GreenMobile - interest awareness for mobile energy saving
Filipe Alexandre Barroso
filipe.barroso@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
November 2014

Abstract
Energy-efficiency is a fundamental consideration for mobile devices. As performance expectations are increasing towards the levels of personal computer, power efficiency is becoming a major bottleneck. However, battery technology hasn’t been improving at the same speed as other components. The goal of this project is to develop a system that propagates a huge number of context-oriented messages while it minimizes the energy consumed by mobile devices. This document presents an overview of the greenMobile architecture and implementation. The greenMobile system delays the message transmission and uses techniques such as aggregation and compression to improve the network performance. This system is composed of a component on the mobile clients and a server. To demonstrate the results of our project, we developed an application using the greenMobile system. The results show that we have successfully developed a scalable system that efficiently propagates context messages.

Keywords: context-awareness, scalability, performance, portability, and energy-efficiency.

1. Introduction
In recent years, mobile devices have emerged as a dominant computing platform for users. These devices have been improving dramatically. Throughout time, central processing units, memory, displays, and other components become more reliable, better, faster, and cheaper to manufacture. However, battery technology hasn’t been improving at the same speed. In this case, we don’t see the exponential increase in performance we see with other types of technology. To further worsen the situation, the energy consumed by these new components, that are part of the mobile devices, has increased substantially. Mobile phone usage is developing towards high speed wireless communication, always online connectivity, high definition multimedia, and resource heavy computation [1].

Consider a scenario in which Joe is walking in a crowded urban environment. As he walks, he is constantly interacting with his mobile device through a mobile application that works in crowd sensing environment. During his walk, Joe came across an event that attracted the attention of a big crowd. It was a live performance of a famous local band. Just like Joe, a huge number of users decided to share their experience with the surrounding people, using the mentioned application.

This type of application, besides taking advantages of the mobile phone’s capabilities, just like gathering information from its sensors, also generates a huge number of messages to be exchanged through the Internet. The quantity of such information is proportional to the amount of people that are using the application, making its usage heavily expensive in terms of battery consumption.

Joe then notices that he forgot to charge his mobile phone, having only a small percentage of his battery left. He wants to take advantage of the application for a modicum amount of time, making it very important for the application to be as much efficient as possible.

The goal of this project is to develop a system, called greenMobile, that propagates a huge number of context-oriented messages while it minimizes the energy consumed by mobile devices.

The requirements for the system are the following: (1) scalability, (2) performance/user experience, (3) battery efficiency, and (4) portability. We detail each one in the next paragraphs:

1. Scalability - ability of the system to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [2]. It allows the system to adapt better to the differences in the number of exchanged messages without the need of re-engineering or duplication of the system;

2. Performance/user experience – computer performance characterizes how well the computer does the work it’s supposed to do [3]. We expect our system to have a good performance. If the performance is not the expected, the user experience will suffer from it;

3. Battery efficiency - the consumption of battery comes from a very diverse number of sources, being the wireless communication subsystem one that accounts for a major component of the total power consumption [4]. Wireless network access is a fundamental feature for mobile devices, but if not optimized for power consumption this component can have a significant impact in the battery lifetime;

4. Portability - a software unit is portable across a class of environments to the degree that the
cost to transport and adapt it to a new environment in the class is less than equal to the cost of redevelopment [5]. The cost of porting an application developed with greenMobile system to a new environment should be lower than the cost of redeveloping such application.

In order to fulfill the mentioned requirements, one must deal with the following challenges:

1. Reduce the amount of data to be exchanged through compact forms of representation;
2. Reduce the number of times the system exchanges messages;
3. Ensure that the system’s usability is not affected while improving the battery efficiency;
4. Schedule messages to be sent according to their priority;
5. Achieve a high level of portability.

To the best of our knowledge, at this time, there is no system that matches the goals and requirements to which we proposed ourselves to reach. However, there is a lot of work that focuses on particular aspects that are part of our interest.

A solution based on the publish-subscribe communication paradigm [6] is well adapted to the loosely coupled form of distributed interaction present in large scale settings. The scalability property present in this style of communication makes it a viable option for our system.

Application-layer multicast solutions [7, 8] produce distribution structures like trees that have a good steady rate performance. However, these protocols face problems in the case of node failure, as the only solution to fix the problem might be to rebuild the multicast tree from scratch, which is an expensive process.

