mHTTP: A Multi-source Multipath HTTP Data Transfer Mechanism

Juhoon Kim*
T-Labs / TU-Berlin

Karl Fischer*
TU-Berlin / SJTU

Ramin Khalili
T-labs/TU-Berlin

Yung-Chih Chen
UMass Amherst

Anja Feldmann
TU-Berlin

Don Towsley
UMass Amherst

ABSTRACT

Today, most mobile devices have multiple network interfaces. Coupled with wide-spread replication of popular content at multiple locations, this provides substantial path diversity in the Internet. We propose Multi-source Multipath HTTP, mHTTP, which takes advantage of all existing types of path diversity in the Internet. mHTTP is a concurrent HTTP data transfer mechanism that requires modifications only at client-side socket APIs. One important design principle of mHTTP is transparency to applications which enables the seamless integration of the mechanism into the existing system architecture thus making it easier to be deployed.

We implement mHTTP and study its performance by conducting measurements over a testbed and in the wild. Our results show that mHTTP indeed takes advantage of all types of path diversity in the Internet, and that it significantly improves the performance of the HTTP data transfer by aggregating the capacity available to users. Specifically, we observe that mHTTP decreases download times for large objects up to 50% and greatly improves web browsing performance.

1. INTRODUCTION

The trend shows that the number of mobile handheld devices and mobile Internet traffic is constantly growing. Moreover, the volume of digital content (*e. g.*, mobile games and applications, and high definition multimedia files) is increasing at a higher rate than the infrastructure bandwidth. As a result, the content delivery, *e. g.*, file transfers over HTTP such as application downloads and updates, file sharing via One-Click Hosters (OCHs), and video streaming, becomes more and more bandwidth demanding.

Fortunately, recent developments have opened new opportunities for reducing end-to-end latencies. First, most end-user mobile devices have multiple network interfaces (*e. g.*, 3G/LTE and WIFI). Second, popular content is often available at multiple locations in the network, *e. g.*, Content Distribution Networks (CDNs). When combined, these provide substantial path diversity within the Internet that can be used by users to improve their quality of experience.

Previous work has taken partial advantage of this path diversity in the Internet. Multipath TCP (MPTCP)[1] uses the path diversity available between a single server and a single client [1, 8]. Content Distribution Networks (CDNs) provide replication of content and smart matching of users to appropriate CDN servers, *e. g.*, via PaDIS [23] or ALTO [28] services, which take advantage of this replication. Moreover, there are application specific video streaming protocols that try to take advantage of the replication of streaming contents provided by CDNs [2, 18, 14, 10]. Application specific download managers and peer-to-peer (P2P) software [31, 32, 11] are other examples of related work which benefits from various types of network diversity. The drawback of each of the above approaches is that they do not combine different types of network diversity or if they do, they are application specific.

We propose Multi-source Multipath HTTP, mHTTP, which enables users to establish simultaneous connections with multiple servers to fetch a single content. mHTTP is designed to combine the advantage obtained from distributed network infrastructures provided by CDNs with the advantage of multiple interfaces at end-users. Unlike existing proposals: a) mHTTP is a purely receiver-oriented mechanism that requires no modification neither at the server nor at the network, b) the modifications are restricted to the socket interface; hence, no changes are needed to the applications or to the kernel, and c) it takes advantage of all existing types of path diversity in the Internet.

mHTTP is proposed for HTTP traffic, which accounts for more than 60% of the total traffic in today's Internet [20]. As stated in Popa *et al.* [24], HTTP has become the de-facto protocol for deploying new services and applications. This is due to the explosive growth of video traffic and HTTP infrastructure in the Inter-

*These authors contributed equally to this work

[1]Multipath TCP is an extension to regular TCP that allows a user to simultaneously use multiple interfaces for a data transfer.

net in recent years. mHTTP is primarily designed to improve download times of large file transfers (such as streaming contents). Measurement results have shown that connections with large file transfers are responsible for the bulk of the total volume of traffic in the Internet [21]. While mHTTP decreases download times for large objects by up to 50%, it does no harm to small object downloads and considerably improves web browsing performance.

