ABSTRACT
Reducing the energy consumed by mobile wireless devices to extend the lifetime of the batteries that power them is one of the major challenges in designing such systems. While this problem can be addressed at various levels (e.g. device, operating system, middleware, application), we think that it is possible to get a higher energy saving when considering the needs of the applications (application-aware). Our approach is centred on analyzing the behaviour of the applications more than the resources. In this paper, we present an application-aware framework which utilizes the needs of the applications and Dynamic Power Management (DPM) techniques focused on the wireless network card to control and reduce power consumption at runtime. We include a novel technique that increases the energy savings. We have also developed a simple prototype to test the viability of our idea. The experimental results show that the proposed approach achieves between 10% - 40% average reduction in the energy consumed by the wireless network interface.

Categories and Subject Descriptors
C.2.1 [Network Architecture and Design]: Wireless communication; C.5.3 [Microcomputers]: Portable devices.

General Terms

Keywords
Mobile applications, Network and power management, XML-based policies, Application awareness.

1. INTRODUCTION
Mobile computing systems are constrained by scarce resources, such as small memory, slow CPU, etc [1]. They are specially constrained by limited battery capacity due to the weight/size limits of the battery. These limited resources (e.g. battery or memory) and the lack of continuous connectivity among others, force the applications to be adapted or even redesigned before they can be inserted in mobile environments.

Different strategies can be applied to try to reduce the use of energy in mobile wireless devices but it still remains as a major challenge. These strategies are usually applied at different layers e.g. device level, operating system level, middleware or less frequently at application level. We think that considering the needs of the applications (so called application-aware) can lead to better results with relatively small effort. Therefore our approach is focused on how the applications are going to use the wireless interface more than on the other resources.

The remainder of this paper is organized as follows: In section 2 we present how to model the application behaviour using XML, some considerations of the main aspects of this model and example scenarios in which it is useful. In section 3 we review the bases of Dynamic Power Management and we describe how to combine it with the needs of the applications. Section 4 presents a framework where these techniques are applied. Section 5 details the measurement apparatus and it presents the experimental results. In sections 6 and 7 we conclude the paper with some related work in this area, the conclusions and the future actions in the development of our framework for mobile applications.

2. MODELLING APPLICATION BEHAVIOUR
From an operational point of view, mobile applications can be oriented to work with data (data-centric), services (service-centric) or both. Working with data implies, most of the times, the need for a database in the mobile devices. These mobile databases are usually part of a greater one, belonging to the enterprise boundaries. A synchronization process would later assure that the system remains consistent. So, if we want to model the application behaviour (that is, how the database is likely to be accessed), we have to know the synchronization needs, not only for the whole database but also for each one of the tables or data within. Regarding the services, or more precisely, their invocations, we should know the endpoint (regardless the technology) and if possible, how long it takes to execute the service or at least when the result should be obtained.

Based on these two concepts (data and services) we have defined an XML-based language (called ANE-SML (Application Needs
Specification Mark-up Language) to model the expected and desired behaviour of an application. This language covers the aspects related to the data and the services. With this information, and the operation of the application, the framework will be able to decide not only when to connect to enterprise services, but also what data has to be sent or received, but each time just what really needs to. For example, if the user is a commercial agent, it might need to have updated information of the availability of certain products (forcing to connect to a server every half an hour, for example) but the new contacts made along the day might wait to be updated in the database at the end of the day.

In summary, applying these policies the use of the network interface can be scheduled in order to achieve energy savings and reductions in the communication costs. As a result, there is an inherent trade-off between energy and data delay/freshness, both of which are aspects to be analyzed when designing the policy for the application.

### 2.1 XML Connection Policy

Table 1 shows an example of the connection policy related to an imaginary mobile application that has both a mobile database and also performs service invocations.