We also studied some solutions based on gossip protocols [9, 8], which emerged as a viable strategy to implement primitives highly scalable and resilient. The operation model of the protocol provides a high level of fault tolerance, as its intrinsic redundancy is able to mask the failures that might occur in either the network or the nodes. However, these characteristics come at the price of performance.

In general, the problem with the previously mentioned solutions is the lack of adaptability that is necessary to cope with a highly dynamic environment. These systems suffer from the drawback that they assume that changes in the interests are done infrequently, so the process of setting them up is not very efficient [10]. Also, both gossip protocols and publish-subscribe systems are designed to either propagate messages or discard them, even though the immediacy may not be required.

The other solutions analyzed are called context-aware systems. These systems [11] use context to provide services and/or information to the user with relevancy depending on the user’s task. Context-aware systems address some of the previously mentioned problems. These systems are particularly interesting regarding the usage of the technique aggregation. Aggregation is defined in context-aware literature as the combination of information from multiple sensors, along with conflict resolution [11]. This process can occur at the inference level or at the distribution level.

To achieve the proposed goal, the greenMobile system delays the message transmission and applies techniques such as aggregation and compression. The immediacy of the information isn’t always required, thus the system can delay the message propagation and subsequently improve the system’s scalability. The techniques aggregation and compression allow the reduction of the amount of information needed to represent a context message, effectively reducing the bandwidth consumption. This improvement is specially important in the achievement of the greenMobile goal because the wireless network communication subsystem accounts for a major component of the total power consumption.

With this work, we make the following contributions:

1. Development of the greenMobile system. This system improves the scalability by substantially reducing the amount of information transmitted. The energy consumption of the application is also substantially reduced. The system is also able to adapt to the user expectations;
2. Development of an application on top of the greenMobile system, called TestApp. The objective of this application is to perform the evaluation of the greenMobile system;
3. Evaluation of the greenMobile system regarding the proposed requirements: scalability, performance/user interaction, and portability.

In the remainder of the paper, we start by presenting the related work. In Section 3, we describe the architecture of the greenMobile system. Section 4 describes the implementation details of the proposed system, and in Section 5 we evaluate the greenMobile system and discuss the results. Finally, we draw some conclusions.

2. Related
In this section, we review the literature regarding publish-subscribe, application-layer multicast, gossip protocols, context-aware systems, and mobile energy saving.

2.1. Publish-subscribe
A solution based on the publish-subscribe communication paradigm [6] is well adapted to the loosely coupled form of distributed interaction present in
large scale settings. The scalability property present in this style of communication makes it a viable option for our system. ISIS [12, 13] is one of the earliest publish-subscribe systems developed and uses a subject-based approach. Siena [14] is a content-based publish-subscribe system and was designed and implemented with the intention of maximizing expressiveness and scalability. XMessages [12, 15] is a system that performs the filtering of messages based on the subject and the content.

In the context of our project, the usage of a publish-subscribe system raises some issues. In publish-subscribe systems, messages are delivered as soon as possible, even though the immediacy may not be required. The participants are completely oblivious of the presence of other subscribers and have no information regarding the activities in other channels. This makes it impossible for any visualization of context-aware information [16]. These systems also assume that the interest of the subscribers are fixed or that it changes too infrequently, therefore changes are too expensive. In the system we are developing, it’s expected to change very frequently, hurting the system’s overall performance.

2.2. Application-layer multicast
Application-layer multicast solutions [7, 8] produce distribution structures like trees that have a good steady rate performance. However, these protocols face problems in the case of node failure, as the only solution to fix the problem might be to rebuild the multicast tree from scratch, which is an expensive process. Bayeux [17, 9] is an example of an application-layer multicast protocol that employs a source-based strategy. This protocol scales to arbitrary large receiver groups while tolerating failures in routers and network links. Scribe, just like Bayeux, is a scalable application-layer multicast infrastructure. Scribe is a fault tolerant system and provides a mechanism to handle root failures. However, it only ensures best-effort delivery of multicast messages and doesn’t specify a delivery order.

Just like publish-subscribe systems, application-layer multicast approaches also have the problem that they assume that the interests of the peers change very infrequently, leading to poor performance when they do [10].

