Power Aware HTTP: Modifying HTTP to optimize power consumption

Ankit Goyal
University of Texas at Austin
ankit@cs.utexas.edu

Sreedevi Surendran
University of Texas at Austin
sreedevi@cs.utexas.edu

Abstract

A major part of the Internet works on the HTTP protocol. Optimizations to the protocol have been suggested to improve the performance of the protocol. However, less attention has been paid to the power consumption aspect of the protocol which is bound to play a significant role in the coming years due to an increase in mobile devices which are constrained by limited power capabilities. In this paper, we first analyze where exactly the power is consumed, how configuration changes affect power consumption and then we propose a novel solution to reduce the power consumed by HTTP without compromising on performance.

Keywords Network Protocol, HTTP, TCP, Power Aware, Wireless Networking

1. Introduction

The field of wireless communication has seen tremendous progress in recent years. Mobile devices are on the increase and being connected to the internet while on the move is essential in todays society. One of the greatest constraints to realizing this goal is finite power supply which leads to short continuous operation time of mobile devices.

In this context, we first try to pinpoint the part of the HTTP protocol which leads to power consumption. We then make changes to the configurable server side properties including timeout, keepalive values etc. to analyze the level of impact these parameters have on the power consumption and finally, propose a power aware HTTP protocol that can optimize the power consumption of HTTP.

The rest of this paper is organized as follows: Section 2 highlights the background and related works of hypertext transfer protocol, transmission control protocol and power consumption issues over HTTP. Section 3 highlights the motivation behind this paper. Section 4 gives an overview of our proposal. Section 5 presents our proposed implementation of a power aware HTTP. Section 6 outlines the challenges faced during the course of the implementation. Section 7 describes the results of our implementaion. Section 8 lists some other optimizations to optimize the power consumption of HTTP. Section 9 discusses related works. Section 10 includes our concluding remarks. Section 11 discusses the future work in the context of our work and power optimziation of HTTP.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Background

In this section we give a brief overview of the HTTP protocol. This is followed by an overview of the connection establishment and teardown in TCP. We then discuss the inefficiencies in the HTTP protocol as it is implemented today and the power consumption observed in HTTP.

2.1 HTTP Protocol Overview

The HTTP protocol follows a client server model where a request (in simple ASCII format) is sent from the client to the server, followed by a response (in simple ASCII format) sent from the server to the client. An HTTP request includes the following elements: a method which can be either GET, PUT, POST, DELETE etc; a set of headers which specifies the type of content the client is ready to accept, authentication data etc; a payload field to hold data for PUT method etc. The server processes the request and sends a response to the client. The response includes a status code that indicates whether the request was successful or not and if not, the errors in the request are returned. The response also includes information about the data returned by the server like content-length of the response and the output generated by the server-side script (can be an HTML/ XML/JSON file etc).

HTTP utilizes TCP for reliable transfer of information over the internet. To analyze the power consumption model of HTTP, it is relevant to give a brief overview of the TCP connection establishment and teardown.

2.2 TCP Protocol Overview

2.2.1 Connection Establishment

For a client to establish a TCP connection with a server, the client must send a SYN and receive an ACK for it from the server. Thus, conceptually, there should be four control messages passed between the devices. However, it's inefficient to send a SYN and an ACK in separate messages when both of these can be communicated simultaneously. Thus, in the normal sequence of events in connection establishment, one of the SYNs and one of the ACKs are sent together by setting both of the relevant bits (a message sometimes called a SYN+ACK). This makes a total of three messages, and is called a three-way handshake.

