Bachelor's Thesis Computing Science

Radboud University Nijmegen

Toward Sustainable Software Energy Strategies for Communication

---

Energy Benchmarking of Push and Pull Methods

Author:
Nikita Kuprins
s1080434

First assessor:
Bernard van Gastel

Second assessor:
Mart Lubbers

June 16, 2025

Abstract

As sustainable software engineering becomes increasingly important in reducing the environmental impact of digital systems, developers face new challenges in choosing energy-efficient communication methods. This research investigates the energy consumption trade-offs between two widely used communication methods push and pull and lays the foundation for future energy sustainable strategies that can help developers. Through controlled experiments using a simulation-based setup, we examine how key parameters such as message size, number of connections, failed pulls, and keep-alive frequency influence overall energy use. Our findings demonstrate that push sometimes in some scenarios is less sustainable due to the cumulative cost of keep-alive messages, while pull suffers from energy overheads in connection management and unsuccessful data fetches. For a given application on certain hardware, we could identify the break-even point where the energy cost of the push method surpasses the pull method.

Contents

Chapter 1

Introduction

1.1 Context

With growing concerns about climate change and the environmental impact of technology, there is an increasing need for sustainable software engineering practices. Just as information technology (IT) has contributed significantly to economic growth and quality of life, IT has an important role to play in sustainability[10]. One of the key aspects that is crucial for reducing the carbon footprint is the optimisation of computational resources, as many systems often operate in power-limited environments, such as battery-powered devices. Optimising energy consumption not only extends the operational life of these devices but also reduces energy waste and carbon emissions.

Existing approaches for enhancing energy efficiency often focus on implementation aspects of the program, such as algorithm optimisation, efficient data structures, parallel programming and others. These approaches are valuable, but there is a lack of guidance for developers in choosing the most sustainable one. This research initiates an exploration into finding ways in doing so to fill the gap. We lay the groundwork through a case study on the energy implications of different communication methods, namely pull and push, and anticipate that this preliminary investigation can be expanded into comprehensive strategies for choosing sustainable communication methods in subsequent research.

1.2 Goal

The goal of this thesis is to initiate an exploration into developing energy strategies that assist developers in choosing a more sustainable communication method. The case study on pull and push is used to achieve that. Specifically addressing the question: *Can we lay the groundwork for creating energy strategies that help developers in choosing a more sustainable communication method?* To answer this question, we first investigate the

following subquestions:

- Q1: Is it possible to find the trend of how the pulls that do not fetch any data affect the energy use?
- Q2: Is it possible to find the break-even point when push starts consuming more energy for a specific application on certain hardware?

1.3 Scope

This research does not propose comprehensive energy efficient strategies for choosing communication methods, but instead establishes foundational insights to guide developers. To that end, we also limit ourselves to pull and push communications methods. Besides, these methods also target the application layer, not the network layer.

1.4 Contributions

This thesis will produce:

1. A reusable program set up that measures the energy consumption under different workloads.
2. A guide for developers on how to choose the most sustainable communication method for different scenarios

1.5 Approach

This thesis builds groundwork toward sustainable energy strategies by investigating the energy implications of push and pull communication methods. The research proceeds in a structured way of dependent stages.

We begin by establishing the distinction between push and pull communication methods (Chapter 2). This includes not on a theoretical distinction but also pseudocode and timeline visualisation, which serves to help highlight where the energy consumption can differ. To strengthen it, we discuss the transmission protocol that is used in both communication methods, and analyse the role of keep-alive messages in push method idle periods.

From this foundation, we define the experiment's evaluation criteria (Chapter 3), including key parameters such as message size, number of connections, keep-alive messages, failed pulls, etc. These criteria are directly derived from the specific characteristics of push and pull. To make our parameters usable for realistic scenarios, we create formulas that transform the timing parameters into counting parameters.

Based on these parameters, we design a simulation-based experimental setup. This setup provides repeatable and controlled energy measurements within a contained environment. The simulation design avoids real timing/delay, focusing on immediate communication events. This design choice was made due to the limitation of idle energy subtraction, which becomes unstable in long application runs due to high variance.

We continue by transforming our setup and criteria into a concrete set of benchmark scenarios. These scenarios are designed not to test all possible variations but to test variations where the spark energy difference is expected.

After all our design choices are finished, the next section addresses the validity issues arising from that, specifically addressing the limitation of idle energy subtraction.

Experimental results (Chapter 4) offer empirical evidence of energy tradeoffs. Discussions (Chapter 5), show how these results directly connect to a guideline for developers, outlining under what communication conditions push or pull is more sustainable. These guidelines represent an initial yet foundational step toward comprehensive strategies for energy-conscious communication method selection.

Then, we discuss the related works (Chapter 6) to strengthen the need for research into the energy strategies that help developers.

Lastly, we make a conclusion and discuss possible future works (Chapter 7) by bringing together all the results we achieved in our experiments and wrapping up the research question. Moreover, the conclusion reflects on simplified simulation design and its alignment with real-world scenarios.

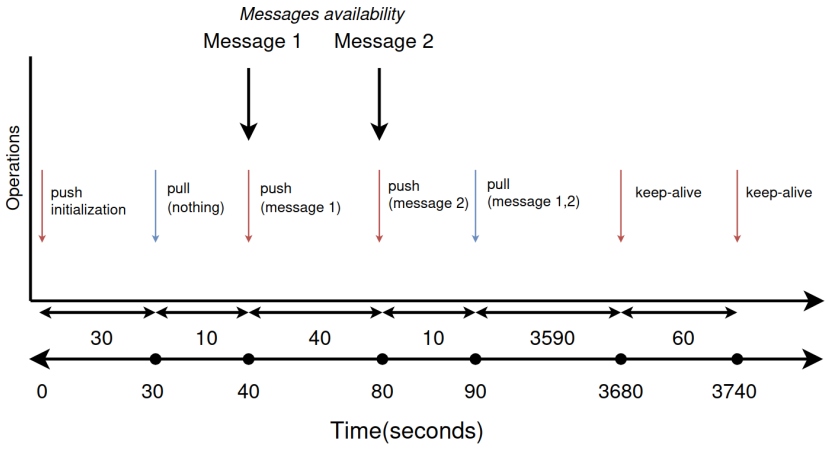
Chapter 2

Preliminaries

2.1 Pull and Push

In software systems, when discussing energy use, the state of a process can be divided into two phases: idle and active. In an active state, the process's instructions are executed by one of the system's CPUs, which may involve data processing, interaction with external components, or other computational tasks. In contrast, an idle state is when the system is not engaged in its main tasks, but it is still powered on and may run background processes. This commonly occurs in systems that await user input, periodic updates, or external triggers. Push and pull are communication methods where both idle and active states play a significant role in determining overall energy consumption.