2.3. Gossip protocols
We also studied some solutions based on gossip protocols [9, 8], which emerged as a viable strategy to implement primitives highly scalable and resilient. The operation model of the protocol provides a high level of fault tolerance, as its intrinsic redundancy is able to mask the failures that might occur in either the network or the nodes. However, these characteristics come at the price of performance. NeEM [8, 19] is an epidemic protocol whose main goal is to ensure that it doesn’t contribute to the congestion of the network during overload periods. CREW [8, 20] is designed to perform fast dissemination of content over heterogeneous networks and unpredictable conditions.

In general, the problem with the previously mentioned solutions is the lack of adaptability that is necessary to cope with a highly dynamic environment. These systems suffer from the drawback that they assume that changes in the interests are done infrequently, so the process of setting them up is not very efficient [10]. Gossip protocols are also designed to either propagate messages or discard them, even though the immediacy might not be required.

2.4. Context-aware systems
The other solutions analyzed are called context-aware systems [11]. These systems use context to provide services and/or information to the user with relevancy depending on the user’s task. The Context toolkit [21] and CenceMe [22] are context-aware systems that use the technique aggregation to improve the efficiency of the system. This technique is defined in context-aware literature as the combination of information from multiple sensors, along with conflict resolution [11]. The system Solar [23] doesn’t perform message retention so the aggregation is not very efficient. Tickertape [24] is a context-aware framework that successfully uses a publish-subscribe approach to propagate context messages. ReconMUC [25] is a multi-user chat application that uses an adaptable consistency model. This system propagates the important messages immediately and delays the not so important. Radiator [26, 10] was designed to assist programmers in the implementation of efficient context propagation mechanisms on their applications. This system improves the scalability and privacy of an application. ReconMUC and Radiator also use the technique aggregation.

2.5. Mobile energy saving
The studied systems/approaches perform improvements at the operating system level that involve strategies like caching, improving communication efficiency, scheduling, and energy management [27]. TailEnder [28] is an energy-efficient protocol for scheduling data transfers. CoolSpots [4] is a system that enables wireless mobile devices to automatically switch between multiple radio interfaces in order to increase battery lifetime. Mazliza Othman and Stephen Hailes [29] proposed a different approach of power conservation based on the concept of load sharing. Philippe Baptiste [30] presented a theoretical work that focuses on the shut-down mechanisms that puts a system into a sleep state when it’s idle. Eprof [31] is an energy profiler for smartphone apps that analyzes the energy consumption of mobile applications and provides advice on
how to improve it.

3. Architecture
The greenMobile system delays the message transmission and uses techniques such as aggregation and compression to improve the network performance. This improvement is specially important in the achievement of the greenMobile goal because the wireless network communication subsystem accounts for a major component of the total power consumption. The idea behind the development of the greenMobile system was not to develop from scratch but instead to pick a system and improve it with the desired functionality and requirements in mind. Radiator [26, 10] was the system chosen to perform this task.

The greenMobile system is based on a client-server architecture and is composed of a component on the mobile devices and the greenMobile server. The clients send messages to the server when they are interested in sharing context with the other clients. The server works as an intermediary between the multiple mobile clients, being responsible for the delivery of the context information. The described communication model is shown in Figure 1.

The greenMobile server is composed of four different modules, each of them with different responsibilities. The client management module is responsible for managing the information regarding the clients that are registered in the server. This includes not only information about how to reach them but also the data whose transmission has been delayed by the communicator module. The aggregation and compression module are responsible for applying the techniques aggregation and compression, respectively. Finally, the communicator module is responsible for communicating with the clients.

Figure 2: Client’s architecture.

Figure 2 presents the client’s architecture. The greenMobile system is designed to work as a software layer that lies between the operating system and the application (i.e. middleware). Application developers build their own applications on top of greenMobile to take advantage of its capabilities, confident that the best decisions are made. The system is developed for mobile clients that run the Android operating system. As the name suggests, the modules that compose the client’s middleware have similar responsibilities to the ones that compose the server.

3.1. Communication
The server follows a push-based approach, meaning that it’s responsible for delivering the information to the clients. This type of approach is harder to implement but provides several improvements. Clients do not always require the immediacy in the propagation of messages, thus the system can delay the propagation and subsequently improve the system’s scalability. The system must then decide when to push the information and exactly what to push.