**Table 1. Example of an XML Connection Policy.**

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <dataConnectionRules syncMode="total" defaultTimeout="08:00:00">
    <table name="Clients">
      <noChange timeout="1h"/>
      <anyChange timeout="20m"/>
    </table>
    <table name="Orders">
      <anyChange timeout="5m"/>
      <insert timeout="1m"/>
    </table>
    <dataConnectionRules>
      <serviceConnectionRules>
        <service targetUri="http://www.myenterprise.com" defaultResponseTime="30s">
          <method name="WSCheckStockValue" requestTime="2m" serviceTime="2s" responseTime="1s"/>
          <method name="WSBuyShare" serviceTime="30s" responseTime="2s"/>
        </service>
      </serviceConnectionRules>
    </dataConnectionRules>
  </dataConnectionRules>
</mosePolicy>
```

The meaning of this policy is as follows:

**DataConnectionRules**

This tag includes the information related to the mobile database, divided into tables. For each table you can set a number of attributes that will tell when to synchronize, based on the application behaviour. We have divided these attributes according to the common activities that can be made with a database. The value of **timeout** of these attributes resembles the maximum amount of time to wait before the table needs synchronizing:

- **NoChange**: the table has not varied since the last synchronization.
- **AnyChange**: a change has been made to the table (no matter what type).
- **Insert**: new rows have been inserted.
- **Update**: some rows have been modified.
- **Delete**: some rows have been deleted.

SyncMode and defaultTimeOut attributes help define special features if needed. While the first one tells whether synchronization affects the entire database or only the tables that really need it, the defaultTimeOut parameter establishes a shared timeout for the synchronization process, so that we do not have to define every single table, but we can define only special conditions and use the defaultTimeOut for the rest of the database.

To be able to implement this schema, the actions that are being performed against the database during the lifetime of the application need to be monitored. This can be easily accomplished by slight modifications of the database access interface, intercepting all the invocations and notifying the changes in the database.

**ServiceConnectionRules**

Basically, we have defined an invocation to a service as an URI, a specific method, and a set of times expressing different requirements of the service:

- **requestTime**: it determines how long an invocation can be delayed, that is, the maximum amount of time that can be waited before invoking the service.
- **serviceTime**: it represents the estimated duration of the execution of the service.
- **responseTime**: it determines when the response should be processed, in other words, the maximum amount of time that can be waited before returning the response.

Combining the values of these attributes, we have classified the invocations to a service in groups. Table 2 shows a summary of our classification. Depending on the type of each invocation, the time attributes should be considered in different ways. This classification could help the developer define which type of invocation is going to be used so that it can be automatically processed. This classification affects the way the invocations should be treated for the resource management. For example, if the request Time is not required, those services can be delayed until another process needs connection, so the system will connect only once.

Using different times for the request and the response allows the integration of asynchronous services as well as the synchronous ones.

This classification can be extended if particular cases are considered. For example, we can define a service to be invoked immediately as a special case of “Specific Time” behaviour. There are also combinations that have no sense being considered, because they refer to impossible scenarios where the requestTime is greater than the responseTime.

Defining this kind of policy for a complex application might seem a hard task, but as in the database rules, we can define default times for the services. And even more, if we have a service or a database that has not been defined in the policy, it is considered to be irrelevant for the resource management, so they will be executed according to these default times.
This policy can be looking at the definition of the policy we can see that the Orders and orders in the way the developer has designed and forgets knowing anything about the database; he manipulates the clients and orders placed by them. This database synchronizes without a policy, the user has to decide when the database is synchronized. He may not know the number or kind of the tables it has, so normally he will synchronize the entire database. However, using a well defined policy, the user does not require it has, so normally he will synchronize the entire database.

Considering the synchronization process, in an environment without a policy, the user has to decide when the database is synchronized. He may not know the number or kind of the tables it has, so normally he will synchronize the entire database. However, using a well defined policy, the user does not require knowing anything about the database; he manipulates the clients and orders in the way the developer has designed and forgets about when or what to synchronize.

Looking at the definition of the policy we can see that the Orders table only defines a time for changes made. This policy can be read as: “When the table Orders has changed, it should be synchronized with the server in 5 minutes. If the change is an insertion (a new order has been placed) only wait 1 minute.