Push is when the communication is initiated by a server, while the pull method is when the communication is initiated by a client. One software system where it often occurs is email notifications. With the push method, the server sends you an email as soon as it arrives in your inbox, without you having to check for it. This differs from the pull method, where you would have to manually refresh your inbox to see if any new emails have arrived.

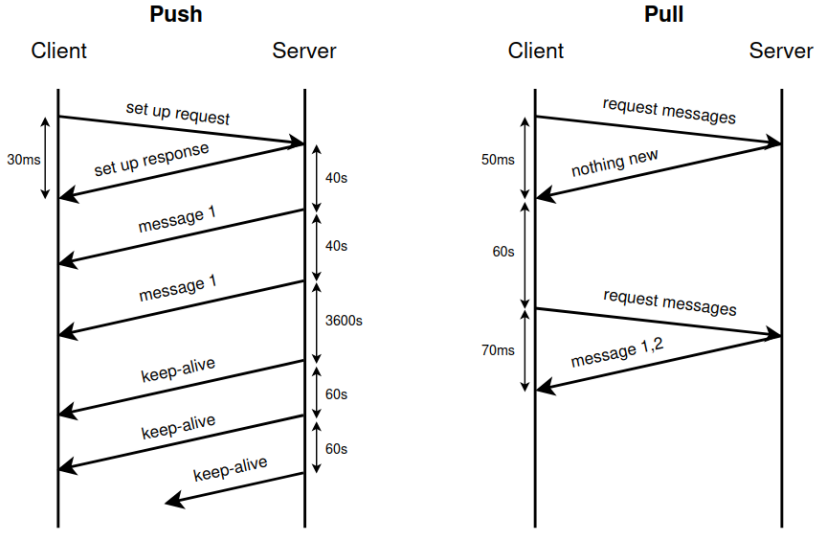


Now, let's look at how push and pull can be illustrated by looking at it from the perspective of the timeline. The graph above displays a sequence of communication events, with red colour indicating push method operations and blue colour indicating pull method operations. In a realistic application, the distribution of message availability and pull timing varies, but in our scenarios, we make an assumption that it can be transformed into frequencies. For example, the pull method above starts after 30 seconds and starts pulling every 60 seconds. If we had a scenario with 3 pulls, then with such frequency we would have had another pull after 60 seconds after the 2nd pull. Our program's flow consists of sending 2 messages, after which no additional messages are transmitted. Let's describe the timeline in detail:

- *0 second:* The timeline starts with the push initialisation phase, when client 1 sends a request to ask a server to start pushing messages, and the server sends a response.
- *30 second:* Now, client 2 uses a pull method to try to fetch a message: he sends a request to the server and awaits a response with a message. However, since message 1 is not available yet, he does not receive it, but instead receives some data indicating data unavailability.
- *40 second:* Message 1 becomes available, and subsequent to that, the server starts pushing that message to client 1.
- *80 second:* Message 2 becomes available, and subsequent to that, the server starts pushing that message to client 1.
- *90 second:* Client 2 now tries another pull, but this time he successfully retrieves both message 1 and 2, since they were already available at the server.

- *80-3680 interval*: In our scenario, there are no other messages on the server, but the push connection is still open. After 1 hour of inactivity, the server starts sending keep-alive messages to the client 2 with 60 seconds frequency.
- *3680 second:* Client 2 receives the first keep-alive.
- *3740 second:* Client 2 receives the second keep-alive.
- *... second:* Client 2 keeps receiving keep-alive messages until connection interruption or close.

Now, let's look at how push and pull with the same times can be illustrated separately in a different dimension:



The same scenario above, but now with more insights. The runtime of the push set-up is assumed to take 30ms. The runtime of the pulls are assumed to take 50ms and 70ms, with the second pull taking more time due to messages overhead. In these timeframes, the system also remains idle.

The timeline and the 2 graphs above can now help make a bridge between these communication methods and sustainability, and show how an underestimation of the energy consumption in the idle state may lead to a wrong choice in sustainable development when selecting between push and pull. If keep-alive messages never occurred for the push technique or would stop after a few, then push and pull would consume approximately the same amount of energy. However, the push scenario may continue sending them for a long time, reaching a break-even point when push starts consuming more energy than pull.

The main idea of why the push method is mistakenly considered to always be a more sustainable choice, comes from the fact that it does not

have to set up the connection and close the connection for each message transmission, like pull does. To explain this, we look at the pseudocode.

- **N** - is the number of connections to establish with the server. We assume that each thread is a client.
- **J** - is the number of messages we push or successfully pull
- **K** - is the number of pulls when we fail to receive our message

Pull

```
1:  connectionThreads = [ ]
2:  for i ← 0 to N - 1 do
3:      In a new thread [
4:      for j ← 0 to J + K - 1 do
5:          connection = TCP.connect(ADDRESS)
6:          send(connection, "I request the message")
7:          read(connection)
8:          close(connection)
9:          wait if needed before the next pull
10:     end for
11:     ]
12:     Add the thread to connectionThreads
13: end for
14: Wait for all threads in connectionThreads to terminate
```

Push

```
1:  Input: connections - a list of TCP connections of size N
2:  connectionThreads = [ ]
3:  for connection in connections do
4:      In a new thread [
5:      send(connection, "Please start pushing messages")
6:      for j ← 0 to J - 1 do
7:          read(connection)
8:      end for
9:      ]
10:     Add the thread to connectionThreads
11: end for
12: Wait for all threads in connectionThreads to terminate
```

As can be seen, the algorithms mainly differ in connection management and data exchange. In the pull, we initiate a connection in every thread on *line 5* and close it on *line 9*. Between these lines, we send a request to

receive a message if it is ready or a failed pull response if it is not ready. This approach creates a high load on the network and the server due to frequent connection establishment(TCP handshake), TCP connections closing, and requests sent. Besides, we also have K number of failed pulls - when we fail to receive our message.