2.2.2 Connection Teardown

Each side terminates its end of the connection by sending a special message with the FIN (finish) bit set. This message, sometimes called a FIN, serves as a connection termination request to the other device, while also possibly carrying data like a regular segment. The device receiving the FIN responds with an acknowledgment to

the FIN to indicate that it was received. The connection as a whole is not considered terminated until both sides have finished the shut down procedure by sending a FIN and receiving an ACK. Thus, termination isn't a three-way handshake like establishment: it is a pair of two-way handshakes. The states that the two devices in the connection move through during a normal connection shutdown are different because the device initiating the shutdown must behave differently than the one that receives the termination request. In particular, the TCP on the device receiving the initial termination request must inform its application process and wait for a signal that the process is ready to proceed. The initiating device doesn't need to do this, since the application is what started the shutdown process in the first place.

2.3 Inefficiencies of HTTP

The HTTP protocol as it is implemented today has some notable inefficiencies. According to the current specification, while fetching a webpage using HTTP, explicit calls to the server are required to fetch individual resources on a webpage even though both client and server know that multiple resources need to be shared. In HTTP/1.0, a new conenction is established on every subsequent request (Every such call requires the setup of a new TCP connection). This increase in the number of calls to the server adds to the power consumption and latency. However, this was changed in the specification of HTTP/1.1. Modern browsers try to optimize this problem by maintaining "persistent" connections and pipelining requests. Also, HTTP does not impose any requirements on data compression.

2.4 Power Consumption in HTTP

2.4.1 Non Persistent HTTP Connections

In non-persistent HTTP connections, all the resources on a webpage are fetched using different TCP connections. Since the client sets up a new TCP connection for every HTTP request, the number of requests are propotional to the number of resources on the page. A typical webpage consists of a number of images and text, and all the resources are fetched using a different HTTP connection which increases the number of round trips. Each round trip requires allocation of new resources such as port numbers, bookkeeping data structures, etc. which leads to more processing power. Figure 1 shows the establishment of a new TCP connection to fetch each resource on a webpage.

Due to an increase in the effective number of requests, the probability of collisions increases and TCPs response to network congestion by the slow start approach [2] leads to inefficient use of the available network bandwidth. TCP does not reach full throughput until the effective congestion window size is at least the product of the round-trip delay and the available network bandwidth [1]. This means that slow start limits TCP throughput, which leads to more roundtrips and consequently higher power consumption.

2.4.2 Persistent HTTP Connections

In a persistent HTTP connection, a single TCP connection is used to send and receive multiple web requests and responses using the keep-alive header field. This eliminates the need for establishment of a new TCP connection to fetch each resource which in turn reduces the number of effective calls. As corroborated by our experiments, power consumption is directly propotional to the number of requests made. Figure 2 shows all the resources being fetched using a single TCP connection. The average energy consumed by non-persistent connections in 802.11 was found to be 11.8% more than that consumed by persistent connections.

However, in practice, each server has an upper limit on the number of concurrent HTTP connections it can keep open. Wwe need to find an optimal time period to keep a TCP connection open

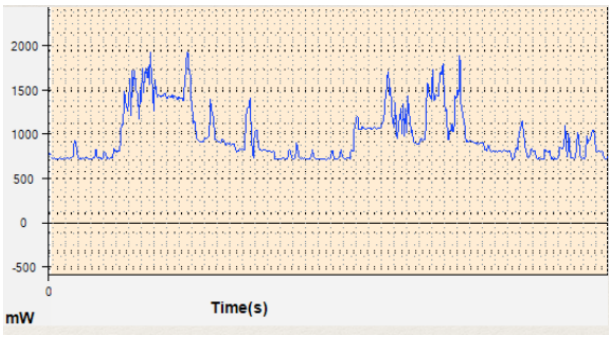

Figure 1. Non-Persistent HTTP Connection on 802.11

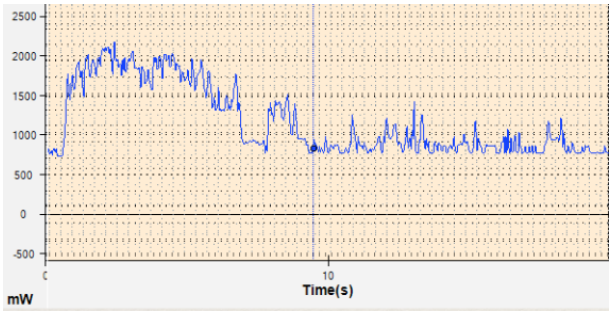

Figure 2. Persistent HTTP Connection on 802.11

given variables like content-length of the data being exchanged. Moreover, keeping connections open on a cellular network may not be the most optimcal strategy since a client (unnecessarily) remains in a high power consumption state during that period as seen in Figure 9.