In other words, this table will not synchronize unless it has been modified. Also we prioritize by type of change. In this case, an insertion has less time because that process is considered more important, maybe because it is a shared table and the information being updated is vital.

In the case of the service invocation, due to the requirements of the application, a Web Service must be invoked once an hour. Without a connection policy, the developer must control the time and invocation attempts. This may lead to an ineffective way of dealing with this requirement. Defining the responseTime in 1 hour and invoking the service through the framework will have the same effect and frees the developer from worrying about the underlying process.

The main benefits of this approach are:

- Due to the management of the connection based on how the applications behave, resources (e.g. battery and connection) are saved because the device only connects when it really needs to.
- The cost of the communication can also be improved by saving network transmissions and amount of data.
- Using the policies as an underlying layer, the developer is completely agnostic of the communication processes that are carried out by the system. He/she no longer worries about whether the connection is available or not and the experience of the user would be the same.

To summarize, using an application-aware decision mechanism based on these policies eases the use of data synchronization and service invocation mechanisms, saving energy and leveraging the possibilities of mobile applications with no additional effort.

### 3. Dynamic Power Management

Dynamic power management [10] reduces the power consumption of complex electronic systems by trading off performance for power in a controlled fashion, taking system workload into account. DPM encompasses a set of techniques that achieves energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle. In a power-managed system it is possible to set components into different states, each characterized by performance and power consumption levels. The main responsibility of a power management policy is to decide when to perform component state transitions and which transition should be performed, depending on system history, workload, and performance constraints.

In this work, we apply DPM techniques to the wireless network card. Most of them can operate in a low power mode called idle mode, which consumes a fraction of the power used when transmitting or receiving.

However, changing between power states has an overhead, as Figure 1 shows. If there were no overhead, power management would be trivial; just turn off the device whenever it is idle. Unfortunately, we cannot ignore this delay and/or energy consumption. Consequently, a device should only sleep if the saved energy justifies this overhead. The rules that determine whether to shut down a device are called policies.

<table>
<thead>
<tr>
<th>Request Time</th>
<th>Response Time</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not required</td>
<td>Not required</td>
<td>Services must be invoked but there are no time restrictions. They are processed when the system connects for any other reason. Example: a weekly report of a salesman.</td>
</tr>
<tr>
<td>Not required</td>
<td>Specific time</td>
<td>The important aspect is when the response is received; the moment of the invocation is delegated to the system. In this case is important to know that the requestTime is not defined by the developer but in fact it must not be greater than the responseTime minus the serviceTime. Example: Everyday by the end of a work shift (6:00 pm), a sales report must be sent.</td>
</tr>
<tr>
<td>Specific Time</td>
<td>Not required</td>
<td>This covers the services that can be delayed a specific time without any damage to the invocation itself. Example: sending a SMS at a specific time.</td>
</tr>
<tr>
<td>Specific Time</td>
<td>Specific Time</td>
<td>These services are constraint by both times, they must be invoked at a specific time and the response is needed by another time. As in previous cases, the request time must be less than the response time minus the time the service takes.</td>
</tr>
</tbody>
</table>

### 2.2 Functional Comparison

In order to clarify how this mechanism works we present the following example (based on the policy shown in Table 1) and provide a comparison between the operation of an application with and without the proposed approach.

An imaginary application for a mobile device has a local database of clients and orders placed by them. This database synchronizes with the enterprise database. Furthermore, the application invokes a Web Service that returns the state of the stock market every work shift (6:00 pm), a sales report must be sent. The cost of the communication can also be improved by saving network transmissions and amount of data. Using the policies as an underlying layer, the developer is completely agnostic of the communication processes that are carried out by the system. He/she no longer worries about whether the connection is available or not and the experience of the user would be the same.

To summarize, using an application-aware decision mechanism based on these policies eases the use of data synchronization and service invocation mechanisms, saving energy and leveraging the possibilities of mobile applications with no additional effort.