The key insight behind mHTTP is that HTTP allows one to fetch "Chunks" of files via byte range requests and that these chunks can be downloaded from different servers as long as these servers offer identical copies of the object. mHTTP learns about the different servers that host the same content either by using multiple IP addresses returned by a regular DNS query or by sending multiple queries to multiple DNS servers. mHTTP also works with a single source server when multiple paths are available between the receiver and this server. Hence, mHTTP can also be used as an alternative to MPTCP for HTTP traffic.

The mHTTP design consists of: (i) multiHTTP: a set of modified socket API functions which splits the content into multiple chunks, requests each chunk via individual HTTP range requests from the available servers, reassembles the chunks and delivers the content to the application. (ii) multiDNS: a modified DNS resolver that obtains IP addresses for a server name by harvesting the DNS replies and/or by performing multiple lookups of the same server name by contacting different domain name servers. Moreover, we propose a scheduling algorithm, as part of multiHTTP, which learns about the quality of each established connection and dynamically adjusts the size of the chunks that should be requested over each path.

Our key contribution is the concept of mHTTP along with a prototype implementation and evaluation. We evaluate the performance of mHTTP through measurements over a testbed and in the wild. Our results indicate that

- mHTTP indeed takes advantage of all types of path diversity in the Internet.
- For large object downloads, it decreases download times up to 50% compared to single-path HTTP transmission.
- mHTTP does no harm to small file downloads and greatly improves web browsing performance.
- mHTTP performs similar to MPTCP while only requiring receiver-side modifications. As MPTCP requires changes to the kernel, both at the sender and the receiver, we consider mHTTP to be a viable alternative when using HTTP.[2]

[2]The comparison with MPTCP is performed in a single-server scenario as MPTCP is restricted to the use of a single server. mHTTP, on the other hand, can be used both in single-server and in multi-server scenarios.

The rest of this paper is structured as follows. In Section 2, we describe the high-level design of the mHTTP prototype from a system viewpoint. We explain the two core modules of mHTTP (multiDNS and multiHTTP) in Sections 3 and 4, respectively. Our scheduling algorithm is introduced in Section 5. The performance evaluation of mHTTP for single file downloads is presented in Section 6. Section 7 presents real world experiments carried out by accessing web pages of well-known web servers. In Section 8, we discuss the limitations of our current implementation and planned future work. Section 9 presents related work. Finally, Section 10 summarizes the paper.

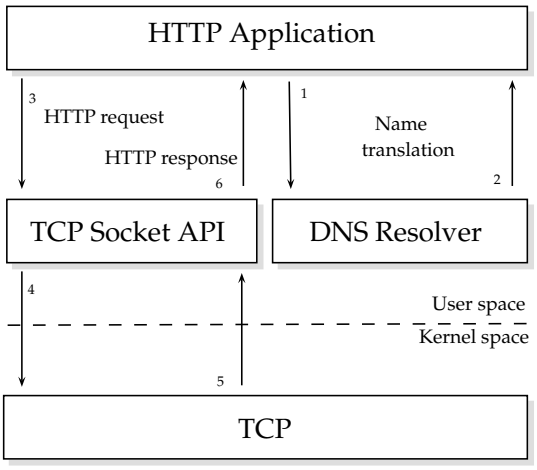
2. DESIGN OVERVIEW

We first describe the high-level design of mHTTP and describe extensions to the current socket API in order to support mHTTP. We then describe the implementation of each component.

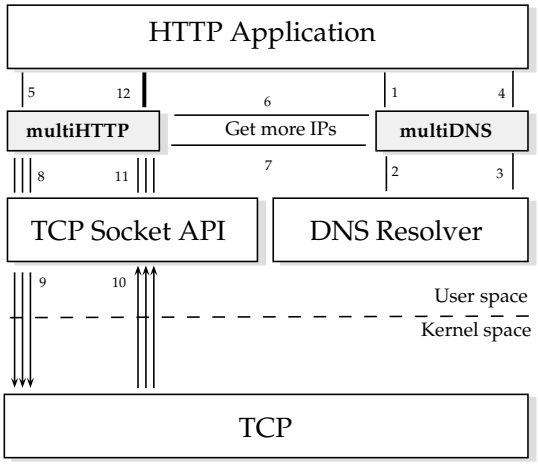
In the HTTP architecture (Figure 1 (a)), an HTTP download from a web server typically requires the name translation from a human-friendly URL to a set of IP addresses. A DNS server can return multiple IP addresses due to load balancing across multiple servers or content distribution; however, traditional HTTP cannot take advantage of this. To overcome this limitation, mHTTP extends capabilities of the client-side socket API. mHTTP is designed with the following three features in mind:

- mHTTP must take advantage of multiple built-in interfaces, multiple paths, and multiple data sources, by establishing simultaneous connections via multiple interfaces to multiple data servers storing identical content.
- mHTTP must not require any modifications at the server-side or in the protocol stack.
- The client-side implementation must be transparent to the application, *i. e.*, modifications must be limited to only the socket API.