3. Motivation

The increase in the number of effective calls has a severe impact on the battery life of resource-constrained devices. Since the power consumption increases with the number of requests made, we focus on novel methods to reduce the number of HTTP requests made by a webpage.

Today, websites try to reduce latency and power consumption by caching various resources. However, clients need to check with the server about the freshness of the resources on a webpage. Typically, conditional HTTP enabled by *If-Modified-Since* and *If-None-Match* header fields are used by clients to check the validity of the cache. For such requests, a resource is returned only if it has been changed on the server otherwise a status code of 304 (not modified) will be returned without any message-body.

In HTTP/1.1, to load a cached webpage, the number of conditional GET requests equals the number of resources. For resources that are not modified often, this is an unnecessary overhead.

Table 1 shows the number of unmodified resources relative to the total requests made by the 10 most popular websites based on a study by Alexa Internet (which ranks webistes based on page views and unique site users). It can be seen that a majority of the resources on most of these cached webpages are seldom modified and unnecessary HTTP calls are being made to the server to check the freshness of resources.

Table 1.

	Total Requests	Not Modified
Google	21	18
Apple	83	81
Facebook	24	20
Youtube	21	8
Yahoo	98	39
Baidu	13	10
Wikipedia	33	33
Twitter	5	3
Tencent QQ	201	69
Amazon	215	190

4. Proposal

In pratice, a majority of resources on a website change infrequently. We try to reduce the number of conditional requests required to check whether the resource has been modified or not. For a page that is loaded from the cache, we propose *Power-Aware HTTP* , a novel extension to HTTP which involves the addition of two new header fields to the HTTP GET request method - **check-resource** and **resource-status** . These headers will enable a client to check the status of multiple resources in a single web request and a server to respond with a single bitmap containing the resource change status of multiple resources.

As shown in Figure 3, the *check-resource* header field will allow a client to pass MD5 hash values for multiple resources to the server and the *resource-status* header field will allow the server to send the validity information for multiple resources to the client. To summarize, when a cached webpage is to be loaded, a single GET request will be made by the client, which will contain the MD5 hash of all the resources on the page. The server uses the MD5 hash to match resources and check whether the resources have been modified (one-by-one) or not. The server then returns the status (modified/not modified) for every resource to the client. The client then makes GET requests for only those resources that have been modified.

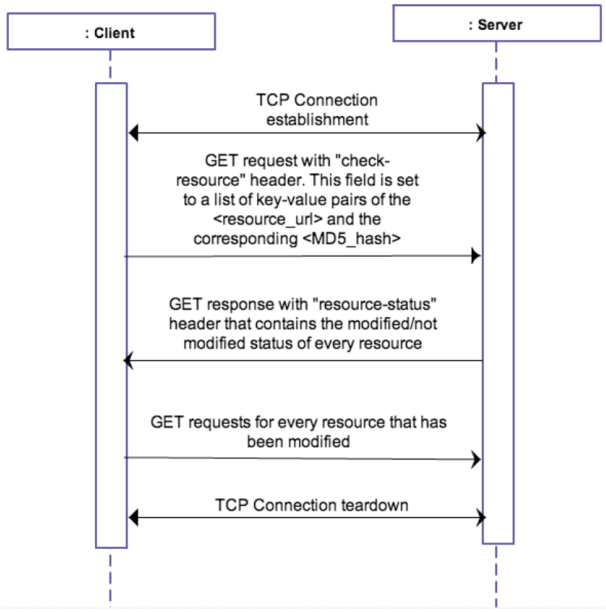

Figure 3. Client-Server interaction using Power-Aware HTTP

5. Implementation

We implemented our proposal at the application layer, and tested its performance over wifi and cellular networks (3G).