The server transmits the information to the clients in two different manners: i) it can either propagate the information to all the clients or ii) selectively propagate the information to only a subset. Usually, in context-aware systems the information is propagated to all the clients. The devices that are not interested in the message simply ignore it. However, if only a subset of the mobile clients are interested in the context, it’s not very efficient to transmit it too the whole set. Having this in mind, we designed a mechanism that only transmits the context to the clients that are related to it.

3.2. Aggregation
The greenMobile system inherits the aggregation capacity from Radiator. The aggregation can occur at the inference level or at the distribution level. Aggregation at the inference level or semantic aggregation it’s used to provide the application developers with data on a higher level of abstraction. It works by combining data together into more rich-level information. Aggregation performed at the distribution level or syntactic aggregation consists in combining small and multiple messages into a single one and possibly compressing it.

Initially, Radiator only allowed the specification of the aggregation mechanism to work on the server. GreenMobile goes a step further and also allows the specification on the client side. The aggregation mechanisms are completely independent of each other and can possibly use completely different configurations.

The greenMobile system also adds another improvement regarding the way the aggregation functions are used. The Radiator system uses the aggregation functions to concatenate one message to the set of messages that were previously aggregated.
In the greenMobile system, the middleware begins by concatenating all the messages, and only then it applies the aggregation function. With this improvement, the aggregation functions become more powerful, easier to develop, and more efficient.

For a better understanding regarding the aggregation mechanism, let’s imagine a scenario where a client is interested in constantly sharing his current location. This interaction is performed using an application built on top of the greenMobile system that periodically sends a message, containing two locations. Figure 3 shows the contents of a send queue containing the messages ready to be processed.

![Figure 3: Example of a send queue.](image)

Figure 3: Example of a send queue.

To achieve this result, the aggregation was setup to aggregate five messages and no aggregation function was specified, meaning only a simple concatenation was performed. As we can see, with the application of this technique we effectively send less information. The system performs less messages transmissions, resulting in the decrease of structural information and headers.

![Figure 4: Result after applying syntactic aggregation.](image)

Figure 4: Result after applying syntactic aggregation.

The results can be further improved by applying semantic aggregation. Let’s imagine that in this scenario the application it’s only interested in finding the smallest square where all the locations can be found. Figure 5 presents the result of applying semantic aggregation to the contents of a send queue. The aggregation was setup once again to aggregate five messages, but this time it was configured with an aggregation function that calculates the interesting information. As we can see, the application of this technique further reduces the information to be transmitted, thus improving the benefits of syntactic aggregation.

![Figure 5: Result after applying semantic aggregation.](image)

Figure 5: Result after applying semantic aggregation.

### 3.3. Compression

The compression module allows application developers to optionally implement a system that compresses all the messages between the clients and the server. This mechanism is completely independent from the aggregation mechanism.

The objective of using compression is to further reduce the amount of information to be transmitted. The idea is to provide application developers with some alternatives that show the best results in the context of our project. After a brief study, we concluded that the **BZip2** algorithm was the most suited, although algorithms such as **Gzip** and **DEFLATE** also proved to be promising alternatives. Application developers are also able to implement and extend the system with their own algorithms.

![Figure 6: Result after compression.](image)

Figure 6: Result after compression.

The code that implements the greenMobile system is written in Python and can be accessed from [https://bitbucket.org/goku236/greenmobile](https://bitbucket.org/goku236/greenmobile).

The developer of the Radiator model provided a reference implementation in the Python language to aid developers that wished to use it. The applications developed for the Android operating system are usually programmed in Java using the Android Development Kit. Since the Radiator model is implemented in Python, a solution needed to be found to solve this problem.

After a brief analysis, we reached the conclusion that Python for Android is the most suited project that allows us to create Android applications in Python. This project also allows the usage of GUI libraries to design the interface of applications. The libraries need to be compiled for Android. Also, since Python doesn’t natively allow for the creation of a user interface, a toolkit like Kivy is needed.

The developers describe Kivy as an: "open source Python library for rapid development of applications that makes use of innovative user interfaces, such as multi-touch apps". The code developed with Kivy runs untouched on Android, iOS, Linux, OS X, and Windows. This fact greatly increases the portability of the greenMobile system, being this one of the system’s requirements.

