRED>

<!-- Inform if the service is able to be subscribed (default: yes) -->
<!ELEMENT sub EMPTY>
<!ATTLIST sub
  able (yes|no) "no">

<!-- Define the type of the INPUT parameter for the action -->
<!ELEMENT in EMPTY>
<!ATTLIST in
  type CDATA #REQUIRED>

<!-- Define the type of the OUTPUT parameter for the action -->
<!ELEMENT out EMPTY>
<!ATTLIST out
  type CDATA #REQUIRED>
```

Fig. 1.6. Tiny Discovery Protocol – DTD for service description Version 0.2

The XML description of a server presents the actions listed in a hierarchical way, containing attributes on their tags, and internal tags to represent characteristics such as amount, type and order of input and output values, and ability of an action to be subscribed. Notice that a server has to contain one or more actions to justify its advertisement. The server can modify dynamically this XML description, to inform only a set of actions to determined clients, because of security policies or deficit of resources.
For better understanding of the service description usage, Figs. 1.7 and 1.8 present examples for a service with input and output parameters and a service without parameters, respectively, and the corresponding TDP packets created for their utilisation.

**Fig. 1.7. XML description of an action with input and output parameters (left) and request (center) and response packets (right)**

<table>
<thead>
<tr>
<th>Client Request Action</th>
<th>Server Executed Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDP-0.5</td>
<td>TDP-0.5</td>
</tr>
<tr>
<td>RT:RA</td>
<td>RT:EA</td>
</tr>
<tr>
<td>SID:EEEF</td>
<td>SID:EEEF</td>
</tr>
<tr>
<td>CID:AAAA</td>
<td>CID:AAAA</td>
</tr>
<tr>
<td>ACT:Temp(C,true):0001</td>
<td>RET:(25):0001</td>
</tr>
</tbody>
</table>

**Fig. 1.8. XML description of an action without parameters (left) and request (center) and response packets (right)**

<table>
<thead>
<tr>
<th>Client Request Action</th>
<th>Server Executed Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDP-0.5</td>
<td>TDP-0.5</td>
</tr>
<tr>
<td>RT:RA</td>
<td>RT:EA</td>
</tr>
<tr>
<td>SID:EEEF</td>
<td>SID:EEEF</td>
</tr>
<tr>
<td>CID:AAAA</td>
<td>CID:AAAA</td>
</tr>
<tr>
<td>ACT:TriggerAlarm():0002</td>
<td>RET():0002</td>
</tr>
</tbody>
</table>

This protocol is used as communication model for the simulation, and is modified to a more current version to fit the purposes of this research.
2. HIERARCHICAL TREE CODIFICATION

Representing the structure of composed services through hierarchy trees facilitates the management of such services, as it allows the node to know how its services are correlated. To easier its manipulability, giving a code or identifier to the nodes help us to distinguish each node as a unique object, and depending on the codification, we can generate identifiers related to the position of the node in the tree or even to some of its links, allowing us to recognize the parents or children of the analysed node. This recognition will be useful for later processing on activation/deactivation of services in a managed node.

When leading with trees having more than 2 children in some of their nodes, simple binary codifications could not be fitted to generate optimal identifiers for each node. In the following, we analyse two methods to generate binary identifiers for every node, first by simplifying the tree into a binary tree and codifying it, and later by assigning identifiers to each node in a n-ary tree by means of binary strings with one significance bit different from zero. Finally, we will take components of this two approaches, to generate an identifier codification method for n-ary trees.

2.1. Binary Path Representation of N-ary Hierarchy Trees

There exists already a mapping technique to obtain a binary tree out of a n-ary tree, done by reorganizing the links between the nodes. This transformation follows some simple rules (see also Fig. 2.1 to visualize them):

- Every X node in the n-ary tree will have its equivalent X’ in the binary tree.
- Any node will have at the most two children: left and right (binary tree).
- If the node X has children, the first (leftmost) child of X on the left side in the n-ary tree, will be the left child of X’ in the binary tree.
If a node $Y$ has at least one sibling to the right, the first sibling of $Y$ to the right in the n-ary tree, will be the right child of $Y'$ in the binary tree$^5$.

Fig. 2.1 Binary tree transformation of a n-ary tree

This mapping is usually referred as first-child/next-sibling representation of a tree [9].

If we want to identify every child of a node, we could start naming the main node with a single bit, and every subsequent node with the name of its parent (refer here to the binary tree representation of the hierarchy), plus an additional bit, where this bit would be “1” if the node is the left child of its parent, or “0” in case it is a right child, and repeating the process the same way for every node. The result can be better appreciated in Fig. 2.2 through an example.

Fig. 2.2 Binary path codification for nodes of a n-ary tree

---

$^5$ Siblings are nodes which are children of the same parent node
Every node is coded this way through a bit string with variable length. This bit string will represent the path followed from the tree root, going from parent to child, from child to siblings, moving as necessary based on the binary representation of the n-ary tree. Additional to this string, a code mask with the same corresponding size will be generated simultaneously, in order to register movements inside and between levels, this will ease later the analysis made to calculate the existence of a parent/children relation between two nodes. Fig. 2.3 represents the concepts of parent, child, sibling and levels in a tree hierarchy.

![Fig. 2.3 Representation of base concepts in a hierarchy tree](image)

Every movement between levels (i.e. going from a parent to a child) will be represented through a “1” valued bit in the code, while any movement to a right sibling will add a “0” valued bit. Every bit added to the identifier code will add a bit in the mask, changing its value when the new code bit is “1” (jump from a parent to a child) and repeating the last bit value when the new code bit is “0” (jump from node to sibling). By default, the code for the root node will be “1” with a mask “0”.

To clarify, the first child of the root node will have a code “11”, where the first bit to the left is inherited from its parent, and the second refers to the “parent to child” movement needed to get to this node. The corresponding mask will have the value “01”, where the first bit “0” is inherited from its previous node (root node in this case) and the bit “1” represents the value change due to the level change. Fig. 2.4 gives an example of the codes and masks generated for the nodes in an example tree.
Notice that every node will contain to the left, in its code and mask, the code and mask of its previous node in the path to the root node. This could be also referred as code/mask inheritance. Additionally, this codification will allow us to know in which level the node lays in the tree structure, since the number of ones in the code is obtained going down through the tree, this way, a node with the code 1100011 should be in the fourth layer of the tree.

To add new nodes to an existing hierarchy, the correct placement is identified in the hierarchy, and the corresponding code is generated. If the new node will be the first child of another node, it will take the code and mask of its parent and add a 1 (level change) to the code, and the opposite bit to the last mask-bit of its parent to its mask. If there exist already children of its parent, it will take the code and mask of the last already existing child of its parent, and add 0 to the code and the same last mask-bit of its sibling to its mask.

2.2. Node representation through Significance Bits and Code Inheritance

This rather extensive but clear codification model, identifies a node through the concatenated codes of its parents, plus a new bit string where most values are 0, and only one significance bit \(^6\), which differentiates every node from its sibling, has 1 in such position. To exemplify the

---

\(^6\) Significance refers to the magnitude represented by a bit in a certain bit string. For example, an 8 bit string will normally have its most significance bit to the left and its less significance bit to the right, given that a change in the left-most bit will change the value in 1 unit, but a change in the right-most bit will change the value of the string in 128 decimal units.
meaning of significance bits, Fig 2.5 represents the significant position of 4 bits, where the last bit on the left will be the Most Significant Bit (MSB) and the last bit on the right will be the Less Significant Bit (LSB). This way, when a node has 4 children (siblings between them), their own-part codes will correspond to 0001, 0010, 0100 and 1000.

![Fig. 2.5 Bit significance in a 4-bit string](image)

To understand it graphically, Fig. 2.6 represents an example using this codification.

![Fig. 2.6 Codification through Significance bits and Code inheritance](image)

Even though every code will have an unnecessary amount of bits with value “0”, it will allow us to combine two identifiers when any 2 sibling classes are parents of the same child. For example, if two siblings have identifiers 010 and 001, they will inherit to their child the code 011 for the parents field, which can be done through the simple use of a logical operation “OR” between the two parent codes. Fig. 2.7. shows clearly an example of it, where S₉ has the value 110 as parent code.
As the nodes have to be represented through their code, given normally as a bit string where the inherited codes are present to identify the parents of the service in the upper levels, we can make use of a masking method to differentiate the level components of a service code. To achieve this in an easy way, we will alternate the usage of “1” and “0” values in the mask, setting “0” in the places where there are bits in the code which belong to odd layers. The same way, we will set “1” in the mask, in those places where there are bits in the code which belong to pair layers. Fig. 2.8 shows the bit strings corresponding to the code and mask for an example service.