Bearing this in mind, we have defined a parameter called $idleTime$, which is the minimum period of time for the system to be idle in which we can guarantee that a power saving will happen, because the overall energy consumed in the idle state is less. Figure 1 shows the energy consumed with and without a DPM technique. We can calculate the minimum acceptable value for the $idleTime$ by equating the energy consumed in both cases. The result is shown in equation (1).

$$T_{idle} = \frac{P_{w,s} T_{w,s} + P_{s,w} T_{s,w} - P_s (T_{w,s} + T_{s,w})}{P_w - P_s}$$  \hspace{1cm} (1)$$

where:

- $P_w$: average power consumed in working state.
- $P_s$: average power consumed in sleep state.
- $T_{w,s}$: time spent to go from working to sleep state.
- $T_{s,w}$: time spent to go from sleep to working state.
- $P_{w,s}$: power consumed to go from working to sleep state.
- $P_{s,w}$: power consumed to go from sleep to working state.

If an idle period is going to be longer than the $idleTime$, the energy saving is guaranteed. Nevertheless, the problem is how to predict the length of an idle period before it starts. First we use the needs of the application, modelled with ANE SML to calculate when the next connection should be established. This time has been called communication time ($T_{com}$). Taking the information provided by the policy as a reference we can establish the times in which a connection to a server should be made, either to synchronize the database (or some of its tables) or to invoke a service.

In order to achieve the greatest energy savings, we have defined a novel technique, which is called CIST (Consumption Improvement by Stuffing Time). Our approach will try to provoke a change to the idle state whenever it is possible. Applying the policy, we will have a set of connection requests to synchronize a database or to invoke a service and their time restrictions. As a consequence, we will try to send as many pending requests as possible when a new connection has to be made, in order to minimize the network connections. To decide which pending requests are to be processed in advance we have defined a time span which is added to $T_{com}$ to indicate which requests are going to be sent. This time is called stuffing time ($T_{stuff}$).

When the pending requests are processed, not only will those which have $T_{com}$ as their restriction be sent but also those that fall into the $T_{com} + T_{stuff}$ period. This way the idle period would be long enough for the change from working to sleep state to be worth it. Consequently, stuffing time should be greater than $T_{idle} + T_{proc}$, where $T_{proc}$ is the processing time, representing the estimated duration of the sending/receiving of the waiting requests, so that a change of state is carried out.

Unfortunately, $T_{proc}$ depends on several variables (concrete device characteristics, type of operation, number of requests to process, etc.), which are unknown in advance. So we need a mechanism to estimate this value. For this purpose, we are using the exponential average approach. This technique predicts the next $T_{proc}$ calculating the accumulative average of the previous $T_{proc}$. The recursive prediction formula is shown in (2).

$$T_{proc}[n + 1] = (1 - a)T_{proc}[n] + a \sum_{i=0}^{n} (1 - a)^i T_{proc}[n - i]$$  \hspace{1cm} (2)$$

This approach is based on the premise that the recent future is likely to be similar to the recent past, giving more weight to previous values that are closer to the one being calculated. The parameter $a$ is a way of controlling how much the near past affects the predictions.

4. Framework Architecture

Based on the previous idea, we have designed a framework where our approach is applied. The framework follows client-server architecture, but here we will focus on the client side. An overview of the architecture of this side is shown in Figure 3.

Figure 3. Overview of the framework (client side).

The Policy Management module is responsible for including in the framework all the benefits that the application-aware policies can provide. As the application connectivity needs are set in the connection policy, data or service invocations can be performed even when the device is offline. Based on the policies and network availability, the device is able to decide when to connect and if it is not possible, it caches the requests and waits till the network is available again. When the connection is lost, some of the policies might not be correctly applied, that is, a delay might be introduced, because the device has to wait for the network connection to be re-established. Regardless the situation, the framework is able to adapt to the connectivity status transparently to the user.

4.1 Policy Management Module

This module is responsible for the application aware policies. It will decide when to activate a communication process (either synchronization, service invocation, or both) based on the information gathered from the rest of the components (see Figure 4):
• **Battery Manager** monitors the battery level. The connection policies determine when the network connections are going to be done, but monitoring the battery level we could apply a correction factor, increasing the time to wait between two communication processes. When the battery of the device is low, we can wait and additional time to launch non critical pending processes.

• **Connection Manager** is responsible for activating or deactivating the wireless interface. It knows how to put the interface to sleep and how to put it again to the working state when needed.

• **XML-Connection Policy** is the policy explained before, which defines the information related to the database and the services used by the application.