In contrast, in the push, the connections are already initiated, and we pass a list of them as input to the function. In practice, connection establishment is only once before pushing and is kept open until interrupted, which usually happens if it is open long enough(e.g. 12 hours), or is manually closed. Then, in a loop, for each connection, in each thread, we do a push set up to ask the server to start periodically pushing messages to us. After that, we continuously read messages from the server.

It is now clear that for each pull there is an overhead due to TCP handshake, sending requests, closing the connection and failed pulls, whereas the push technique only has the set up after which the server sends data to the client, so that is why push may actually be more sustainable in some cases.

2.2 TCP

Both communication methods are built on Transmission Control Protocol (TCP)[6]. TCP is a popular transport-layer protocol in the Internet protocol stack, and it has evolved over decades of use and growth of the Internet. It provides a reliable, in-order, byte-stream service to applications, which makes it ideal for use in both push and pull techniques. Apart from these characteristics, TCP is connection oriented, meaning it establishes a connection between endpoints for some duration. In both push and pull techniques, the connection is between the client and server. However, in the push technique the connection usually persists until the connection is interrupted, whereas in the pull technique, it lasts only for the duration of a single pull.

2.3 Keep-alive

By design, TCP has no liveness detection capability. However, it can be implemented to address the issues arising when having an application or system crash. In such cases, no packet with a FIN flag will be ever sent to close the connection. In addition, when Network Address Translation (NAT) cannot determine whether the endpoints of a TCP connection are active, it may abandon the connection if it has been inactive for some time[4]. Similarly, a firewall may also close the connection due to inactivity[11].

All these issues are undesirable for the push technique. The use of keep-alive messages helps to prevent them by periodically checking if the connection is still operating. The implementation of it may vary but the general idea is to send a signal with a predefined interval and expect a reply. It can

be implemented both on top of TCP, like in Transport Layer Security (TLS) protocols with heartbeats[7], or in TCP like in Linux[1].

From the sustainability standpoint, it is important to highlight the following characteristics[5]:

1. Keep-alive amount - the total amount of keep-alive messages that we send in total over the period of connection. It highly depends on the implementation. For example, in Linux TCP implementation there are the following parameters:
    - Keep-alive time - the duration of inactivity after which we begin to send keep-alive messages. If during that duration there is no need to send new data, and no data or ACK packets have been received, then it is triggered. The default value is 2 hours.
    - Keep-alive interval - the interval between successive keep-alive messages if the previous keep-alive was not received. The default value is 75 seconds.
    - Keep-alive retry - the maximum amount of keep-alive messages to send before killing the connection. The default value is 9 messages.
2. Keep-alive size - the keep-alive messages should have no data or for compatibility with some TCP implementations 1 byte of data. Keeping them so small is significant for sustainability, as the bigger the message the higher the load.

As can be seen, in Linux the default keep-alive implementation is designed to be energy efficient even in long connections. For example, with the default parameters, a keep-alive message is initiated after 2 hours of inactivity, followed by 8 unsuccessful keep-alive messages and the last one 9th is successful, with a 75-second interval each. Over 24 hours period, this configuration results in approximately 108 keep-alive messages.

However, as highlighted in the documentation for Linux TCP, underlying connection tracking mechanisms and application timeouts may be much shorter, such as those implemented in firewalls or NAT. This is especially significant for Constrained-Node Networks (CNNs), which are a characteristic of the Internet of Things (IoT) and require a lightweight TCP.

Chapter 3

Experiment design

3.1 Design and Implementation

3.1.1 Portability

The challenge of this experiment is to design something that helps developers. The machine that developers use for their application is going to be different from what we use. This implies that there is a need to rerun all benchmarks on different hardware to get the actual energy use, because energy is strongly hardware-dependent. Therefore, it is important to design something flexible, where for any hardware and for the specific application, the developers will be able to find the break-even point.

In addition to that, as already mentioned, the exact numbers of energy use in our measurements will vary across different hardware, but the trend is expected to be the same. The communication operations: sending/reading message, keep-alive, TCP handshake, etc., are fixed by software level. If we measure the energy of a specific application on one machine, we are binded by the cost of the operations on that machine for some specific number and type of operations. On another machine, the number and type of operations stays the same. Only the cost per-operation (in joules) changes due to hardware. Since total energy is just the sum of per-operation costs, the energy use on a new machine is a scaled version of the original, not a completely different shape. Therefore, if push was more expensive than pull after N keep-alives on one machine, the same will happen on another machine, but after J keep-alives. The values change, but the pattern does not.

Another perspective, but simpler, can be made about the claim that it is possible to find the break-even point on any hardware for the specific application, meaning when the push starts to consume more energy than the pull method. We back it by a logical reason: in the push method, even if keep-alives consume a very small amount of energy, in the end, if you send enough of them, they will surpass pull energy use at some point.

3.1.2 Benchmarking

Benchmarking is used to measure the performance of a program under standardised workloads. To measure the energy consumption we use the energy-bench library[8] v0.1.30, written in Rust and developed by Jordy Aaldering. The library measures the energy usage of the entire machine rather than only the CPU and DRAM. By doing that, we make our result more accurate and rigorous.

We assume that a machine is powered on throughout the measurement period and for an arbitrary amount of time before and after the measurement period. In that way, we ensure proper communication and a method that we want to measure adds something on top of the idle energy consumption. Therefore, we can exclude idle energy consumption to receive the energy use of the communication method itself.

In addition to that, we give another strong reason to do it. In reality, every time we measure something, the runtime will slightly vary, and this variance will impact the final results if idle time is not subtracted. In other words, if one computation takes N seconds and another N+1 seconds, and idle is not subtracted, then the final results are not fair, because the second measurement has 1 more second of idle energy use.

The subtraction is implemented in the library by first waiting for the system to stabilise for N seconds, followed by measuring the idle energy consumption for J seconds, and later subtracting it from the subsequent measurements.

To avoid one off fluctuations and avoid other inference, the number of measurements we do is 5. To diminish the deviation in idle energy, the duration of the idle energy measurement is 30 seconds. Finally, to ensure reliable sampling in the chip, the minimal duration of measuring is 2 seconds. If the communication method ends up running less than 2 seconds, it will be rerun as much as needed to reach the 2-second threshold, and the final energy use is then normalised.

3.1.3 Environment

The experiments are run on ODroid H3 with clean Debian 12 with most services disabled. The energy is measured using an INA260 ampere meter. The ODroid H3 is a quad core computer and the one used in this research has 16 GB of RAM and NVMe SSD. In our set up the INA chip has 6.3HZ reading frequency and 806.45HZ sample frequency.