Finally, to add new nodes to the tree, the placement is identified in the hierarchy and the corresponding code is generated. If the new node will be the first child of another node, it will take the code and mask of its parent and add a 1 (level change) to the code, and the opposite bit to the last mask-bit of its parent to its mask, being “0” if the node is placed at an odd layer, or “1” if it is placed at a pair layer. If the code is intended to inherit from two nodes (siblings between them), the code resulting from a logical OR operation between their codes, will be taken as base code for the new child, adding 1 (level change) to the code, and the corresponding “0” or “1” to the mask, depending on the last bit of the parents mask (or the level where the new node is placed). If there exist already children of its parent(s), it will take the result from a logical OR operation between the codes of its parents as the inherited part of
the code, then it will add a bit “1”, and then as much “0” bits as bits present in its sibling code with the longest own-part code (last fragment of its address, see Fig. 2.8). The mask will be established by setting “0” at the odd level positions, and “1” at the pair level positions.

2.3. Codification schemes unification

In the past two sections, two binary codifications were introduced, the first one following the path from the node root to the coded node by coding a level change through “1” and a jump to next sibling through a “0”. In the second one, every node has an inherited part from its parent(s), plus an own part composed of a bit string where only one significance bit has value “1”, in a position where no sibling has this bit set to 1, to differentiate it from its siblings.

To analyse the requirements of such codifications in terms of amount of data required to codify a tree, we can take a simple example and codify it through the aforementioned methods. Fig. 2.9 show an example tree codified through both methods, where the binary path codification utilizes 82 bits codifying the tree while the significance bit codification utilizes 102 bits.

If we think about the node addition in those trees, specifically for the significance bit codification, we can appreciate that every node has an own-part code with a number of bits equal to the number of siblings who share the same parent (i.e. four siblings would have own-part codes 0001, 0010, 0100 and 1000). If we experiment by taking out of the code the “0”
bit(s) at the left of every level code, we will be able to shorten some of the codes, without losing any important information about the node, like it is represented in Fig. 2.10.

![Fig 2.10 Code abbreviation for “significance bit” method](image)

Now, looking at the resulting code/masks obtained through this abbreviation, we can notice that the resulting codes are exactly the same ones obtained through the binary path codification, thus allowing us to obtain the same 82 bit representation of the tree by abbreviating the significance bit codification.

Following the analysis, we can take now into account the node aggregation in the tree. Given that significance bit codification requires that every node has an own-part code with a number of bits equal to the number of siblings who share the same parent with some unnecessary bits, node aggregation for the significance bit codification could imply the modification of the already existing codes for the siblings of the new node, and their respective children, as seen in Fig. 2.11.

![Fig. 2.11 Node aggregation in a “significance bit” coded tree and later abbreviation](image)

If we would make this aggregation into the binary path coded tree, the final tree would be the same from Fig. 2.11 (significance bit), as we proved before that the abbreviation of the
Significance bit codification results in a binary path coded tree. Notice that in the worst case for a tree, every new node will require two bits more for its code/mask set (one each), since the next-sibling of a node and the first-child of the same node have the same length, thus making variable the relation or rate between the number of nodes and the amount of required bits to codify the whole tree. However, in the worst case the amount of bits required to codify a tree will be equal to \( 2 + 4 + \ldots + 2n \), where \( n \) is the number of nodes present in the tree. Given that \( 1 + 2 + \ldots + n \) is equal to \( n(n+1)/2 \), we will find out that the number of bits required to codify the worst case tree is equal to \( n(n+1) \).

Finally we need to be able to codify nodes which are children of multiple parents, in order to support multiple inheritance at some level in this codification scheme. As mentioned in section 2.2, a node who inherits from different parents will take as inherited part in its code, the result of a logical OR operation between its parents codes. Starting from the binary path representation of the tree, as a logical operation between two bits string suppose that both bit strings have the same length, we will make the length of the parents code equal, by adding bits “0” on the left in such level-part codes where the size of such part does not have the same length. The mask has to be extended correspondingly by repeating the bits already existent for that level, remembering that the mask represents which positions in the code correspond to odd/pair level fragments. Fig. 2.12 exemplify the process made for two example nodes which want to inherit to a new node.

Fig. 2.12 Multiple inheritance codification. Logical operation OR between two codes (left) and node location in the tree (right)
Notice that a node which inherits from two parents will not be considered sibling of a node which shares only one of its parents, but sibling of those nodes which have exactly the same parents of it.

2.4. Relationship evaluation

One of the main purposes of this codification, besides identifying a node in a uniquely manner, is to offer a mechanism to recognize whether two nodes are related in the hierarchy tree as sibling, as well as direct parent or remote parent and child. This analysis is possible due to the analysis of the code by levels.

2.4.1. Sibling relationship evaluation

When two nodes inherit from the same parents they are said to be siblings between them. As two sibling nodes inherit the same inherited-part code\(^7\), they will share a common part in the left of their identifiers. For this assumption to be valid, their codes will be compared, comparing the whole left part of their code (inherited code), up to their private own-part code (last segment of the code). Fig. 2.13 shows some example of evaluations to determine if pairs of two nodes are siblings between them. Only the last part of the code, which indicates the unique value for the node in the level where it is located, will be excluded for the comparison. The rest of the code will be subject of comparison, and only in such cases, where the inherited part (left part without the last segment) is exactly the same, it will be said that the evaluated nodes are siblings.

\(^7\) see Fig. 2.8 for definition of “inherited-part”
Note that for the comparison between services $S_{11}$ and $S_{12}$ in Fig. 2.13, the third-level part of $S_{11}$ was extended through left “0” bits addition, still finding that such services are not sibling related.

### 2.4.2. Parentship evaluation

When a node inherits from another node (single inheritance), it will also inherit its code, which could be at the same time partially inherited from parent nodes two or more levels above the analysed node. In other words, a node will inherit its parent code, which was also generated by inheriting the code of its parent’s parent, containing in its code information about parent ship from several layers above itself. To make this comparison, we will take the code from the node supposed to be parent of its comparison counterpart, and will look for the exact existence of this code in the inherited part of the analysed child node, by means of calculating a logical AND operation between the supposed parent code and the analysed child code. This comparison has to be made level-part by level-part, extending the code-parts through “bit 0 addition at the left” in the smaller code when it is necessary to compare two code-parts with different size, as explained before. Whenever the result of the AND operation is equal to the value of the supposed parent, it will be determined that the supposed parent is in fact parent of the analysed child, as seen in Fig. 2.14. This detailed comparison level by level is needed to avoid recognizing parent ship between two nodes which have the same inherited code but different inherited masks.
For nodes with multiple parents, it is important to remark that if a level code (code bits from a certain level) from the child has more bits set to “1” than its parent’s code for the same level, the child is inheriting from more than one parent on that level. In this case, the bits set to “1” in the parent should be set to “1” in the same locations in the level code of the child, which is already tested by the AND operation proposed to verify parentship between two nodes. When the parentship is found, it will determine that the child node inherits from the supposed parent code among other possible parents. It is important to notice that logically the number of level components in the analysed codes has to be different, as a parent should have less level components on its code than its child. For that purpose, it is important to take the mask into account for parentship analysis as they represent the level components, to avoid two nodes appearing to be parentship related when analysing the codes only, as seen in Fig. 2.15.

Fig. 2.15 Example of erroneous parentship analysis when taking nodes with the same number of level components
When analysing nodes for common inheritance, the case when two nodes share some parents, the comparison between nodes with the same number of level components will be allowed, but always taking the mask into account for a proper comparison. The code which is equal to the result of the AND operation, will correspond to the code of the node who has less inheritance, meaning that all its parents are parents of its comparison counterpart, but its counterpart may have additional parents. The only case where a both codes are equal to the result of the AND operation, is where a node is compared against itself. In Fig. 2.16, an example is depicted, where the node A and B have common inheritance, and the set of parents of B contains the parents of A.

Notice that all nodes in the trees have a first-level part with value “1”, which is the code of the root node, since they all inherit from it. Notice also that as much as the parent code coincides with the inherited part of the child node, the parent ship between the nodes will be closer, where the root node has the farthest parent ship and the direct parent (first upper level) has the closest parent ship, as the code of the direct parent will be equal to the whole inherited part of the child’s identifier, and not only equal to part of it, as it is the case of farther parents. Fig. 2.17 shows an example node with its different parents.