• **Decision Engine** is the component that collects the information needed to decide when to launch a pending process (synchronization or service invocation) and when to change the state of the resources based on our the proposed technique.

• **Application Operation** is not exactly a component; it represents the actions that the mobile application performs, which might have influence on the decision process (e.g. a table in the database that has been changed, added or a service that has been invoked).

Combining these modules a developer can add new features to any mobile application. The use of application aware policies frees the final user from the burden of worrying about the connection or battery state (except to recharge it when it runs out).

Besides, considering how databases and services are invoked in mobile devices (usually invoking some data access API or a service invocation proxies), these calls could be easily decorated with a wrapper that interacts with the framework accordingly. The result is a mobile application using the framework would be programmed the same way as a non-framework application, because the APIs are the same. The framework would act as a middleware, managing the policies, which should have been previously defined by the developer.

The **Connection Policy** is based on the needs of one application but it would also be possible to define a default policy just in case the application does not have a policy previously defined. This default policy would be more generic, yet useful in some cases.

---

**Figure 4. Components of the Policy Management Module.**

Combining these modules a developer can add new features to any mobile application. The use of application aware policies frees the final user from the burden of worrying about the connection or battery state (except to recharge it when it runs out).

Besides, considering how databases and services are invoked in mobile devices (usually invoking some data access API or a service invocation proxies), these calls could be easily decorated with a wrapper that interacts with the framework accordingly. The result is a mobile application using the framework would be programmed the same way as a non-framework application, because the APIs are the same. The framework would act as a middleware, managing the policies, which should have been previously defined by the developer.

The **Connection Policy** is based on the needs of one application but it would also be possible to define a default policy just in case the application does not have a policy previously defined. This default policy would be more generic, yet useful in some cases.

---

**Figure 5. Process of Policy Management Module.**

A general and simplified operation of the whole framework would be as follows:

1. Based on the application domain, the policies for the different databases and available services are defined.
2. The mobile application is developed and linked to the framework.
3. The application is started in the mobile device and the Policy Management Module is activated. The operation of this module is depicted in Figure 5. Basically it will:
   - Calculate, based on the policies and the activity of the application, when the connection should be made, trying to schedule a connection when the interface could be changed to sleep mode afterwards (using CIST).
   - When this time expires, it will try to connect to the corresponding servers (synchronization or service invocation) and send the pending requests.
   - Change the interface to sleep and recalculate the next time the connection should be made, starting the process again.

This procedure has variations when the battery level is low or when another action is performed in the application which might override the current policy (for example an insertion with higher priority is done). In all these cases, the policy is readjusted transparently to the user.

---

5. **EVALUATION OF THE FRAMEWORK**

In this section, we present how we have tested the viability of our approach, whose main objective is to increase the energy efficiency. The prototype of the framework was developed in C#.
for the platform .NET Compact Framework. We first discuss the experimental setup and then present the experimental results.

5.1 Experimental set-up

![Figure 6. Experimental setup.](image)

We conducted our experiments on a PDA, using a test set-up as shown in Figure 6. All our experiments were made for an i-mate PDA 2K GSM/GPRS Pocket PC, with a 400 MHz Intel PxA263 processor. The PDA used a Texas Instruments wireless network interface card for communication. We connected the PDA to a 6.5V power supply with an oscilloscope measuring the voltage across a 1 Ohm resistor. We calculated the instantaneous and average power consumption of the PDA using formula (3):

$$P_{pulse} = V_{pulse} \frac{V}{R}$$  

(3)

Formula (3) is just Ohm’s law to calculate the power consumed by the PDA, knowing that a 1 Ohm resistor has been introduced in the circuit.

5.2 Experimental results

To evaluate our framework, we designed several scenarios. The selected application was a system to manage technical problems in the maintenance of subway facilities, such as elevator malfunctions, light problems etc. Every technician has a mobile device with an application which had a small database with the assigned tasks and their state and also the ability to reserve via a Web Service certain materials in the warehouse.

For the validation of each scenario we carried out three tests:

1. Running the scenario without our framework, that is, considering that the wireless interface is always on working state and that the different requests are processed when the user requests, because no connection policy is defined.