5.1 Client

An additional header field *check-resource* is added to the single GET request made by the client. The value of this field will be set to a list of key-value pairs of the resource-url and the corresponding MD5 hash. When the client receives the modified/not modified status of the individual resources from the server's response header, it will then make GET requests for only those resources that have not been modified.

We implemented a native android application that allowed us to make a single GET request for multiple resources using the new *check-resource* header field.

5.2 Server

The server responds with an additional response header field (*resource-status*), which is set to a bitmap containing the modified/not modified status of every resource on the webpage.

We used a Ruby framework called Sinatra, which allowed us to add a middleware to intercept the conditional request and add a new header field to the response header of the Web-brick server.

5.3 Technical Setup

The components used for the implementation are listed below.

5.3.1 Hardware

1. Power Monitor by Monsoon Solutions (Figure 4)
2. Mobile Device: HTC desire one running Android 4.2.2

5.3.2 Software

1. Application: Native android application
2. Servers: Apache Tomcat, Web-brick
3. Framework: Sinatra
4. Analysis: Matlab

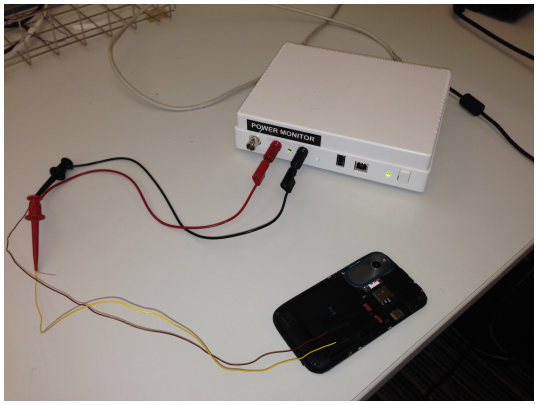

Figure 4. Monsoon Solutions Power Monitor and PowerTool software were used to capture power measurements for HTTP interactions on the Android device.

6. Challenges

While measuring power for different HTTP requests, a lot of spikes were observed in the power consumption graph which were not related to HTTP, since the connection was either in an idle state or the HTTP request had been completed. These could have happened

due to the following reasons:
a) Different browsers have different power consumptions and have different scheduling policy to fetch the results. For example, Google Chrome starts fetching a page as soon as you start typing the URL which may require additional processing and thus adds to the power consumption
Solution : To address this, we created an Android app that allowed us to make HTTP calls with custom headers for various scenarios eliminating any power consumption due to rendering or other optimizations. All the HTTP connections were reused and the persistence of a connection was controlled from the server side by setting the KeepAlive flags.

b) There are lot of background processes in an Android device that consume power and it becomes difficult to know whether the power spikes are due to the monitored HTTP calls or due to other processes in the system
Solution : To alleviate the effect of other applications, we considered the average power consumption over several readings. We also subtracted the baseline from the original signal to mitigate the effects of continuously running background processes. We applied the Butterworth low-pass filter to mitigate the interference of high frequency noise.

7. Results

We evaluated the performance of Regular HTTP and Power-Aware HTTP over 802.11 and Cellular networks for different number of modified resources.

7.1 Power Consumption in Cellular

Table 2 shows the average energy consumed by both Power-Aware HTTP and Regular HTTP over a cellular network for different number of modified resources. It can be seen from Figure 5 that Power-Aware HTTP consumes much less power than the Regular HTTP in case the number of modified resources are less. Our solution performs worse than Regular HTTP only when all the resources are modified, but in a typical webpage this is not a common scenario (as seen in Table 1). In the course of our experiments, we observed an average improvement of 112.3% in power consumption assuming the probability of a resource being modified to be 1/2 for all the 10 resources that we tested for.