3. SERVICE AVAILABILITY MANAGEMENT

In this section, the availability problem is analysed and defined, proposing a methodology for service deactivation in a hierarchical composition model, based on metrics for evaluation of service execution costs.

3.1. The Availability Problem

The reliability of a system requires the correct coordination of all its parts to work together, performing their duties in a timely manner, such that no part of the system will suffer delay by waiting for answers from external resources/services. For that purpose, availability is a key factor, as it represents the probability that the system will be operating at a certain time, or in other words, the ratio of the time a functional unit is capable of being used out of a total time interval [10].

Composable services will be seen here as systems which are composed of smaller services which perform small duties in a collaborative scenario, designed to achieve a bigger task representing the service itself. A service is provided if one or several concurrent processes are successfully executed in a timely manner. A process itself, can be described by all possible sequences of actions therein, where each requires some specific resources.

Introduced this way, the availability problem for services is “under what conditions a request for a service leads to the provision of that service, and how such conditions can be satisfied in a timely manner”. The interaction among the involved processes may affect the sequence of actions (and the required resources), which may vary from time to time depending on the involved processes and available resources. Therefore, the analysis of service availability requires the examination of the involved processes and their interactions, as well as their corresponding resources in time [22].

As some resources could not be available at a certain moment in time, it is ideal to dispose of different alternatives to execute a service by using alternative and/or fewer resources.
Wireless Sensor Networks are characterized by the presence of several nodes, where some of them share the same capabilities, it is possible in some scenarios to find different alternatives (i.e. nodes offering the same sub-services) to execute a high level service. First, the HSIP model, explained later in section 3.2.1, proposes a good mechanism through which a service can be described in different manners, by having different “maps” on how to execute such service, using different sets of sub-services and/or resources. This maps are called “configurations” for a service, and can be seen as different trees describing different ways (requiring different sets of components each) to execute the service. Second, whenever a service cannot be provided due to a high amount of resources implied on its execution, a hierarchical model of service inheritance where every child is an extended version of its simpler (more abstract) parent, could be used to select the parent of a service as its replacement, thereby offering less capabilities but still offering some service related solution with lower resource requirements. Two corresponding hierarchical models are presented in section 3.2 to support the manageability of service availability provision.

3.2. Hierarchical Models for Composable Services

A composable service can be defined in different ways. As composed services could have different levels of complexity, their anatomy can be structured often as the composition of another smaller services, where a service can be seen just as the execution of different smaller (atomic) sub-services. Another approach is seeing a service as a subtype of other services, raising their complexity by inheritance and thus complementing their functionality, as classes do in object oriented programming. This approaches could be described easily by representing the service structure through hierarchy trees. In the following in this section, two models are presented to manipulate service composition: HSIP for sub-services composition and Type Inheritance for functionality inheritance.

3.2.1. HSIP: Hierarchical Services of Interacting Processes.

As an example of an approach on seeing a service as a composition of smaller subservices and their interaction, the HSIP model, proposed by Fallah and Sharafat [22], describes high level services as the result of executing a sequence of atomic services (outcomes of actions at the
bottom of the hierarchy having no sub-action within), utilizing a hierarchical tree to describe configurations as the way that atomic services interact to execute the composed service.

In this hierarchy a high level service could be executed through different configurations, which describe a selection of atomic services and their interactions to produce an output out of a possible input of the service. This way, a service execution can be described in different ways by using similar resources depending upon their availability. Notice in the example in Fig. 3.1, that for every service and sub-service, there is one or more configurations to lead to its execution, for instance, service $s$ can be executed either through $c_1$ or through $c_2$ independently, while $s_1$ can only be executed using configuration $c_3$, which requires a service $s_{12}$, used also by $s_2$ through its configuration $c_4$.

![Hierarchical Composition Diagram](image)

*Fig. 3.1 Example of a hierarchical composition for a service “s” under the HSIP approach*

This structure give us as starting point a hierarchy, where a *high level service* is composed by smaller services, and a small service can be needed to execute different *high level services*. Notice that this way, an *atomic service* can take place as child of several services, and if it is imperative to execute those services, the unavailability of this *atomic service* could lead to the unavailability and deactivation of their dependent *high level services*.

The HSIP can be used to model the multiple configurations of processes involved in the provision of a requested service with the objective of improving the availability of that service.
through fault tolerance mechanisms (i.e. quick reaction to unavailable sub-services by replacing them with available ones). It can also be used as a model for dynamic and adaptive resource allocation, denial of service, and quality of service [22].

### 3.2.2. Type inheritance

This model describes a composed service as an extension of basic defined services, where a child service takes the functionality from one or more parent services by inheriting their structure, and complementing it with its own characteristics. This can be used to diminish functionality of offered services in order to save resources as battery energy, when a complex service already offered is replaced by a parent service, offering now less functionality in the same kind of service.

For better understanding of this structure, Fig. 3.2 represents a hierarchy tree, where every child service inherits from one or more parents, and adds its own extensions, the same way as class inheritance is done in object oriented programming.

![Type inheritance model for composable services](image)

Notice that this structure allows to have multiple inheritance, where a service can have multiple children, and be child of multiple parents, making its children dependent of their stability, as the unavailability of a parent service implies the unavailability of its children.
This approach can be mainly used to manage on-site (internally) the offered functionality of services in every server node, taking decisions on the complexity of the offered services based on the amount of available resources (memory, battery, etc.).

### 3.3. Abstract Reachability Graph

During the analysis for service provision, where a node has to evaluate the disposable alternatives to execute the high level service, it is sometimes going to be needed to evaluate, among various factors, the replacement of currently unavailable services or sub-services with another ones of the same or similar type, available in the environment (i.e. in the same node or closer nodes). To achieve this task, a reachability graph is able to assist the analysis, by offering the node the possibility to include factors like the distance or response time in its evaluation for the replacement of a sub-service selected out of a set of available candidates in the environment. A graph drawing should not be confused with the graph itself (the abstract, non-graphical structure) as there are several ways to structure the graph drawing. All that matters is which vertices are connected to which others by how many edges and their characteristics and not the exact layout [13].

![Graph example](image)

**Fig. 3.3. Graph example**

These graphs, as exemplified in Fig. 3.3, can follow simple rules of graph theory, and are proposed mainly to provide a tool for including the connectivity conditions (like distance, throughput, etc.) among the metrics used to select best choices, attributed to the edges, which
represent the connectivity. The Vertex, in the other hand, will be the conceptual presence of a node in the graph.

3.4. Service Management

To manage service composition and availability, two key hierarchical models were presented for decision making at the availability management stage. The first of them, based on the HSIP service representation, will bring the description of a service as the execution and interaction of small atomic sub-services, which compose the main service, also referred as high level service in this document. The second hierarchy will bring an inheritance map, where the relationship between services from the same kind will be shown through an inheritance tree, where the root service represents the most basic service of its genealogy, and will inherit its characteristics to most complex and extended child services, as inheritance does in object oriented programming. Additionally, a reachability graph is proposed to retain connectivity conditions among the factors to be judged in the decision-making processes.

These two hierarchical representations are going to be used by the managed node to analyse and evaluate:

1. How to execute a service (cheapest resource alternative)
2. How to replace a service description (abstract service replacement)

For a node to manage the availability of its services, it needs first to have knowledge about the disposable resources (i.e. battery level, available memory, etc.), and the resource requirements of the services that could be offered. To be able to evaluate and compare services and configurations based on their requirements, it will be useful to establish certain characteristics of each service, such as memory requirement, estimated execution time and priority, among others. Based on these metrics, the managed node will be able to make decisions on service availability, by choosing to offer: all services, a set of low cost services or a set of high priority services, depending on the management policy. Some metrics proposals are described in the following, in order to set an starting point for analysis in availability management.
3.4.1. Resource Metrics

Some resources in a node are general to all offered services, since the execution of a service implies the their usage independently of the nature of such service. Critical resources in sensor embedded systems are battery, memory and processor, given the “small and disconnected” design for individual devices. Some metrics depending on such shared resources are:

- **Battery level**: Percentage of remaining charge in the battery.
- **Memory availability**: Amount of disposable working memory (RAM) to execute services in the server node.
- **Processor activity**: Percentage of time that the processor is already busy in a given time interval.

Sensor devices are typically feed from a battery as energy source, having small RAM memory resources and processors with neither high speed nor multithreaded capabilities, where operations as processing, utilization of wireless interfaces (high energy-consumers) and data input and output to/from memory storage and memory buffers will traduce in battery consumption, being part in the management of the battery as a resource. Processor speed and usage will additionally traduce into delay to attend other services when a service demands too much processing time, as well as the memory availability will disable other services to be executed as soon as possible.