2. Running the scenario with our framework applying DPM techniques but not using CIST technique, only processing the requests whose time is in the limits of T\text{com} (ignoring the stuffing time defined in the CIST technique).

3. Running the scenario with our framework applying DPM techniques and CIST technique. Now the stuffing time will be considered and requests whose processing time is in T\text{com} + T\text{dur} will also be processed.

We implemented 5 different scenarios where different situations were represented. In each scenario a user carries out several activities on a PDA including operations on a mobile database and invocations to services. The scenarios differ from one another in the activities that were carried out and the possibility of making the network connection at the desired moments. We forced the system not to have network connectivity in certain tests to see how it behaved in such situations. A brief description of each scenarios is presented next:

- **Scenario 1**: is analysed in section 5.2.1.
- **Scenario 2**: Two services are invoked, being the second one more restrictive than the first. When applying the CIST technique both are processed, instead of one by one.
- **Scenario 3**: One database action is performed and three invocations to services. When applying the CIST technique they are rescheduled and send in pairs, saving two connections. When the second group is sent, the response of the other invocation is received.
- **Scenario 4**: In this scenario just a database update and an invocation to a service are performed, but there is no wireless connectivity available during the first half of the test, so the framework forces the wireless to the sleep state. This scenario is used to test that the framework also works when there are temporal connectivity problems. Test 1 in this scenario does not apply, because, the way it has been defined it cannot be carried out without wireless connectivity.
- **Scenario 5**: this scenario was modelled with a default synchronization policy to demonstrate how default policies work. One database update and one service invocation are carried out and two automatic connections are scheduled due to the CIST technique.

5.2.1 Scenario 1 analysis

To show how the experiments were performed we now analyse Scenario 1 in detail. The rest of the scenarios followed the same schema.

Table 3 shows a comparison of the activities carried out in each test. In Figure 7 we can see how much energy each of these activities consume (marked with an inverted triangle) and the effects of applying the CIST technique.

Test 1 presents the most energy consumption rate as expected, because the wireless interface is always on. In Test 2 we see that applying the connection policies the wireless interface is most of the time in the sleep state, but still both operations on the database are synchronized independently (5 and 6) because their time requirements are different. In Test 3, applying the whole CIST mechanism, only one connection is done to synchronize the database. In this concrete scenario, the service invocation and the synchronization do not overlap, because the time requirements exceed the T\text{dur} so they are not processed at the same time.

Figure 8 shows the average power consumption in each test in different scenarios and the percentage of energy saved.

The results show that our framework reduces power consumption in all the scenarios and that the CIST technique increases the energy savings even more. Depending on the conditions of the scenario the framework manages to save up to 40% of energy, which proves that the technique presented in this paper is a viable alternative for designing mobile applications.
Table 3. Steps of the different tests of scenario 1.

<table>
<thead>
<tr>
<th>Test</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Turn on the application</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Read a task from the database</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Insert a new task in the database</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Synchronize the new task</td>
<td>Insert a note in the database.</td>
</tr>
<tr>
<td>5</td>
<td>Insert a note in the database.</td>
<td>Synchronize the new task in the database.</td>
</tr>
<tr>
<td>6</td>
<td>Synchronize the new note</td>
<td>Synchronize the new note</td>
</tr>
<tr>
<td>7</td>
<td>Reserve material in the warehouse (service request)</td>
<td>Reserve material in the warehouse (service request)</td>
</tr>
<tr>
<td>8</td>
<td>Invoke the service</td>
<td>Invoke the service</td>
</tr>
<tr>
<td>9</td>
<td>Receive the response (availability of the material)</td>
<td>Receive the response (availability of the material)</td>
</tr>
<tr>
<td>10</td>
<td>Turn off the application</td>
<td>Turn off the application</td>
</tr>
</tbody>
</table>

6. RELATED WORK

In today’s world of mobile communications, one of the most precious resources is power [4]. The mobile device can operate only as long as battery maintains power. Traditionally, energy management has been seen as an issue to be solved within individual hardware devices (e.g. with circuit design).

The next step was to enable dynamic power management [10], which allowed to control and reduce power consumption at runtime. Many techniques and algorithms have been suggested but they are focused mainly on the hardware and operating system levels.