The energy measurements are taken directly from the DC power input of the ODROID-H3, rather than from individual components like the CPU, RAM, or SSD. By measuring in that way, the energy consumption of the entire system is captured.

To run the program we configure a job in the GitLab CI pipeline with

Docker image that contains Rust and necessary dependencies. The job is then run on the GitLab runners. The results are written to the .csv files.

3.1.4 Experiments approach

In a typical scenario for benchmarking push and pull communication methods, there would be pull and push rates. The rate is some delay between each pull or push, so a timer would control the process. When the timer elapses, a push is triggered, while for a pull, the message readiness is triggered so that the next pull response retrieves the message instead of returning a failed pull response.

However, we make a limitation and assume that communication should happen in one go. In other words, this implies that the amount of energy used between the pull/push rates is subtracted. The reasons behind this decision bring us back to section 3.1.2, where we already discussed that we exclude idle energy. It might now look trivial to just build a typical implementation with rates and let the energy benchmarking library subtract idle energy, but this turned out to be a bad approach for two main reasons.

First, having real timings introduces a big limitation for developers who want to do benchmarking. In small cases, when it is a matter of seconds or minutes, it does not really matter, but a realistic scenario may take a full day or even more. This makes it impractical for developers to benchmark all their cases.

Second, while idle subtraction may look trivial, especially in stable systems with most services disabled, a close look revealed significant challenges related to energy consumption in the realistic application. The primary difficulty encountered was the substantial variance in idle energy consumption for long runs. This variance affects the results a lot, as the duration of idle states in a realistic scenario far exceeds the active periods and energy use of pull or push techniques.

To be concrete, the results could take the following form:

1. The average total energy use among 5 runs was reasonable but the standard deviation could be more than 40% of the final result

2. In certain runs, the total energy use of 4 out of 5 was low and varied around 1-3J but there was one outlier with energy consumption that could be 10 times higher.

3. For some runs, the energy use of CPU was $0J$, this is because idle energy use was much more than our communication methods.

Despite practical challenges, we theoretically stand that the simulation accurately reflects the energy consumption of a realistic application. This assertion is based on the fact that no computational work is performed

during idle states in our communication methods, and therefore, there is no overhead during these delays. Given all that, we construct a simulation without delays. The simulation code is discussed in 3.1.7-3.1.9 sections after discussing key parameters.

3.1.5 Experiments parameters

For measurements, we identify parameters that affect the energy consumption. We start with the general idea, which is to make a scenario, such that there is an N number of clients and each client pulls for J times or the server pushes for J times. Therefore, we introduce the following parameters:

- N_{conn} - specifies the number of connections. Having this as the configuration parameter enhances the flexibility of the program. While it is possible to benchmark with a single connection and handle each pull sequentially, real-world scenarios usually involve a server receiving thousands of pulls at the same time.

  Besides, scaling up the connections scales up the results, which helps produce more reliable results, because the energy use of a single push or pull is negligible, making the results highly sensitive. If the energy use is too small, then even a slight deviation in idle energy consumption impacts the result substantially. This could also be fixed by having hundreds of pulls or pushes.

- N_{push} - specifies the number of pushes.

- N_{s_pull} - specifies the number of successful pulls. A pull is considered successful if it retrieves the message.

We give a few examples:

1. Example pull: N_{conn}= 50 clients connect to the server, each making N_{s_pull}=10 successful pulls.

2. Example push: N_{conn}= 50 clients connect, and the server pushes N_{push}= 10 messages to each client.

Other important configuration parameters that we have to declare are:

- M - message size, specifies the size of a message being pulled or pushed.

- N_{f_pull} - number of failed pulls, specifies the number of pulls that failed to receive the message, because it was not ready yet.

- $N_{keepalive}$ - number of keep-alive, specifies the number of keep-alive messages in the push technique.

One could also define parameters for the ratio, such as "*How many keep-alive messages are in between the actual data messages?*". For example, a ratio of 1:4 would mean 4 keep-alive messages after each push and before the next push. However, this would not bring new investigations, as given the number of keep-alive and the number of messages, we can derive the ratio and the other way around. Let's say we send 3 messages and the ratio is 1:4. But then it is actually reducible to the following: $4 \cdot 3 = 12$ keep-alive in total.

Besides, the ratio can be derived from our parameters. Let's say we again have 3 messages and 12 keep-alive, then $12 \div 3 = 4$. Hence, we do not define the ratio, as it is possible to reduce it to the number of keep-alive and the number of actual messages, and also possible to derive it the other way around.

3.1.6 Timing parameters

In the previous section, we defined parameters based on the number of times and these parameters are used as input for the simulation. In this section we define parameters based on the times. While our implementation has no delays as explained earlier, it may still be more natural to have some timing parameters for developers. The important assumption we make in our definition is that we have frequencies instead of realistic-like varying times of pull/push. This assumption is based on the fact that replicating realistic times may be business-specific, and generalisation of it requires another research. The parameters are the following:

- T_{data} - how often a new message arrives for pull or how often we push the message in seconds
- T_{pull} - how often a client pulls in seconds
- $T_{keepalive}$ - how often send keep-alive messages in seconds. The amount of time between the last push and the start of sending keep-alives is not required as a parameter, as it is assumed that in this timeframe the system remains idle.
- T_{total} - total connection time in seconds

As an example, a pull scenario runs for $T_{total} = 3600$s, pulls data every $T_{pull} = 60$s, but message arrives every $T_{data} = 120$s. Another example, a push scenario runs for $T_{total} = 3600$s and push to the client every $T_{data} = 120$s, and keep-alive are configured to send every $T_{keepalive} = 60$s.

Based on these timing parameters we can also define formulas to calculate the number of times that are going to be used in the program for benchmarking. In other words, we define formulas to calculate the input for our application.

- $N_{push} = \lfloor \frac{T_{total}}{T_{data}} \rfloor$
- $N_{s\_pull} = \min(\lfloor \frac{T_{total}}{T_{data}} \rfloor, \lfloor \frac{T_{total}}{T_{pull}} \rfloor)$
- $N_{f\_pull} = \min(\lfloor \frac{T_{total}}{T_{pull}} \rfloor - N_{s\_pull}, 0)$
- $N_{keepalive} = \max(\lfloor \frac{T_{total}}{T_{keepalive}} \rfloor - \lfloor \frac{T_{total}}{T_{data}} \rfloor, 0)$

3.1.7 Pull

The pull implementation has been shown earlier in the introduction. The main client-side change is that we exclude the wait on *line 9*.