Metrics based on resources will be mainly used by clients to choose between different servers to request a service, and by servers to know when to diminish the amount of offered services in order to save resources.

3.4.2. Service Metrics

Every service execution implies the consumption of system resources, and is characterized by its importance in the network, often understood as priority. For a service to be compared against other services (heading to select groups of low-cost services, high-priority services, or mixed groups of services to offer), each service will be qualified dynamically in key aspects
that represent the cost of executing a service. Individual or sets of such metrics will be grouped in cost functions, which will be used as the final comparing and analysis tools.

These cost function values will be attached to service and sub-service descriptions, to be taken into account when deciding about requesting (client case), executing or offering a service (server case). Some proposal for metrics that could be used to evaluate a service are:

- **Processing time**: Estimated execution time of a service. It can vary dynamically depending on the current work load in the server node, or the clocking speed of its processor.

- **Memory Load**: Rate of memory required by the service to memory available in the executing/server node. It varies dynamically for a service, depending on the current available memory in the server.

- **Priority**: Importance of the service in the network (or system as a whole). A service which does not demand extensive usage of the device resources (i.e. low cost) not necessarily has to be offered. Instead, an offered service should be highly useful by itself when offered, avoiding offering simple services as getting date or time from server clock, since the service could exist and be useful for composed services, but could waste resources when offering to all foreign devices. This indicator would be abstract and dependent on the service/network designer.

- **Usage**: Rate of the amount of requests for a single service to the total amount of service requests made to a server in a certain sampling interval. Such interval can be measured in time (i.e. requests in the last hour) or in terms of “last $n$ requests” (i.e. number of requests for the analysed service out of the last hundred request calls). When there is only one service offered in a server, the “last $n$ requests” measure will be ineffective as the cost will rise until it reaches a rate of 1 (i.e. hundred calls of a service out of the last hundred calls). This metric can be used to detect a service that is being used excessively, and discourage its usage to save resources for less often used but important and/or unique services, as the highly requested service can still be used from other servers. Notice that a high cost for a service does not keep a service from being offered, but just discourage its usage from an specific server. If the server is however the only node offering the service, even if it has a high cost, it
will be also the cheapest and only choice in the network to get the service, and will still be used by the clients.

∇ **Presence**: Rate of one service to the amount of servers offering the same service. If a service is offered by only one server, its cost should tend to be cheaper than other services offered in the same server, but if there are a big amount of close servers offering this same service, its usage should be discouraged to save resources for services which are rare or unique in the network. Notice that the smallest the rate of a service (rate is always between 0 and 1), the higher its cost should be. In other words, rate will be 1 when there is only one server offering the service, but the cost should be minimal to promote its offering over other services being already offered in multiple additional servers, therefore it would be appropriate to express the presence of a service as \( l - r \) where \( r \) represents the above explained rate.

∇ **Subservices**: Amount of subservices it depends on, as they could be executed every time the service is invoked. It can also be taken as the functionality that a service offers, since a service composed of several subservices is more likely to offer a wider variety of uses and atomic functionalities (responses or specific data that can be obtained from such service).

∇ **Transferred data**: Amount of data received and transferred from and to the wireless media bounded to the usage of the analysed service. This metric includes in the analysis the cost of using the wireless interfaces, as wireless data transfer consumes high amounts of energy compared to other processes. It will mainly be useful for environments where the services retrieve big amounts of data as results other than simple types as double, integer or similar.

∇ **External calls**: Amount of calls to external subservices needed to execute a service, which exist out of the local server, since it implies the usage of wireless interfaces and new service requests. This metric is particularly focused to environments with services composed of atomic subservices rather than simple local services.

The set of metrics finally used in every management scenario, should be determined by the management depth and approach, depending on whether it is more important to keep availability for the highest amount of services, or for the key network services.
3.4.3. Deactivation policies

A managed node who is making decisions on service availability has to modify dynamically the set of offered services, as resources become accessible/inaccessible and the system functionality policies are followed. Deactivation is a key factor on service availability management.

Two policies are analysed in this document, recognized as two typical cases of service availability management:

∇ **Service variety offering:** This approach will be used when the purpose of availability management is to sustain the maximal amount of services available in the network, whenever a network has a wide variety of services and most of them are simple or have small resource requirements. It is useful when all nodes are reachable among them, or when they can behave as intermediate nodes (proxies) to bring a service closer to clients which cannot reach the server, mainly in scenarios where the server nodes are deployed in static locations, since the servers will need to have some knowledge about the services offered by other servers, and this demands high consuming data traffic in dynamical networks. The metrics recommended in this scenario will be usage, presence, and memory load, as they will give priority to less often used, unique and small non-consuming services respectively.

∇ **Key services offering:** This approach will be used when the server nodes are focused to bring a specific service, due to network requirements (services needed to achieve the goal of the system) or local capabilities (nodes equipped with a specialized function, like Global Positioning System, Internet access, etc.) not generally found in every device for whom the network is intended. The metrics recommended in this scenario will be priority, transferred data, and subservices, to promote key services, tend to keep low consumption, and preserve services by lowering down their functionality while keeping them available (i.e. replacing services with more abstract services of the same type).

In order to manage resources, simple mathematical functions are determined, whose values indicate the cost that a service will imply in terms of one or more resources, to be able to
compare them. Such functions can exist in both parts of the communication, using them in the server to optimize resource consumption, and in the clients to determine the server to request for a service, thereby optimizing local and system resources, where the system can be understood as the whole environment or network, which contains multiple nodes interacting as clients and servers. For the client to be able to take decisions based on the alternative costs, the data overload required to transfer information regarding metric values could imply also a disadvantage as it requires additional battery consumption for periodic updates, reason why, the management in the server nodes is the main concern for service availability and resources management.

The HSIP model describes composable services as sets of atomic subservices, executed in order to achieve the high level service’s goal. For this reason, it will be useful for the purpose of finding the cheapest alternative for executing a service, by studying the service composition hierarchy, and execute its parts (atomic subservices) using the known sources which offer the lowest costs. On the contrary, the type inheritance model will be useful when making decisions about service functionality, as complex services will require higher resource consumption, and can be replaced dynamically by resources from the same nature with less functionality and requirements. For each of these models, a reasoning will be made when making decisions about service/sub-service deactivation.

### 3.4.3.1. Deactivation using the HSIP model

In the HSIP model (see section 3.2.1), composed services are presented as a combination (configuration) of multiple smaller services, also called “atomic services”. As the high level services need every component of such combination to be executed, failures on the service availability of atomic services, will lead to the disability to provide or execute a service. Nevertheless, when an atomic service is available from different providers, the composed service can still be executed by replacing the disappeared or failed service by the next attractive alternative of an atomic service or even of the whole composed service. This decision should be based on the execution cost of every alternative service.

The deactivation of services will take place as the resources are diminished, mainly as the battery level gets lower, but also when the processor and memory are busy due to other
processes consuming the resources of the node (see section 3.4.1). When the battery reaches a certain level, the most expensive services will start to be deactivated or replaced.

In Fig. 3.4, an example is represented for a service unavailability in the HSIP model. When service $s_{12}$ goes unavailable for any reason (i.e. deactivation due to low battery level), and there is no cheaper alternative to get the service executed, services $s_1$ and $s_2$ cannot be executed, since they are configured to use $s_{12}$ to successfully achieve their tasks. Services $s_{11}$ and $s_{21}$ are atomic services which, since they do not depend on $s_{12}$, are not deactivated. The higher level service $S$ will also not be deactivated, since it knows another configuration $c_2$, through which its execution can be obtained by means of executing another service $s_3$.

![Fig. 3.4 Service deactivation in HSIP model](image)

The same way, if $s_{12}$ would have another configuration different than $c_7$, it could have been replaced by a substitute service. Different configurations for the same service can be found for instance, when a service can be executed locally with a given resource cost, or externally, with a given cost in terms of wireless data transfer. This analysis will be useful for servers and clients when deciding how to execute composed services out of a group of known subservices, remembering that it will require higher data transfers or programmed knowledge about the service architectures in the nodes taking part in the network system, this way, this management mechanism will be recommended for networks pursuing the availability of service variety, explained at the beginning of this section.