There have been strong efforts to improve energy awareness in hardware devices [4]. CPU voltage scaling [2] [5], disk spin down policies [6], energy-aware networking [17] among others, have been the focus of these researches.

In spite of the improvements achieved in low-power hardware design and battery life, there is a new growing awareness claiming that energy management techniques should include higher levels of the system. The reason is that the high-level applications are the ones that know best what their needs are. This information is essential to improve the energy management [4], [7], [8], [9].

Next we present related work of operating system, middleware and application levels to show where our approach is located. Inside each level we have focused on those works that cover the energy-aware networking approach, especially those centred on Wireless networks.

At the operating system level, the goal is to try to reduce energy consumption by unifying resource management, introducing the concept of accounting and enabling control of energy consumption and battery lifetime.

Our idea belongs to the energy-aware networking approach. The reason why we have selected this area is that there is a growing trend in mobile computing towards an increase of communication-dependent activities. These activities, including sending and receiving data, are prone to consume a lot of energy [16].
Scenario 1 - Test 1

Scenario 1 - Test 2

Scenario 1 - Test 3

Figure 7. Measurements of scenario 1 in each test.
Some approaches are FIS [11], ECOSystem [7], Odyssey [4], and STPM [17]. One problem is that these techniques require the modification of the Operating System. Moreover, they have mainly been tested only on laptops.

As it was proposed in ECOSystem, future work should consider the interaction between OS and the applications to improve its management. Following this idea we also want our approach to be applied on any platform and with no changes to the underlying operating system.

STPM focuses on wireless network power management. It adapts to the characteristics of the network interface, mobile computer and applications. It was implemented as a Linux Kernel module, providing a simple interface that allowed applications to disclose hints about their intention of using the network interface. STPM then adapts its power management strategy to the observed network access patterns.

In our proposal the system does not need to learn about the network access patterns. Using simple policies previously defined, the system can decide when to activate or deactivate the network interface.

At the middleware level, the objective is to reduce energy consumption without modifying the OS. There are several approaches which can provide adaptation as a common service or as a domain-specific service. Puppeteer [12], PAGODA [15] and PAWP [7] are domain-specific middleware whereas Spectra [13] and PARM [14] are common services middleware.

Of these concepts, we have adopted a hybrid solution, an extensible common service with domain-specific policies. This way we achieve a more effective adaptation.

Finally, at the highest level, the application can participate in the power management.

There are different levels of participation depending on the role the application assumes in the power management process [4].

- **Application transparent approach** the system manages the adaptation without the participation of the application (e.g. Puppeteer, PAGODA y PARM, PAWP and Chandra).

- **In laissez-faire approach** the power management is responsibility of the application. The system just helps providing interfaces for monitoring and making statistics. This approach is the least used.

- **In application aware approach** the system and the application collaborate to reach a compromise between the two previous schemes (e.g. Odyssey and STPM). This usually requires source code modifications. The applications can cooperate with the system in different ways. For example, the applications can inform about their demands and the system on the basis of these needs can schedule the power management [8], [17]. In other situations, applications are rewritten so that they can dynamically modify their behaviour according to changing restrictions in energy consumption (Odyssey).

Our approach merges two of these philosophies. It can be application transparent because the system can manage the power consumption without the applications being involved (using a generic connection policy) and it can also be application aware, because if they want, applications can specify their needs either at runtime or with the previous definition of connection policies.

7. **CONCLUSIONS & FUTURE WORK**

In this paper, we have presented an application-aware framework that adapts its behaviour to the needs of the applications. The framework uses Dynamic Power Management techniques focused on the wireless network card to control and reduce power consumption at runtime. We have also defined a technique that increases the energy savings called CIST.

Finally, all the above techniques were integrated into a prototype system and power consumption was measured. The results show that our approach improves energy conservation compared to other management strategies.

We are aware that in the future, enterprise applications, the primary target of this research, would need something more than simply monitoring the wireless card, even though today it remains as the most energy consuming part of a mobile device. We plan to explore other power management schemes such as frequency and voltage scaling, middleware adaptations for improving the power consumption of displays, storage devices, etc. and integrating them into the framework.

8. **REFERENCES**