However, the implementation of failed pulls also requires a small change due to the absence of delays, leading to the absence of message readiness. In a single-client scenario, one could simply send N successful pulls followed by J failed pulls. However, with multiple clients it does not work as we deal with concurrency, so only 2 options remain. The first option is to track each client, and the second is to add an extra byte in the pull request for failed pulls. As tracking each client introduces additional complexity, which can lead to some overhead, the second option was chosen. Let's say that the usual pull request is of size M, then the failed pull request is $M + 1$. In such a way, it is easy for a server to differentiate between a successful pull and a failed pull.

For the size of a successful pull request, 46 bytes were used. Consequently, for the failed pull, we send 46+1=47 bytes. There is no universally optimal size, as each developer may have their own requirements for that. Some developers may implement larger pull requests, while others may use smaller ones. 46 bytes is a reasonable size in the real world.

3.1.8 Push

The push implementation has also been introduced earlier. The are no client-side changes for it because the server is the one that sends data. However, one aspect that needs further discussion is keep-alive messages.

Our keep-alive messages are treated as another message, but 1-byte long. In other words, these keep-alive messages are at the application layer, meaning that the server transmits J number of 1-byte messages to the client after M number of messages. If a keep-alive transmission failed, it would have given an OS error, indicating that the connection had been closed. Failed keep-alive retransmission is also possible, but we assume that all our keep-alive messages successfully reach the client. The scope is to send a specific number of keep-alive messages and see how it impacts the energy. This implementation corresponds to the RFC requirements discussed in the preliminaries. Other implementations are also possible, but since it is very business-specific, we limit ourselves to this one.

3.1.9 Server

Below is the pseudocode of the server. Some parts were omitted for brevity, namely, resending in case of partial sends and shutdown event.

Server

```
1:  token = 0
2:  message = generate the message string
3:  selector = reads events from OS
4:  server = TCP.bind(ADDRESS)
5:  selector.register(token, server)
6:  connections = HashMap.new()
7:  loop
8:      event = selector.get_event()
9:      if event.token == SERVER then
10:         connection = server.accept()
11:         token = token + 1
12:         selector.register(token, connection, READABLE & WRITABLE)
13:         connections.put(token, connection)
14:     else
15:         connection = connections.get(token)
16:         Read from connection (in push technique only once for set up)
17:         Write to connection
18:     end if
19: end loop
```

3.1.10 Expected results and scenarios

Having these many parameters makes it unfeasible to test all possible values. Besides, even if we did test all possible values, the results would not be useful to developers because we do benchmarks on a single machine, and they have to be rerun on their machine. Hence, we want to identify cases that allow us to answer our subquestions, to later be able to answer the research question. We benchmark these cases and make general conclusions. All other cases are considered to be business-specific and should be run by developers themselves with their own parameters, and if needed, with their own business-related modifications of pull and push methods.

For all experiments, we have to define the message sizes: { 25KB, 50KB, 100KB, 250KB, 500KB, 1MB, 2.5MB }. These values are expected to touch all critical message sizes: small, average, and big. We do not benchmark anything below 25KB, because the results will be too insignificant, leading to higher variance. All message sizes above 2.5MB are expected to be too big to display the overhead from pull.

The first experiment is going to be done to analyse how the N_{conn}(number of connections) and N_{f_pull} (number of failed pulls) affect the results. This is required to construct all other experiments. We expect connections to scale the results linearly. In that way, we will be able to keep with 1 configuration of this parameter, as changing it will not bring any new investigations. We first benchmark 100 connections, then benchmark 50 connections and compare. In general, 50 connections should be enough to scale up the pull and push energy use, and we expect the benchmark with 100 connection to consume approximately twice as much energy.

For the same experiment, we also have to define the timing parameters, such that they also scale up the results. The exact numbers here are not important as long as they align with our subquestions. The $T_{total} = 36000s$, $T_{data} = 3600s$ and $T_{pull} = 360s$. Therefore,

- $N_{push} = \frac{36000}{3600} = 10$
- $N_{s\_pull} = \min(\frac{36000}{3600}, \frac{36000}{360}) = \min(10, 100) = 10$
- $N_{f\_pull} = \min(100 - 10) = 90$

In the experiment above, we already considered a case when $T_{pull} \ll T_{data}$. Now, assuming that $T_{pull} > T_{data}$ is never the case, the remaining are $T_{pull} < T_{data}$ and $T_{pull} = T_{data}$. For these 2 cases, we have to adjust the T_{pull}.

1. $T_{pull} = 1200s$, so $N_{s\_pull} = \min(\frac{36000}{3600}, \frac{36000}{1200}) = \min(10, 30) = 10$, $N_{f\_pull} = \min(30 - 10) = 20$
2. $T_{pull} = 3600s$, so $N_{s\_pull} = \min(\frac{36000}{3600}, \frac{36000}{3600}) = \min(10, 10) = 10$, $N_{f\_pull} = \min(10 - 10) = 0$

The remaining critical parameter that has not been touched yet is the keep-alive messages. So far, in all scenarios above, we had no keep-alive message at all, meaning $T_{keepalive} = 0$. Now, we want to see how they interplay to answer the second subquestion of this research. Finding the break-even point is expected to be possible for all scenarios above. The number of keep-alives to reach the break-even point should increase with the increase in failed pulls or message sizes. The break-even point is only an approximate number and is also not reusable for other hardware or other applications. We only try to identify its existence in our scenarios on our hardware. We find it by making reasonable guesses.

3.2 Validity

Our benchmarks were performed in an idealised network scenario. In real-world, persistent connections, such as in the push technique, may be actually interrupted and dropped. It is not expected to have this every 10 minutes, but if the connection is open for long enough, then it could happen. It would lead to client needing to reconnect. Such reconnections introduce additional energy costs, which are not taken into account in this research. As a result of this, the energy efficiency of the push technique may be slightly over-optimistic. Another issue with the idealised network scenario is the absence of packet loss and latency, which typically occur in real world.

The pull technique may be optimised in real-world scenarios, such that one pull can immediately retrieve all messages that are ready, leading to a smaller energy use due to fewer requests sent. Our benchmarks do not consider this and send requests for each message.

The data size for pull requests may vary in real-world scenarios. While our benchmarks use a reasonable pull request size, if changed, it may lead to an energy use increase or decrease depending on the request.