3.4.3.2. Deactivation using the Type Inheritance model

When there is no execution alternatives, the type inheritance model will bring additional possibilities to keep a service available. Another way to continue offering a service when it becomes too expensive to stay offered, is to select a more abstract version of the same type of service, sacrificing functionality for a better cost. For this purpose, we will take the type inheritance model as starting point to find services with diminished capabilities of the same type.

Fig. 3.5 Type hierarchy example

Fig. 3.5 depicts the hierarchical composition of services of the example type $S_1$. Every service inheriting from it, will take its functionality and complement it to become a more complex service, which will imply normally higher requirements. Some services will also inherit from different types, meaning that they at least collect their capabilities and are able to extend them. For instance, a navigation system can inherit GPS (Global Positioning System) and map navigation types to extend them by bringing route tracing capabilities. In the abstract example in Fig. 3.5, the service $S_{1,123,1}$ bring the most complete functionality collected from its inheritance from $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ types.

Using this model, servers can focus on offering a complex but meaningful service as long as their resources allow it. When resources have to be saved or are simply not enough to execute the complex service, such service can no longer be offered and the type hierarchy model will allow for an easy recognition of services of a more abstract type, which offer some of the deactivated service capabilities. For example, when the whole route tracing processing (including location finding and map processing) becomes too expensive in a navigation
system, the service can be replaced by a location finding service (GPS) that still could be used to find the current location over a local map in the client.

When a composed service becomes too expensive to be offered, those services which also inherit from some of the same parents will become additional possibilities, and if they are still affordable, can be taken by the server as then next services to be offered, because of their similar nature (common inheritance) with the now deactivated complex service. When they become at their time too expensive for the remaining resources in the server node, finally the base types of the service will become the offered service catalogue. It is mandatory to manage service costs for every type, since it will allow to determine when a service has to stop being offered, and which services fit in the current capabilities of the server. In some designs, if a complex service should be only replaced by its base types and no other nodes with common inheritance, the analysis will not take into account alternative services in the same level of the type hierarchy tree, where the deactivated service is located, as seen in Fig 3.6.

To ease the deactivation procedures in processing time, a codification scheme will be useful to allow for the rapid comparison of service identifiers and recognition of hierarchical relationships. The codification scheme introduced in section 2 brings the possibility of recognizing whether two services are as siblings or parent-children related, as well as if they share any inheritance in upper levels. Using the procedures explained in section 2.4.2 for evaluating the parentship relation between two services (recognizing whether a node is children of another one), the catalogue of offered services can be dynamically adjusted as response to the failure and deactivation of services. An example codified tree is depicted in Fig. 3.7
When a service type (refer here to the type inheritance model) cannot be anymore offered because of a failure or lack of resources, its closer parents will be selected to be offered as replacement, as long as the available resources are enough to execute them. If the resources are still not enough to execute its direct parents, the tree will be followed upwards, to find the closer parent who is affordable to be executed, and this one will be offered on its place. When a service inherits from multiple service types, all the parent types will be offered, as long as there are enough resources left to execute them.

Additionally, when a service type has to be deactivated, the codification will be used also to identify the children of such services and deactivate them, since a complex service should not be offered if one of its components is not anymore available. To ease this part of the analysis, it could be helpful to take a look to an inverted sketch of the tree, as depicted in Fig. 3.8, where a parent type can be seen easier as a component of a subtype (more complex type). To resume the analysis, Fig. 3.8 shows an scenario where unavailable service type $S_{1,2}$
deactivates all services which inherit from it (use it as component), and where $S_{1,13,1}$ remains as the more complex service which can be offered.

### 3.5. Deactivation Procedure

The management of service availability will take place during time in the managed nodes, making effective the dynamic change of the set of offered services. To follow this task, the node will evaluate at certain points in time the affordability of each service, based on the utilized cost metrics and the current local resource availability.

The first step on deactivation will be to decide how often the set of advertised services will be evaluated, where a node could to this task periodically or based on certain events. Following the most used ways of advertising, a server should advertise the environment periodically about its existence, for passive clients to get to know about it when they have already another known server delivering their needed requests, in order to allow for the work load to be distributed among the old and the new arrived servers. Additionally, given that the server advertisement can be deactivated for many design reasons, the evaluation of service affordability in the servers should also be done whenever a client is discovering a server, which can happen when a known server disappears for the client and it needs to discover new available servers to requests its needed services from them. This way, the evaluation and deactivation procedures will take place by *periodical advertisements* and triggered by *server description request* from the clients.

The second step will be to determine which services from the hierarchy will be offered at a certain point in time. For this purpose, the server will evaluate the cost of a service (in case the metrics depend from current resources like memory load or other), or read it from an already existing database of static costs for the services (for metrics like priority, which don’t change through time). Once the cost is resolved for every service, depending on the current resource availability (battery charge, available memory, etc.), the server will deactivate (take out from the list of services to offer) the services which, given their costs, are not affordable to be offered. After cost resolution and deactivation, the server will proceed to evaluate which services are already covered by complex services which inherit and extend their
capabilities, using an inheritance tree like *type inheritance*, presented in section 3.4.2. Summarizing this steps, the server will:

1. Decide to evaluate service affordability (cost evaluation) when server description is requested or a periodic advertisement has to take place.
2. Deactivate services whose costs are not affordable given the current local resources availability.
3. Deactivate all services which have affordable children in the inheritance hierarchy (thus not counting as affordable the already deactivated services from step 2).

![Service deactivation procedure](image)

Fig. 3.9 Service deactivation procedure

This procedure will allow us to offer at every point in time the most complex services which are affordable for the server to offer, offering as few services as possible but covering through them all the affordable functionality from the server. Fig. 3.9 depicts the sequence followed for the deactivation procedure. Notice that the deactivation of basic types (step 3) will make use of the codification scheme introduced in section 2 to optimize the parent-children evaluation between services.
3.6. Conclusion

The problem of optimizing availability of composable services in wireless sensor networks, is solved in an scenario where server nodes have to manage resources to provide long operating life and availability of services by means of selecting sets of services to offer and configurations to execute such services such that the resource consumption is minimized without sacrificing the stability of the system (as a whole) due to delays and timeouts in response times.

HSIP and Type Inheritance models ease the analysis of service composition, and constitute a tool for implementation of service availability management in sensor devices. Deactivation policies and metrics were proposed for service evaluation through cost functions based on the type inheritance model, as it offers the most practical approach for offer/deactivation analysis, being complemented by codification of inheritance trees using the code inheritance codification scheme proposed in section 2.
4. SIMULATION MODEL

To test the functionality of the deactivation mechanism proposed in section 3, a simulation model was designed and implemented, making use of the Ptolemy II framework, concretely using its derivative package “Viptos”, developed for simulation of wireless sensor networks.

4.1. Communication Model

The first step on simulating cooperative work between mobile nodes, is setting a communication protocol for their understanding. As the purpose of the simulation is the service availability management, the protocol TDP is taken as start point and modified for better usability in networks with very limited battery resources.

The communication will succeed in three stages, depicted in red, green and blue words respectively, in Fig. 4.1:

- Server Advertisement / Discovery
- Service offer
- Service usage

---

Fig. 4.1. Communication model for service delivery

---

8 See section 1.4: Ptolemy II
9 See section 1.5: Tiny Discovery Protocol
4.1.1. Data packets

Every step in the communication process, will be done through a message from source to destination. This packets were defined in version 0.5 of TDP protocol, and are slightly modified for its usage on the simulation, giving origin to version 0.6 of the protocol. Fig. 4.2 depicts the human readable packet structure of the communication messages, as they are utilized in the use case simulation.

![Fig 4.2. Communication model and corresponding data packets](image)

Fig 4.2. Communication model and corresponding data packets

![Fig. 4.3. Binary data packaging for TDP 0.6](image)

Fig. 4.3. Binary data packaging for TDP 0.6
As the human readable structure is not optimal for wireless communication between battery dependent devices, due to the high energy consumption from the wireless interface, binary packets are additionally proposed for the effective transmission of messages, whereby this representation is being used in the simulation only for accounting of transferred data amount, in order to register the battery consumption as it will be in the ideal real implementation, but keeping the human readable packages for better understanding of the communication process when working with the simulation. The proposed binary data packaging is presented in Fig. 4.3, and based in the data link frame structure and size of standard IEEE 802.15.4, used by some devices as the Tmote Sky [17][19][20].