For an application developer that wishes to utilize this system in the development of his application, there are two components that he needs to understand: i) the middleware and the ii) server.

---

4.1. Middleware
The middleware component runs along with the client application and abstracts the details regarding context propagation, aggregation, and compression.

The application developer only needs to understand the class Middleware. The instance of this class, after receiving the proper configuration, is able to communicate with the server to propagate messages to a subset or possibly all the clients. If the developer wishes to extend or modify the client functionality, this class should be the one modified.

```python
aggregability = set_type('volume', 50)
aggregation_function = {'Loc': minimum_bounding_box}
compression = compressor.Compressor
middleware_engine = Middleware('Joe', aggregability, aggregation_function, compression)
ctx = context.Context(set(['Joe']), 1, {'Loc': [(-63, -134), (29, 119)]})
middleware_engine.propagate_context(ctx)
```

Listing 1: Example of greenMobile usage.

Listing 1 describes a usage of the middleware component. First, the application begins by configuring the aggregability in line 1. In this example, the aggregation mechanism is configured to aggregate 50 messages before transmitting the information. This is done using the function set_type. In line 2, the aggregation function is set to aggregate the locations by finding the smallest square where all the locations can be found. This is defined with the assist of the function minimum_bounding_box. In line 3, the compression is set to be turned on, using the algorithm Bzip2. In line 4, the application is registering the user Joe with the server. The configuration is also provided to the middleware component. At this point, the client may begin to communicate with the server by exchanging context instances. In line 5, the application creates a context that specifies that Joe, at the time instance 1, is located in the square defined with the provided coordinates. Finally, in line 6 the application propagates the message.

4.2. Server
The server is developed to run on a fixed computer, although it can possibly run in a mobile device. It can be configured using a configuration file, allowing for the specification of the aggregation and compression functions to be used, if any at all. If the developer is interested in adding some extensions to the behavior of the server, the class ContextPropagator and the file server.py contain the code that needs to be changed.

GreenMobile uses HTTP to communicate between the server and clients. This protocol is widely used and works really well with firewalls. Also, it doesn't require a permanent connection between the clients and the server. The communication is implemented using Bottle in conjunction with Paste. Bottle is a fast, simple, and lightweight WSGI micro web-framework. It's implemented with a built-in HTTP development server and support for multiple WSGI capable HTTP servers, such as Paste.

4.3. Compression
The HTTP protocol supports compression. Unfortunately, this approach only works for responses. In most cases, this wouldn't constitute a problem since a big portion of the HTTP lies in the responses. However, since the greenMobile system follows a push-based approach the responses are very short. Also, the compression schemes are limited to the ones available.

The described situation forced us to follow a different approach to implement the compression mechanism. We created a plugin for the client and the server that compresses and decompresses the data before transmitting and receiving it, respectively. This is implemented in the file compressor.py and consists on a simple strategy pattern. The implementation of each compression scheme only needs to extend the class Compressor and implement the compress and decompress methods.

The compression mechanism in the server is independent from the one in the client. Possibly the server might be compressing data, while the client might not be interested in doing it. In this implementation, the client needs to be aware of what is the configuration of the server, so it can properly decompress the messages. On the other hand, the server needs to be aware of the compression configuration of all the clients. This information is exchanged when the client is registering with the server.

5. Evaluation
To test our project, we decided to create an application on top of the greenMobile system, called TestApp. This application registers itself with the greenMobile server and proceeds to share a huge number of context-oriented messages. Then the server is going to send it back to the client, allowing for an analysis of how the system behaves.

We created a workload to test the system containing 10000 different records, each consisting of a name, a number, and two locations composed of latitude and longitude. Then we fed this data to the application, which transformed each record in a file containing 10000 different records, each consisting of a name, a number, and two locations composed of latitude and longitude. Then we fed this data to the application, which transformed each record in a context object.

The aggregation mechanism was configured with two aggregation functions:

- **Calculate average**: this aggregation function calculates the average of the numbers aggregated;

\[^{2}\text{Bottle in: http://bottlepy.org/docs/dev/index.html}\]

\[^{3}\text{Paste in: http://pythonpaste.org/}\]
• Calculate minimum bounding box: aggregates multiple locations and finds the smallest square where all of them can be found.