In 3.1.1 section, we discuss the portability. While we tried to give the general impression of why it is possible to generalise from our experiments, a more formal and rigorous proof is still required.

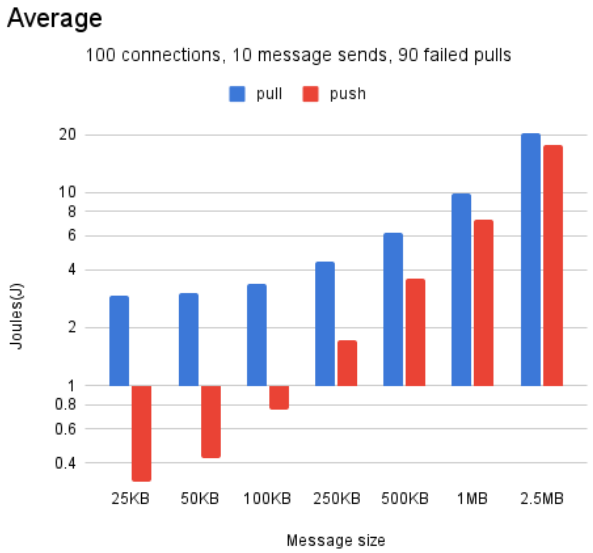
To prove that our simulation produces the same results as the realistic application, we would have to add delays between pulls and pushes in our simulation. However, given the odd inconsistencies and the inherent variability in energy consumption during idle states, a direct comparison between simulation and the realistic application with rates was unfeasible. This still remains a threat to the validity and future research is needed into that.

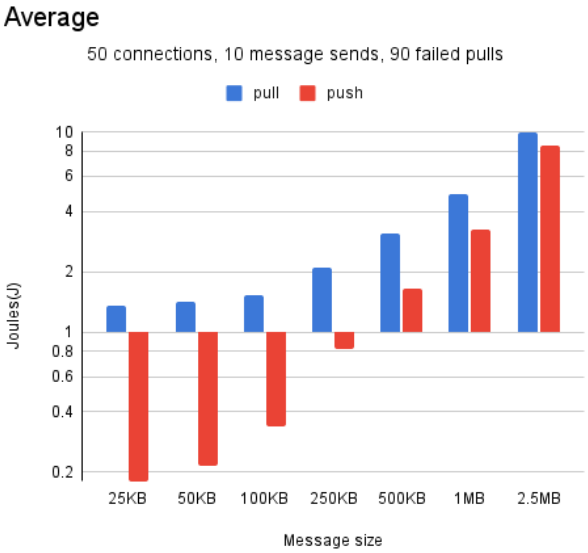
Chapter 4

Experiment results

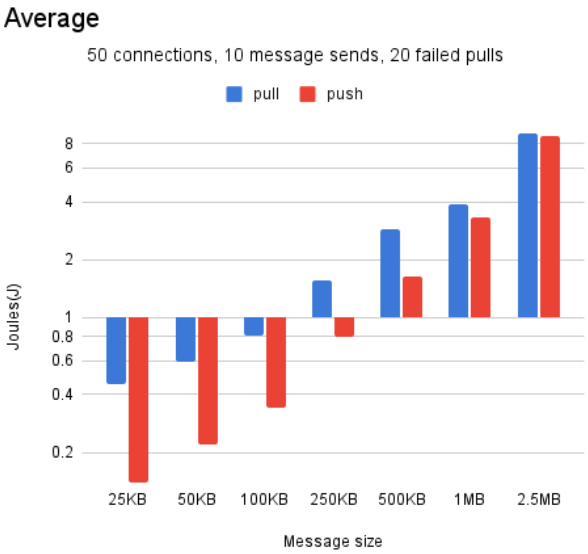
4.1 Connections and subquestion 1

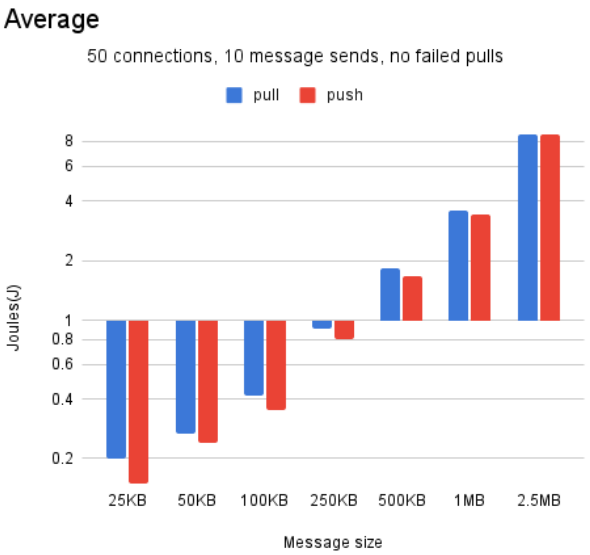
The 2 graphs below depict the experiment needed for investigating how the number of connections scales the results and how the number of failed pulls affect the difference between pull and push. In both graphs, we see that energy usage increases with message size. Pull is significantly more energy inefficient than push due to high number of failed pulls. The difference becomes less spark with the increase of message size, due to the insignificance of TCP overhead with large messages. Doubling a connection number roughly doubled the energy consumption, indicating a linear relationship.





Another 2 graphs below depict the experiment with 50 connections and 10 message sends, but with different number of failed pulls. In comparison to the previous graph, with 90 failed pulls, we can see that the energy use for pull is clearly decreased and the gap between pull and push has become narrower. We can see the same trend here that the difference becomes less spark with the increase of message size. For 2.5MB there does not seem to be any noticeable difference between 20 failed pulls and 0 failed pulls. The last substantial gap for 20 failed pulls was with 500KB. For no failed pulls, we see that for every message size, pull is slightly less efficient than push.