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Binary Code</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery</td>
<td>0000</td>
<td>DS</td>
</tr>
<tr>
<td>Advertisement</td>
<td>0001</td>
<td>AD</td>
</tr>
<tr>
<td>Get Description</td>
<td>0010</td>
<td>GD</td>
</tr>
<tr>
<td>XML Catalogue</td>
<td>0011</td>
<td>XML</td>
</tr>
<tr>
<td>Request Action</td>
<td>0100</td>
<td>RA</td>
</tr>
<tr>
<td>Executed Action</td>
<td>0101</td>
<td>EA</td>
</tr>
</tbody>
</table>

*Table 4.1. Request types and corresponding binary values*

The fields depicted in Fig. 4.3, are proposed to be used as follows:

- ▼ TDP: Identifies the packet as part of a TDP communication, containing the binary representation for the characters TDP following a code like ascii.
- ▼ Version: Binary value of version number. As it is on its early stages, this protocol version will be treated as version 0, having a binary representation
- ▼ Request Type: Identifies the type of request contained in the packet, for the receiver to interpret the fields. The requests used in this research will be identified as indicated in Table 4.1.
- ▼ Service Type: Identifier name for the service wanted to be discovered. It is limited to 16 Bytes (i.e. Temperature)
- ▼ Server Type: Identifier name for the kind of server being advertised (i.e. Tmote, GPS, etc.). It is limited to 24 Bytes.
- ▼ ClientID and Server ID: Network identifier for the node. Limited to 32 bits.
- ▼ CallID: 2 Byte integer to identify an specific call to a service.
This field format will be subject to changes as further development is made in the protocol, or for specific implementation requirements. The above mentioned fields are proposed for statistic purposes on data amount calculation during the simulations in this research.

4.1.2. XML Server description

The biggest concern of service offering for TDP protocol is its XML presentation, as its structure can waste big amounts of data due to its human understandable format, albeit it keeps being used because of its flexibility to extend service management to clients by giving the option to send metric values in the XML catalogue, for them to have knowledge about the local situation of remote servers and make decisions about the source to be used to request for a service. For instance, when a client knows 3 servers around him offering the service he is interested in, he can choose to make requests to the server with higher battery level or with higher available memory (among several metrics), if the information was sent in the XML catalogue of offered services.

```xml
<server id="SERVERID" b="CURRENTBATTERYLOAD" ssn="EnvironmentalServices">
  <act ssn="NetEq" c="1" m="0" p="0"/>
  <act ssn="Temp" c="11" m="011" m1="200" p="10"/>
  <act ssn="Light" c="110" m="01111" m1="200" p="20"/>
  <act ssn="Humid" c="1100" m="01111" m1="400" p="20"/>
  <act ssn="TempLight" c="1111" m="01110" m1="300" p="30"/>
  <act ssn="TempHumid" c="11111" m="01110" m1="500" p="30"/>
  <act ssn="LightHumid" c="111101" m="011110" m1="600" p="40"/>
  <act ssn="AmbientFactors" c="111111" m="011110" m1="700" p="50"/>
</server>
```

Fig. 4.4. XML Server description

To start with the service evaluation, every server will dispose of a server description template in XML, where every service is listed with its attributes inside a group tag that constitutes the server being described. Using this structure, a server can hold information in a group of tags, where it can store the knowledge about other servers in its area, or have different service sets wrapped in server tags with different subserver names "ssn" when there are several functionalities in the server, as a server could bring different services depending on the variety of resources it has (antennas, sensors, etc.). Every server will have only one server "id", but can dispose of several subserver names, each identifying a set of composable services for a determined functionality (i.e. environmental services, navigation services, internet access point, etc.). Using the id attribute of each server tag, the server can store and differentiate in
its database information about remote services offering groups of services. An example of an XML Server description is presented in Fig. 4.4., where the use of the attributes for server and actions can be appreciated.

Additionally to the above mentioned “id” and “ssn” attributes for the server tags, servers can hold an attribute “b” to inform the current battery level to its clients, for them to select their request objectives among several server offering the same services, by choosing the server with more remaining battery charge, as a way to contribute to the stability of the network. As the delivery of information about metrics was already said to be optional, the servers can choose not to inform about their battery level, thus saving battery by shortening the amount of sent data. This lack of information can be interpreted by the managed clients as a battery deficit in the server, thus considering the server with unknown battery level as the last alternative to use when choosing a server to request for a service.

The actions inside a server descriptions (understood as the service types to be offered and marked with a tag name “act”), can hold several attributes to work with their identifiers and metrics when processing in the server the affordability of a service to be offered. In the example seen in Fig. 4.4., the field “ssn” is used to hold the subservice name and the fields code “c” and mask “m” are being used to hold the identifiers for each subservice on the type hierarchy tree structure\textsuperscript{10}. The fields “p” and “ml” are chosen to hold information about priority and memory load, as metrics to evaluate the cost of each service when managing the service availability in the server. Notice that field “ssn” can hold up to 24 characters for servers and up to 16 characters for actions, as well as the “id” field is restricted to 4 characters (32 bits), to fit the bit packaging scheme presented in figure 4.2.. Additionally, as needed by the management implementation, other metrics\textsuperscript{11} can be included in the action tags using the attributes to store their values, since this XML Server description is not going to be fully sent to clients before being processed.

\textbf{4.1.3. XML Catalogue}

Based on the XML database with corresponding metrics for the services to offer, the managed server will proceed to evaluate the ability of services to be offered given the current

\textsuperscript{10} See section 3.4.3.2.: Deactivation using the type inheritance model

\textsuperscript{11} See section 3.4.2 for recommendations on other metrics able to be used to evaluate service affordability
availability of resources (battery charge, memory availability, processor availability, among
others). After choosing the services to be offered, it will send the results in a shorter XML
version to the clients, where there are tags only for the services to be offered, and attributes
are present, except for the subservice/subserver names “ssn” and the server identifier “id”.
Notice at this point, that it is optional to include the attribute “b” (battery level) to give the
clients the choice of selecting a server among different alternatives based on a meaningful
criteria.

```xml
<?xml version="1.0"?>
<server b="0.38663785952" id="EEE" ssn="EnvironmentalServices">
  <act ssn="Temp"/>
  <act ssn="Light"/>
  <act ssn="Humid"/>
</server>
```

*Fig. 4.5. Example of an XML Catalogue received by clients*

The shortened XML document sent to clients will be referred as XML catalogue, as it is a
collection of offered services at a given time, and will be packed and sent inside data link
frames, using the bit packaging scheme proposed in Fig. 4.2.. Fig. 4.5. shows an example of
an XML catalogue extracted from the XML Server description proposed in the Fig. 4.4.. The
choice to compress the XML document will be left to the programmer, and is possible
through binary packaging of values (service names, battery level, server name, etc.), or
through text compression (i.e. zip compression), and will not be treated in the present
document.

### 4.2. Use Case: Service availability for composed ambient factors services

As a guide example on managing composable services, a small set of composed services for
ambient factors measurement was proposed, elaborating a *type inheritance* structure and
codifying it using the *code inheritance* scheme proposed in section 2. The group of measured
ambient factors was inspired in the sensors array available in the Tmote Sky device, which
disposes of temperature, luminance and humidity sensors [17].
Fig. 4.6 Set of composable services used for the simulation

Taking the temperature, luminance and humidity as services that can be requested in a simple way and expect a numerical value as response, they were proposed as simpler service types which will inherit their functionality to more complex services, which make use of this simpler services combined. To ease the conception of every derivated service, they are taken as simple combinations of consulting their parent types. The hierarchical structure resulting from creating these services is shown in Fig. 4.6., where the services TempLight, TempHumid, and LightHumid inherit from two simpler types, and the service AmbientFactors inherit from all three simpler types, resulting as the more complex and complete service in the hierarchy.

4.3. Simulation Scenario

In the simulation, two types of actors were deployed simulating client and server nodes. The communication model explained in section 4.1 is implemented through composite actors inside servers and clients, making server advertisements periodically, or triggered by discovery requests made from the clients. The server nodes manage an XML Server description based on their battery and memory levels, producing an XML Catalogue that can be published to the clients.

The simulation scenario deploys four servers and four clients in a limited zone, where the clients move independently based on a movement pattern [15], and every node has a transmission range of 500 distance units (based on Ptolemy's default distance units), covering an area of 1400 x 1000 units through the static servers location, as depicted in Fig. 4.7.
The main colour of each node will change progressively as the battery gets discharged, having a light green (servers) or a light blue (clients) colour when the battery is fully charged, or a red colour when the battery is fully discharged (both kind of nodes). Two movement patterns can be used to set mobility inside the nodes, one designed for groups and guided by a movement manager [15], and one independent for each node, generating random movements updating the location of the node periodically, using every time two Gaussian distributions around the current position for the \( x \) and for the \( y \) coordinate.