Table 2. Energy consumption (in mJ) in Cellular

	Power-Aware HTTP	Regular HTTP	Difference
0	8813.55	5109.29	3704.26
1	22819.33	7077.08	15742.25
4	26546.05	10279.8	16266.25
6	32878.45	15107.92	17770.53
10	50532.2	55641.49	-5109.29

7.2 Power consumption in 802.11

Table 3 shows the average energy consumed by both Power-Aware HTTP and Regular HTTP over 802.11 for different number of modified resources. It can be seen from Figure 6 that Power-Aware HTTP consumes marginally less power than the Regular HTTP in case the number of modified resources are less. The actual power consumption may be less or more depending on the network conditions but our solution will always perform better(except the worst case) since the number of requests made are less. Again assuming the probability of a resource being modified to be 1/2, our solution showed an improvement in power consumption by 12.4%.

Note that the relative improvement of Power-Aware HTTP is significantly less in 802.11 than the cellular network since in cellular networks, the radio stays in high power mode for the duration during which the transfer is taking place as shown in Figure 7.

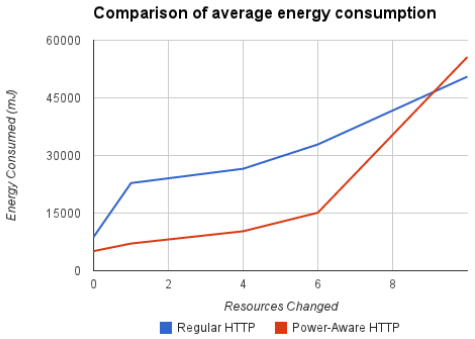

Figure 5. Comparison of energy consumption in a cellular network

802.11 is much more power efficient than the cellular network. It can be inferrred from Figure 8 that the power consumption during the ideal period goes down in case of 802.11.

Table 3. Energy consumption (in mJ) in 802.11

	Regular HTTP	Power-Aware HTTP	Difference
0	402.68	452	49.32
2	418.32	533.68	115.36
5	480	544.56	64.56
8	680.54	758.05	77.51
10	809.23	791.13	-18.1

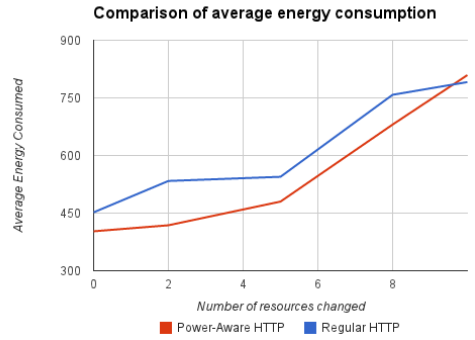

Figure 6. Comparison of energy consumption in 802.11

7.3 Best Case

In this more likely case, for a cached webpage with n resources, out of which m (m <<n) resources have been modified: regular HTTP makes *n* GET requests. Power-aware HTTP, on the other hand, makes *(1+m)* GET requests.

7.4 Worst Case

For a cached webpage with n resources, out of which all n resources have been modified, regular HTTP makes *n* GET requests whle power-aware HTTP makes *(1+n)* GET requests.

8. More Optimizations

We looked at some of the optimizations made by modern browsers. One of the optimizations done by Google Chrome to reduce latency is pre-fetching the URL before a user actually finishes typing the URL.