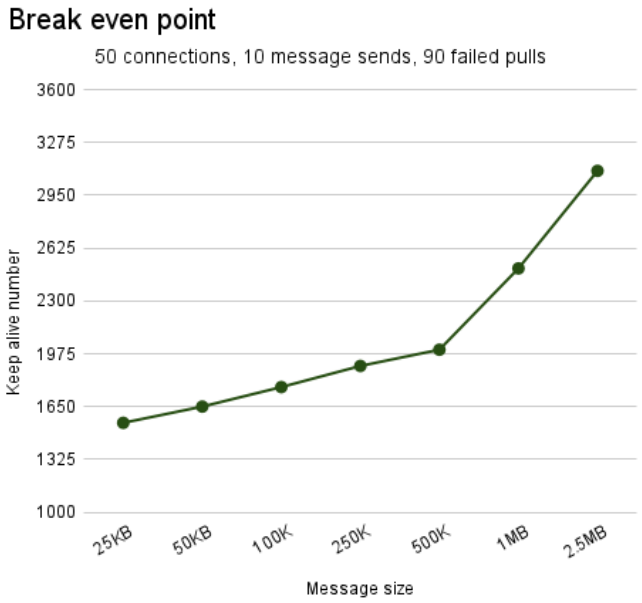


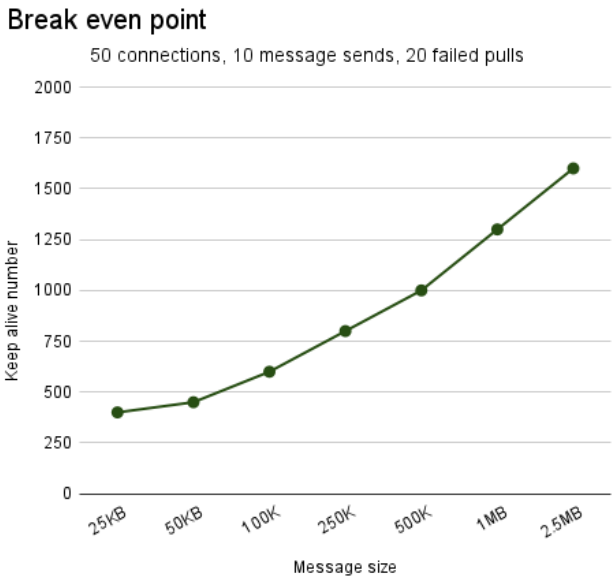


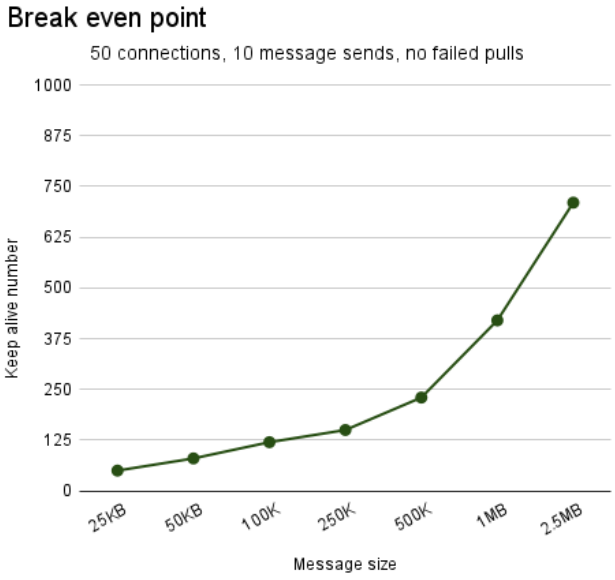
Given all that, we successfully answered the 1st subquestion by identifying that there is a trend of how the failed pulls affect the energy use.

4.2 Subquestion 2

For our 2nd subquestion, we have to find the break-even point for the graphs in the previous section. In other words, we find the break-even point for previous scenarios on our hardware. The 3 graphs below depict the break-even point across different message sizes for 90 failed pulls, 20 failed pulls and no failed pulls, respectively. Previously, we have seen that an increase in failed pulls increases the energy consumption. Now, we also see how it affects the break-even point. The fewer obsolete pulls we make, the fewer keep-alive messages are required to find the break-even point. We also see a general trend that the increase of message size leads to more keep-alives for the break-even point.




Break even point
50 connections, 10 message sends, 20 failed pulls
2000
1750
1500
1250
1000
750
500
250
0
Keep alive number
25KB
50KB
100K
250K
500K
1MB
2.5MB
Message size


Break even point
50 connections, 10 message sends, no failed pulls
1000
875
750
625
500
375
250
125
0
Keep alive number
25KB
50KB
100K
250K
500K
1MB
2.5MB
Message size

Chapter 5

Discussion

After all experiments and answers to our subquestions, we make a bridge between all that and the strategies to answer the research question.

The findings indicate that the choice between push and pull in sustainable development is not trivial and requires rerun on different systems with business-specific requirements. The push communication method, while potentially being more energy efficient under conditions of very sparse data availability, may lead to increased energy consumption through the keep-alive mechanism. Conversely, the pull communication, while usually being considered to be less sustainable due to the overhead of new connections, may be a good choice in cases with low failed pull rates. The main insight is that it is possible to find general trends in pull versus push. These trends can be generalised to different hardware, as explained in 3.1.1 portability section.

Our current simulation captures essential aspects that can lay the groundwork for creating energy strategies to help developers. Specifically, all key parameters of the communication methods (message size, number of clients, data availability, keep-alive messages etc.) are exposed for customisation to allow developers to explore their own use cases. Besides, a customisable number of measurements, minimum duration and idle duration ensure rigorous energy results.

Moreover, with our experiment results, we are also able to provide general rules for developers to help to choose the most sustainable communication method:

- Developer should be careful while choosing the pull method in systems with high traffic of low-size messages, as the TCP overhead is substantial for that size. For large messages (>1MB), the overhead is insignificant in comparison to the energy consumption of the message itself.
- Developers should know that a high increase in failed pulls substan-

tially increases the energy use. The push method is expected to be a better choice in such cases.

- All previous remarks may indicate that pull is always inefficient, but developers should know that persistent connections in the push method incur energy costs from keep-alive. The break-even point exists, such that the cumulative keep-alive cost exceeds pull's connections overhead.
- Developers should know that the number of clients linearly scales the energy use for both push and pull methods.

However, developers should still be cautious and pay attention to the validity of this. They should do their own measurement with their own parameters when necessary.

Chapter 6

Related Work

A few prior studies on energy-efficient communication methods targeted sustainability on mobile devices. One of them is Saifur Rahman et al.[9] research that focused on the long-lived TCP connections in mobile networks. They propose iterative probing techniques for dynamically adjusting heartbeats or keep-alive messages - binary, exponential and composite search. These techniques detect middle-box binding timeout and adjust the keep-alive interval. While their approach optimises keep-alive messages interval, it was not focused on comparing push to the pull and also lacks measurements under varying message sizes to derive energy consumption in an active state as well. Another mobile-focused research was by Burgstahler et al.[2] discusses push and pull in mobile notifications, focusing on energy consumption and notifications latency. They provide an approach that combines both algorithms and transitions between each other, leading to the lowest latency and energy consumption. They show that their approach saves 7% of the smartphone battery daily.