**Table 4.2. Simplified consumption model used for simulation**

<table>
<thead>
<tr>
<th>Operating condition</th>
<th>Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU on, Radio TX</td>
<td>21.8 mA</td>
</tr>
<tr>
<td>MCU on, Radio RX</td>
<td>19.5 mA</td>
</tr>
<tr>
<td>MCU idle</td>
<td>54.5 µA</td>
</tr>
</tbody>
</table>
Each server node is charged with a 250 mAh battery, and utilizes a consumption scheme similar to the characteristic one of Tmote Sky devices [17]. The simulation takes into account the battery consumption due to data reception and transmission, based on packet sizes for TDP according to the binary data packaging model proposed in section 4.1.1., at a rate of 250000 bps. Additionally, a continuous 54.5 µA consumption is taken into account, similar to the idle state consumption of Tmote devices. The simplified energy consumption scheme used in the simulation is described in Table 4.2.
5. MANAGEMENT SIMULATION ANALYSIS

Four simulations are executed, varying the priority of the offered services, heading to study the impact of managing service availability in the battery lifetime. This variation impacts the time when the less important services are going to be deactivated in each server, starting sooner or later to make services unavailable. For that purpose, a priority value is set for each service, where only one service in this example will be offered until the end. Fig. 5.1 presents the service hierarchy indicating the priorities used in one of the executions, where the “Temp” service takes the highest importance, represented through the lowest priority value equals to zero. This means that the server will deactivate this service only when the battery reaches a charge of zero percent, or in other words, when the battery gets totally depleted. Correspondingly, to ease the deactivation understanding in this example, the priority of each service will indicate at which battery level will this service be deactivated. For instance, the service “Humid” will be unavailable in each server when the battery charge is under 40% and the complex service “TempLight” will be deactivated at 30% in the example presented in Fig. 5.1.

![Service Hierarchy Diagram](image)

*Fig 5.1. Example of priority assignment 40-20-0 used during the simulation*

After running the simulation instances, some interesting data is extracted from the different output files and charts, resumed in tables 5.1 and 5.2. To summarize the different assignments, the priority arrangement column describes only the priorities used for the base service types “Humid”, “Light” and “Temp”, as they will be available when their children types with higher priority values are offered.
Notice that for the arrangement where all services have priority 0, the availability of services will be permanent, being equivalent to a non-managed system where all services are offered until server batteries are depleted. Every server node starts with a maximal 250 mAh battery load. The values in tables 5.1 are measured after 3000 simulated seconds (50 minutes), when all server batteries still have remaining battery charge, with an average demand of 3 base services per each 10 seconds (one of each subtype) from each client. The values in tables 5.1 and 5.2 are defined as follows:

- **Average Service Demand**: Amount of times that a service is needed by each client in the lapse of 3000 seconds. It summarizes the demand of all basic types of services at a rate of 18 requests per minute. This value includes discoveries, timed out requests and successful requests.

- **Average Success**: Average of the rate of success service requests to total requests for a service (total includes succeeded and timed out requests).

- **Average Timeouts**: Average of service requests to servers which were known to be offering the service, which have not been answered and have expired due to timeout. It includes timeouts because of service deactivation, and clients getting out of transmission range, where the known server is not reachable anymore.

- **Average Discoveries**: Average amount of discoveries sent out by each client during the first execution period of 3000 seconds. They will be higher as the service becomes unavailable. Notice the difference with timeouts, as discoveries are sent when no server providing the service is known, and timeouts are requests for a
service to a known server, which does not answer such request, inferring thereby that the server went unavailable due to distance or service deactivation.

∇ **Average Requests:** Average number of requests for a service sent by the client to known servers offering the service. It includes requests which were successfully answered as well as timed out requests.

∇ **Average Battery Life:** Average of the time elapsed until the battery level reaches zero in each server node. Measured in seconds.

∇ **Battery Level:** Battery charge remaining in every server node after 3000 seconds (50 minutes).

∇ **Availability:** Average of the rate of the time interval where each service was successfully reached from each client, out of 3000 simulated seconds. It measures the average uptime of a service in the network, rather than the rate from successful request to total number of requests for a service, based on the fact that servers cover the whole scenario and therefore services will be always available, up to the point in time where they disappear due to deactivation.

### 5.1. Analysis 1: From Managed to Permanent Availability

Based on the mentioned indicators, we can compare the effect of starting the deactivation process at sooner or later stages. In this example, for the four given priority arrangements, the average remaining battery after 3000 seconds is graphically represented in Fig. 5.2.

![Impact of Management Start on Battery Level](image)

**Fig. 5.2. Average battery load after 3000 seconds, starting deactivation at different stages**
Notice how the remaining battery charge in the servers will be lower as the service deactivation start is delayed, due to the higher energy savings when some services stop being provided earlier in time. However, an interesting fact arises in this graph, where we can notice that making no deactivation of the service has provided an average remaining battery charge similar to the case of an early management, and higher than doing management at late stages. If we take a look to the availability provided by each priority arrangement during the measured 3000 seconds interval, depicted in Fig. 5.3., we could incorrectly infer that there is no advantage on doing service deactivation nor in terms of remaining battery neither in terms of service availability.

![Average Service Availability](image)

**Fig. 5.3. Overall average service availability**

If we take a closer look to the communication model proposed in section 4.1, we notice that when no servers providing an interesting service are known, clients will try to discover new servers, replacing the traffic generated by service requests with traffic produced by discovery packets looking up for new useful servers. To avoid overloading the network, the servers only answer discovery packets when the requested service is still in the catalogue of offered services, but cannot avoid spending energy in the reception of discoveries, since some of them are asking for a service that they are in fact offering and this uptime for the wireless interface will inevitably consume energy on data reception.
Making now use of the general data consumption chart through time, we can appreciate in Fig. 5.4 the close behaviour between the earlier deactivation start and the non managed scenario up to the measured 3000 seconds time, and we find that the average of the remaining battery load after the measured 3000 seconds period will be consumed at a faster rate in a non managed scenario. It is important to note at this point, that some of the servers reach a 0% battery level between the seconds 3000 and 3600, and thus the average diminishment will be strong in the non-managed scenario, since the few nodes with remaining battery will provide the same 3 services, while the few nodes with remaining battery in the managed scenario will provide only one service at this stage.
5.2. Analysis 2: Discovery impact over Availability Management

The service availability management has to be well complemented by a service provision protocol which utilizes less energy in data transmission for its discovery mechanism than the energy implied in the execution of a service and the transmission of request and answer packages for the service. This way, the benefit of service availability management will appear whenever the amount of data transmitted for an unsuccessful discovery is at least less than the amount of data transmitted to request and obtain the service, inferring no energy consumption in the execution of such service. In such cases where the service execution (excluding data transmission) implies a very high battery consumption, the difference will start to be more evident.

Revising Table 5.3., where the statistics of timeouts and discoveries after 3000 simulated seconds are presented for the four management scenarios. The evident augment on the number of discoveries as services are deactivated sooner shows the traffic caused by discoveries as the main cause of battery consumption when a service is not consuming energy anymore after its deactivation. Notice that the average number of times that a service is demanded (column “Average service demand”) remains homogeneous, as the simulations were configured with a constant average of 3 service demands per each 10 seconds (one of each three base service types), for an average value of nine hundred demands for a service in a time lapse of three thousand seconds.

<table>
<thead>
<tr>
<th>Priority Arrangement</th>
<th>Average service demand</th>
<th>Average Timeouts</th>
<th>Average Discoveries</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 – 40 – 0</td>
<td>925.25</td>
<td>21.5</td>
<td>346.75</td>
</tr>
<tr>
<td>40 – 20 – 0</td>
<td>913.75</td>
<td>45.25</td>
<td>154.75</td>
</tr>
<tr>
<td>20 – 10 – 0</td>
<td>919.25</td>
<td>24.25</td>
<td>161.5</td>
</tr>
<tr>
<td>0 – 0 – 0</td>
<td>889</td>
<td>28.25</td>
<td>40.75</td>
</tr>
</tbody>
</table>

Table 5.3. Statistics for unsuccessful requests.

Notice also the not strong difference between the amount of timeouts taking place during the simulations, given that they are originated when the servers get out of transmission range from the clients due to the movement pattern, and very few times due to the deactivation of services, since only two services are deactivated per server, and the worst case would be a client being in the transmission range of all servers, and noticing the deactivation through timeout rather than through advertisement updates. Remember also that the simulation
utilizes periodical updates, which are common to all simulated cases and allow clients to know about changes in the catalogue of offered services without experimenting a timeout to establish the disappearance of a service source.