The experiment was conducted to aggregate 1, 2, 5, 10, and 20 messages. The system was also configured to use the lzw algorithm, with the objective of further improving the results. The tests were also executed with the compression turned off to be able to understand how effective the compression mechanism is. The configuration was always the same in the server and in the client side.

Two machines were required to perform the described interaction. The server was running Windows 7 Professional N (64-bit), on an Intel Core 2 Quad Q6600 2.40 GHz, with 3 GB of RAM, and a 100 mbps Intel 82566DC Gigabit Ethernet PHY. The TestApp was running on a tablet ASUS MemoPad 10. This device was running Android 4.2 operating system, on a Quad-Core 1.6 GHz processor, with 1 GB of memory, and a WLAN802.11 b/g/n wireless connection.

5.1. Scalability

Regarding this requirement, we are primarily concerned with the balance between the server’ s bandwidth usage, one of the biggest bottlenecks of this type of applications, and the average message latency verified in the messages sent between senders and receivers. The CPU usage and memory consumption of the client and the server are also analyzed.

5.1.1. Total outbound bandwidth and its effect on the message delay

Figure 7 presents the server’s total outbound bandwidth consumption under different scenarios and its effect on the average message delay. The total outbound bandwidth is measured in kilobytes, while the average delay is measured in seconds.

The results show that the total outbound bandwidth consumption decreases as the number of messages that are being aggregated increases. The reason for this result is explained by the fact that the aggregation function gets more effective, thus reducing the amount of information to be transmitted. The total outbound consumption decreases from 1624 kB with volume:1 to 99 kB with volume:20, representing an improvement of 93.9 %. These results were achieved while having the compression turned off. The number of transmissions was also reduced, leading to a decrease in the amount of information transmitted that consists on headers and structural information.

The results get even better when the compression is turned on. As an example, when the volume value is set to 20 the amount of information transmitted gets almost halved, from 99 kB to 48 kb.

The specified improvement comes at a cost. Since we are allowing the greenMobile system to postpone the propagation of messages in the server and in the client, a delay is expected to occur. In the scenario where more messages are aggregated, the average delay a message takes to reach its destination increases to 3 minutes and 24 seconds. Compressing the messages doesn’t significantly increase the average delay.

The scenario where the server and the client are set to aggregate five messages before transmitting, seems to have a good trade-off between the total bandwidth received and the average delay. By aggregating more messages, the delay significantly increases while not providing a significant reduction in the total outbound bandwidth.

5.1.2. CPU usage

Figure 8 presents the server’s average CPU usage as the time progresses. Each line in the graph corresponds to a different scenario, varying in the amount of messages whose transmission has been delayed. The results showed were achieved with the compression turned on. The CPU usage is measured in percentage, while the elapsed time is measured in seconds.

The results show that the CPU usage decreases slightly as the number of messages that are aggregated increases. The average CPU usage decreases from 16.8 % with volume:1 to 10.47 % with
volume: 20, representing an improvement of approximately 37.7%. By aggregating messages on the client side, the server receives substantially less messages. This fact leads to a decrease of the server’s average CPU usage, even though each message received contains more data.

Figure 9 presents the client’s average CPU usage as the time progresses. The results show that the aggregation mechanism doesn’t have a clear influence in the client’s average CPU usage. The difference between the average CPU usage with volume: 1 and volume: 20 is only 0.11%. The aggregation technique reduces the amount of messages that the client and server receive. The difference is that in the case of the client, the change in the number of messages received doesn’t have the same significant impact as in the server. This happens because while the server has to process each message that receives and eventually transmit it, the client only prints the message.

5.1.3. Memory consumption

Figure 10 presents the server’s memory consumption as the time progresses. Each line corresponds to a different scenario. The memory consumption is measure in kilobytes, while the elapsed time is measured in seconds.

The results show an increase in the memory consumption as the number of messages that are aggregated increases. The server is delaying the propagation of a certain amount of messages, and as a consequence the server has to keep that information in memory. The result is very similar to the one seen in the client. This is presented in Figure 11.

![Figure 11: Client’s memory consumption under different scenarios.](image)

5.2. User interaction

While taking in consideration the delay of each message, we take a look at the execution time and see how it affects the user interaction. The execution time counts the amount of time spent processing the requests. This metric is measured in seconds.