Another prior research but with a different focus, Yi Xu et al.[13] discuss push and pull but in the context of sensor-based systems (SBS). When data is needed from the sensor at a specific rate the push has less energy overhead, since only 1 subscription is needed, whereas the pull has a penalty for each data query. However, push is less efficient for sporadic data because it continuously transmits data regardless of whether it is needed or not. Similarly, in this research, TCP push can be less efficient for sporadic data as it continuously transmits keep-alive messages. They proposed the push/pull envelope (PPE) optimization algorithm, that dynamically adjusts push and pull rates for better energy savings.

There is also research focused on 3G networks for always-on applications. Haverinen et al.[3] analysed how much energy is consumed when using keep-alive messages in WCDMA networks - 3G. They showed that energy consumption is significantly influenced by the frequency of keep-alive messages. This is especially the case in the protocols based on UDP, as

they often require a very high frequency. A single keep-alive message may consume 0.15-1.0 mAh energy in a 3G network.

The research with methodological relation was made by Pereira et al.[12] that looked at the energy efficiency across different programming languages. Their setup measures CPU and DRAM energy consumption in multiple benchmark problems using Intel RAPL. Although their subject is different from this research, but the methodology of using benchmarking is alike. Their rigorous and systematic approach to energy measurement using repeated benchmark runs provide a valuable template for this study.

Chapter 7

Conclusion and Future work

This research has laid the foundational groundwork for developing ways for evaluating communication methods in software systems to aid developers in choosing the most sustainable one. Motivated by the growing necessity for sustainable software engineering, we focused on two commonly used paradigms - push and pull communication methods. We investigated their energy consumption under controlled and repeatable conditions. By structuring work around the 2 subquestions, we provided evidence that challenges misinformed assumptions about the efficiency of persistent connections.

Our experiment design integrated a robust benchmarking set-up using low-level hardware, where we measure the DC input of the whole machine using INA260 chips. Our simulation-based environment avoids idle energy - the energy between pulls/pushes, due to variability in long measurement leading to unpredictable results. The workaround taken was to measure everything in one go and subtract idle energy.

The experiment results from our simulation brought us to several insights. First, pull is energy inefficient when failed pulls are frequent. Second, TCP overhead diminishes with larger message sizes. Third, the number of connections scales the energy use linearly. Fourth, persistent connections in push may become energy inefficient in cases with a high amount of keep-alive messages, meaning that the break-even point can be found for a specific application on certain hardware. These findings form a basis for general design principles that developers can apply when choosing between different communication methods.

The parametrised, reproducible design of our benchmark system serves as a groundwork that developers can extend for more comprehensive ways of choosing a sustainable method. The system allows developers to explore various communication parameters and adapt to their business-specific requirements.

Despite the controlled nature of our setup, several threats to validity must be acknowledged. The simulation excludes idle energy and therefore

pull and push rates, which can influence energy patterns in real systems. Moreover, the communication occurs within a single machine, abstracting away from the complexities of networked environments. These limitations constrain the generalisability of the results. Normally, when you measure on one machine, you do not have any additional information about the results on another machine, but due to the nature of our communication methods, we measure on one machine and still find a general trend, backed by the reasons discussed in the portability section.

Future research could consider other communication methods, such as publish-subscribe or message queues with systems like RabbitMq or Kafka. This will lead to better generalisation and, therefore, would extend the practical value of the research across different domains.

Another future work may be used to not broaden but to dive deeper into pull and push methods. Our pull and push were benchmarked with fixed frequencies, but in reality, they can vary, leading to different energy use. Our keep-alive implementation did not consider any optimisations such as adaptive frequency, which decreases the frequency over time for long connections, leading to less energy use. Moreover, while this research looked at the break-even point, future work may try to create a formula that may help developers choose a more sustainable technique based on their parameters.

Bibliography

[1] Linux manual page. man. https://www.man7.org/linux/man-pages/man7/tcp.7.html Accessed on: 06-01-2025.

[2] Frank Englert Ronny Hans Daniel Burgstahler, Nils Richerzhagen and Ralf Steinmetz. *Switching Push and Pull: An Energy Efficient Notification Approach*. 2014.

[3] Henry Haverinen; Jonne Siren; Pasi Eronen. *Energy Consumption of Always-On Applications in WCDMA Networks*. 2008.

[4] Internet Engineering Task Force (IETF). NAT behavioral requirements for TCP. RFC 5382. https://www.rfc-editor.org/rfc/rfc5382 Accessed on: 05-01-2025.

[5] Internet Engineering Task Force (IETF). Transmission control protocol TCP. keep alives. RFC 9293. https://datatracker.ietf.org/doc/html/rfc9293/#name-tcp-keep-alives Accessed on: 05-01-2025.

[6] Internet Engineering Task Force (IETF). Transmission control protocol TCP. key concepts. RFC 9293. https://datatracker.ietf.org/doc/html/rfc9293\#name-key-tcp-concepts Accessed on: 05-01-2025.

[7] Internet Engineering Task Force (IETF). Transport layer security (tls). RFC 6520. https://datatracker.ietf.org/doc/html/rfc6520 Accessed on: 06-01-2025.

[8] Jordy Aaldering. EnergyBench library for benchmarking in Rust. https://lib.rs/crates/energy-bench Accesed on: 09-02-2025.

[9] M. Saifur Rahman; Md. Yusuf Sarwar Uddin; Tahmid Hasan; M. Sohel Rahman; M. Kaykobad. *Using Adaptive Heartbeat Rate on Long-Lived TCP Connections*. 2017.

[10] San Murugesan. *Harnessing Green IT: Principles and Practices*. IEEE, 2008.

[11] National Institute Standards and Technology (NIST). *Guidelines on Firewalls and Firewall Policy*. 2009.

[12] F. Ribeiro R. Rua J. Cunha J. P. Fernandes J. Saraiva R. Pereira, M. Couto. *Energy efficiency across programming languages: how do energy, time, and memory relate?* 2017.

[13] My Thai Mark Scmalz Yi Xu, Sumi Helal. *Optimizing push/pull envelopes for energy-efficient cloud-sensor systems.* 2011.