![Average Discovery issues](image)

**Fig. 5.5. Average discovery requests issued by clients in each management scenario**

In Fig. 5.5, the progression of accumulated discovery issues is represented for the four simulated priority arrangements, appreciating also an obvious strong augment on the number of discovery issues between seconds 3000 and 3600, where some servers are already totally unavailable due to battery depletion.

![Battery consumption with and without discovery accounting](image)

**Fig 5.6. Battery consumption with and without discovery accounting (minimal impact case)**
If we take the amount of discovery requests issued by each client and calculate the amount of energy consumed by each discovery message reception, we can get an idea of the behaviour of the battery depletion ignoring the impact of discovery messages in the battery life. Fig 5.6. presents the slight difference for each priority arrangement between the battery depletion caused by all factors (bigger areas in the chart) and the depletion caused by all factors with exception of discovery traffic, making a first supposition that every discovery is listened by only one server (having a fully covered scenario). Even though the discovery consumption is visible, it does not appear to impact enormously the battery consumption on servers. Fig. 3.7 shows a new comparison between the battery consumption in the four priority arrangements, where the managed availability starts to appear more attractive than the non managed permanent availability of services.

![Battery Consumption ignoring discovery issues](chart)

**Battery Consumption ignoring discovery issues**

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>0</th>
<th>600</th>
<th>1200</th>
<th>1800</th>
<th>2400</th>
<th>3000</th>
<th>3600</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 – 40 – 0</td>
<td>100,00%</td>
<td>83,08%</td>
<td>67,67%</td>
<td>52,32%</td>
<td>35,89%</td>
<td>20,28%</td>
<td>10,54%</td>
</tr>
<tr>
<td>40 – 20 – 0</td>
<td>100,00%</td>
<td>82,83%</td>
<td>65,00%</td>
<td>48,20%</td>
<td>31,40%</td>
<td>13,85%</td>
<td>4,18%</td>
</tr>
<tr>
<td>20 – 10 – 0</td>
<td>100,00%</td>
<td>80,87%</td>
<td>62,24%</td>
<td>44,09%</td>
<td>27,34%</td>
<td>9,85%</td>
<td>2,39%</td>
</tr>
<tr>
<td>0 – 0 – 0</td>
<td>100,00%</td>
<td>83,01%</td>
<td>66,89%</td>
<td>50,49%</td>
<td>33,82%</td>
<td>17,72%</td>
<td>3,61%</td>
</tr>
</tbody>
</table>

*Fig. 5.7. Average battery consumption through time ignoring discovery traffic consumption*

If we analyse the contrary extreme case, where all clients broadcast discovery packets in a zone where they are reachable by all servers, the battery consumption due to discovery traffic
will multiply by the number of clients present in the scenario, in our case, four clients. Remember that this would be an extreme case, and the amount of servers receiving the same broadcasted discovery packets is dependant on the current position of the issuing client, and in the simulation, it will be randomly determined by the movement pattern. However, it is interesting to take a look to the extreme case, since it represent the maximal amount of battery that the issued discovery requests can consume, and shows more clearly the worst-case impact that the presence of various clients issuing discovery requests could have in the overall lifetime of the system.

![Graphs showing battery consumption with and without discovery accounting](image)

*Fig. 5.8 Battery consumption with and without discovery accounting (maximal impact case)*

Figs. 5.8 and 5.9 present the corresponding maximal impact of discovery broadcasting by multiple clients looking for disappeared services (for our simulated scenario), reinforcing the need to have an optimal design for the discovery protocol. Taking into account the impact cases presented in this analysis, we can guess the consumption of the discovery traffic as a random value between those minimal and maximal impact values, lowering the benefits of service availability management in scenarios where a lot of clients start broadcasting discovery requests as consequence of a service deactivation.
Battery Consumption ignoring discovery issues

Fig. 5.9. Average battery consumption through time ignoring discovery traffic consumption (maximal impact)
6. CONCLUSION

After studying the concept of service availability, the viability of managing service provision using a hierarchical composition approach was analysed, proposing sets of metrics to determine service costs as starting point to compare services and be able to take decisions on whether a service is affordable to be offered, as the resources get diminished or unavailable through time, as it is the case of battery depletion and memory overload.

Based on the hierarchical representation of composed services, a deactivation procedure was proposed and simulated, designing a binary codification scheme to identify nodes in a hierarchical tree, in such a way that the comparison between the codes of two nodes let us know if there is a parent and child relationship between them, supporting inclusive trees where there is multiple inheritance. Utilizing this scheme, the tree processing was highly simplified to recognize in a fast way whether a service inherits its functionality to more complex services, and choose only the most complex services in a tree which are affordable to be offered in every point in time, based in the current resource availability. This will result in an abbreviated service catalogue, ready to be sent to service clients.

Using a protocol of own design, and modifying it to improve its capabilities, a communication model was introduced and simulated, making use of protocol packages to coordinate the communication between simulated nodes, and a binary data packaging scheme to account the amount of transmitted data, in order to be able to work with battery as key resource to be optimized in the server nodes, through service availability management.

The benefits of service availability management were evaluated through several simulations using different priority arrangements for an example set of composed services, measuring importance function and performance factors, like successful requests, timed out requests, and discoveries generated by the clients to find new sources of a service. The different priority arrangements were thought to evaluate the different penetration in time of the service availability management, starting it at early or later stages of battery life, including the case of...
waiting upon battery depletion (charge equals to 0%) to start management, or in other words, not doing any management at all.

The above mentioned simulations revealed the strong impact of the discovery mechanism and data traffic on the benefits of service availability management, as the battery charge saved by deactivating the offer of a service, will be partially utilized by the wireless interface when receiving frequent discovery requests from clients, which don’t find any alternate server offering the service. This discovery requests need to be still listened and processed, as they could be requests for services which are still offered. In case of a bad design for the discovery mechanism, it could waste even more battery energy than it would be needed if the services remain offered, when the packets used for the discovery request and reply exceed the sizes of the packets used to request and reply the service provision.

Focusing on the benefits of the service availability management, some charts were produced to reflect the behaviour of the average battery life in the servers network depending on the different start stages of services deactivation. It confirmed the higher average battery life in the servers when the management is started early, at a cost of lower the average availability of services, measured as the average time that every service was being offered by the servers. This availability was relatively stable across the network since the clients were requesting the services to the known service having the highest remaining battery, which represented an extra cost for data transmission of this value, but helped for an even distribution of battery consumption across the network, while the nodes were moving over the coverage area.

Finally, even though the service availability management through composable services was shown to be positive for the life time of the system, its benefits are limited by the impact of the resulting discovery traffic coming from clients looking for the disappeared services. This impact is recognized as a variable dependent on the density of clients around each server, as the deactivation of a service could generate a high consuming storm of broadcasted discovery requests, coming from various clients to the server continuously. To avoid this, the improvement of low cost discovery mechanisms is recommended, i.e. by utilizing small data packets as the one proposed for discovery in the TDP protocol. Additional improvements could be done by studying the uptime management of the wireless interfaces through synchronization with clients for limited and periodic service availability intervals.
FUTURE WORK

As the optimization of resources is mandatory in wireless sensor networks, further developments are researched continuously in this field. Following the line on which this thesis is developed, some ideas are proposed for continued work towards the objective of overall availability in battery dependent devices.

∇ Tiny Discovery Protocol proposes a simple alternative to discovery, advertisement and provision of services, though improvements can be further done in the current proposal of binary packaging. The XML format of service catalogues is specially expensive due to its human-readable format, wasting data by sending some values through their character representation, plus data for document syntax. The format for the data packets could be improved to minimize sizes, being aware of variable size information as the codes and masks proposed in the code inheritance scheme for tree codification.

∇ In cases where there is no need for multiple inheritance, the masks will be redundant to the codes, as the jump from level to level will add a “1” valued bit and from sibling to sibling will add a “0” valued bit. Therefore, a simplified case can be proposed to process trees without multiple inheritance with a reduced algorithm to analyse the parent-child relationship between nodes.

∇ To avoid broadcast storms from clients trying to discover sources for their required services, the viability of an algorithm for synchronized service availability intervals could be studied, for cases where clients are not in immediate need for a service and servers could provide services during short uptime periods, maintaining their wireless interfaces inactive most of the time, thus helping to improve battery lifetime.
REFERENCES