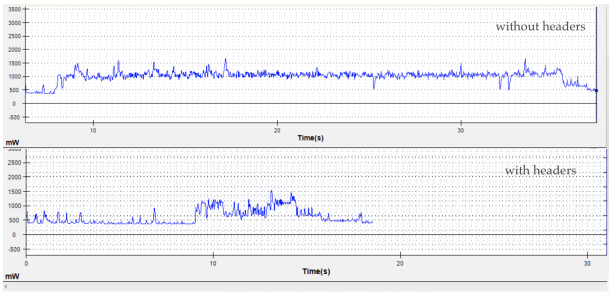

Figure 7. Power consumption pattern in a Cellular network

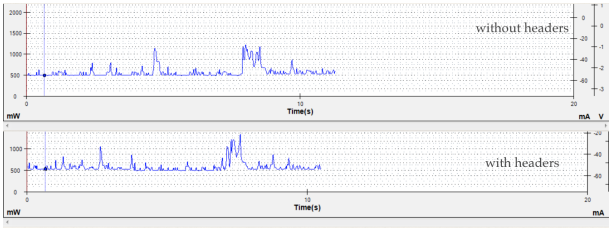

Figure 8. Power consumption pattern in 802.11

Server logs in Figure 9 shows that as soon as the user starts typing, Chrome starts making HTTP requests for each typed character. This approach reduces latency but consumes a lot of power and wastes a lot of network resources.

Solution : We try to address this problem by making the server give hints to the client to prevent premature fetching of urls (by modifying the response of 404). We return the possible URL options matching the pattern in an additional header field. In case of a cellular network, we found that the energy consumption was less by 15.8%.

```
HTTP Requests
-------------

GET /index.hrml          404 Not Found
GET /index.hr            404 Not Found
GET /inde
GET /inde                404 Not Found
GET /ind                 404 Not Found
GET /i                   404 Not Found
```

Figure 9. Chrome URL Pre-Fetch

9. Related Works

Other works that have tried to address the inefficiencies of HTTP include the work on improving HTTP latency by 22% [1]. The authors propose two new HTTP request methods: GETALL and GETLIST to retrieve all the resources on a webpage with a single call. The authors in [1] do not consider the case when the webpage is cached. We propose to have a cient query the server to check the resource change status and make subsequent GET requests for the modified resources based on the response from the server.

Another work suggests the use of a web proxy to reduce power consumption by not waiting for requests on a WAN and the use of request pipelining to reduce the wait period [2].

10. Conclusions

Power consumption in a resource-constrained device depends on a number of factors. In our approach, we have targeted novel means to reduce the number of HTTP requests. In the implentation of HTTP today, all the resources on a cached webpage are fetched using different HTTP connections which increases the number of round trips. We propose an extension to HTTP to reduce the number of requests. The proposed solution shows clear quantitative gains in reducing power consumption over the existing implementation of HTTP.

Our solution to have the server provide hints to the client to handle the pre-fetch url problem may not be the most optimal solution but we try to show that HTTP needs to be smart enough to accommodate optimizations that are aimed at improving performane.

Standardization is necessary for the solution to work accross all browsers. The proposed solution cannot be used until all browsers provide the functionality to use the HTTP GET request method with the *check-resource* header. Different hosting services can provide this functionality if a standard is established.

11. Future Work

We plan to perform an even more robust analysis of the performance of our solution by taking power measurement readings over a period of normal usage. It would also be interesting to analyze the performance of our solution on desktop PCs since the work is equally applicable on all devices.

Acknowledgments

We would like to take this opportunity to thank Prof. Lili Qiu for motivating us for the project and guiding us in the process.

References

[1] Venkata N. Padmanabhan and Jeffrey C. Mogul ,Improving HTTP Latency. Computer Networks and ISDN Systems.

[2] Van Jacobson. Congestion Avoidance and Control. In Proc. SIGCOMM 88 Symposium on Communications Architectures and Protocols, pages 314-329. Stanford, CA, August, 1988.

[3] Jon B. Postel. Transmission Control Protocol. RFC 793, Network Information Center, SRI International, September, 1981

[4] Jen-yi Pan, Wei-Tsong Lee and Nen-Fu Huang, Indirect HTTP: An Energy Efficient Extension of Hypertext Transfer Protocol for Web Browsing