Figure 12 presents the execution times of each scenario. The results show that the execution time has a tendency to improve until it reaches the scenario where 20 messages are aggregated. The increase in the number of messages that are aggregated results in a decrease in the number of message transmissions. This causes a decrease in the time spent communicating with the server. However, by performing the technique aggregation the application spends more time concatenating and processing messages. When the volume reaches the value 20, it passes a point where the time spent processing starts to grow faster than the time spent communicating decreases. In the scenario where the compression is turned off, the execution time decreases from 318 seconds with volume: 1 to 66 seconds with volume: 10, resulting in an improvement of 80%. Compressing the information causes a slight increase in the execution time, as expected.

Users expect certain messages to be delivered in a given amount of time. If the message delivery exceeds that time, the usability of the system is affected. Also, the user expectations are expected to differ depending on the messages. For instance, the user might be willing to wait five minutes before transmitting an email but demand immediacy on the transmission of an SOS message.

Aggregating messages causes the latency of each
message to increase, as shown in Figure 7. When designing the application, the developers should have this in mind and specify for each message a different priority. This way the system can schedule according to the priorities of each message. This should be done using the aggregability function called set_priorities, that allows for the specification of a different aggregability rule based on the content of the message.

5.3. Energy consumption

The energy consumption was evaluated using PowerTutor2. This application is a diagnostic tool for analyzing the energy usage of Android apps. It analyzes the consumption from a different number of sources, such as LCD, CPU, WiFi, and 3G.

In this evaluation, we are particularly interested in the effects of the greenMobile system in the CPU and WiFi energy consumption. Figure 13 presents the energy consumption that was necessary to execute each of the specified scenarios.

![Energy Consumption Graph](image)

Figure 13: Client’s energy consumption under different scenarios.

The results show that the energy consumption decreases as the number of messages that are being aggregated increases. This is a consequence of the fact that the number of transmissions and the total amount of information transmitted decreases, as explained in the Section 5.1. This means that the greenMobile system places itself less times in a high power state during a smaller time duration. When the compression is turned off, we see a decrease in the energy consumption from 1851.3 Joules with volume:1 to 57.6 Joules with volume:20, representing an improvement of 96.9%.

When the compression is turned on, the results generally show a small improvement in terms of CPU and WiFi energy consumption, with the exception of when the volume value is set to 1 in which it almost doubles. This result can be justified by the different delays of each message, due to the compression being turned on, thereby causing the system to consume more tail energy and stay longer at a high power state. This is not visible in higher volumes because as the delay increases it causes both versions to experience the tail energy.

5.4. Portability

As previously explained, Kivy is a cross-platform Python framework for NUI development. It runs on multiple operating systems such as Linux, Windows, OS X, Android, and IOS. This framework contributed tremendously to the increase of the level of portability, as most of the code is able to run untouched on all the supported platforms. However, when developing code for a specific operating system, (e.g. such as taking a picture with a camera) its usefulness is limited to it.

The greenMobile system is developed to run in the Android operating system, but due to the inherent portability of the Kivy toolkit we conclude that it’s possible to port the applications to another operating system, with little to no effort.

6. Conclusions

The greenMobile system delays the message transmission and uses techniques such as aggregation and compression to improve the network performance. This improvement is specially important in the achievement of the greenMobile goal because the wireless network communication subsystem accounts for a major component of the total power consumption.

The tests performed allowed us to conclude that we have successfully developed a system that scales very well, is able to fulfill with user expectations, improves the energy consumption of an application, and the applications developed can possibly be ported to another operating system.

There are some major improvements to the greenMobile system that can be done in the future. The study developed can be extended to include not only the WiFi energy consumptions but also the 3G, 4G, and GSM network interfaces. Another improvement could also consist in adding the capability to automatically change the message transmission fre-
quency, adapting itself to the characteristics of the communication.

Acknowledgements

I would like to express my gratitude and appreciation to all those who gave me the possibility to complete this project: Prof. Dr. Paulo Ferreira for the valuable guidance and advice, the thesis committee, Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento (INESC-ID), Instituto Superior Técnico (IST), and my family for their support and constant encouragement.

References