The key idea of mHTTP is to use the HTTP range request feature to fetch different content chunks from different servers. We define a chunk as a block of content delivered within one HTTP response message. mHTTP is comprised of two main components, *multiHTTP* and *multiDNS* as shown in Figure 1(b). These two components respectively extend capabilities of HTTP and the DNS resolver. The main purpose of multiHTTP is to handle chunked data delivery between the application and multiple servers and that of multiDNS is to collect the IP addresses of available content sources.



(a) Regular HTTP/TCP architecture



(b) mHTTP architecture at a client.

Figure 1: Structural differences between HTTP/TCP and mHTTP

multiHTTP is the core component of mHTTP. It intercepts all messages sent from the application to the remote end-host (*e. g.*, server). When a TCP connection is identified as an HTTP connection, on reception of an HTTP request from the application, the multiHTTP module modifies the header of the HTTP request by adding a range field. Since the initial HTTP response header includes information about the file size, mHTTP can issue multiple range requests to multiple servers (IP addresses are obtained through multiDNS) that serve identical copies of the object.

multiDNS obtains different IP addresses by performing multiple lookups of the same server name by contacting different DNS servers (*i. e.*, the local DNS server of the upstream ISP for each of its interfaces or third-party dns servers such as a Google DNS server and OpenDNS). Additionally, multiDNS can use the eDNS extension to uncover more servers in a CDN infrastructure. Figure 2 illustrates the process for a 2-connection mHTTP operation in a CDN.

In the following, we discuss functional and technical aspects of these two components in greater detail.

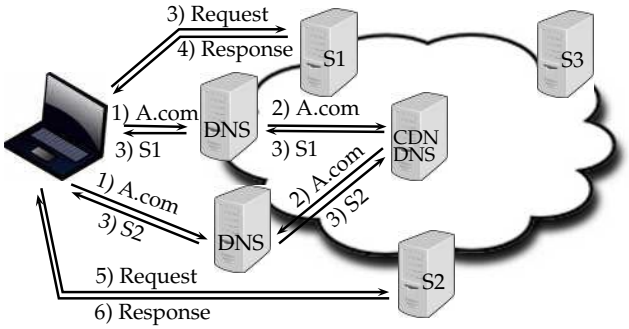


Figure 2: An mHTTP client, with two connections over two available interfaces, operating in a CDN. Servers S1, S2, and S3 host replicas of content of A.com.

3. multiDNS

We describe multiDNS, the core element for discovering IP addresses of servers that hold replicas of a content.

Content replication in the Internet We first analyze the number of IP addresses retrieved by a single domain lookup. We choose the top-1000 domains provided by *Alexa.com* [30] and make requests for the resolution of these names to the local DNS server of a client. As illustrated in Figure 3, even with a single query to the local DNS server, approximately 300 names (30 %) are associated to more than one IP address and respectively 10 % and 5 % of the total names belong to different network prefixes and to different ASes (Autonomous Systems). When performing two lookups, one query to the local DNS and another to the Google DNS, these numbers increase to 35 % (IP addresses), 17 % (prefixes), and 7 % (ASes). This can be seen as evidence that CDNs often provide a different set of IP addresses to a user depending on the choice of a DNS server. Given the fact that the major fraction of the total traffic originates from a small number of popular content providers [3, 7] and the fact that the top 15 domains account for 43 % of the total HTTP traffic in a large European ISP [20], the fraction of the traffic contributed by providers through multiple servers should be significant.

Modification in the domain lookup API In mHTTP, content replication is discovered by multiDNS during IP translation via domain resolution functions such as *gethostbyname()* or *getaddrinfo()*. When these functions return a list of IP addresses, the typical behavior of an application is to choose one of them and to discard the

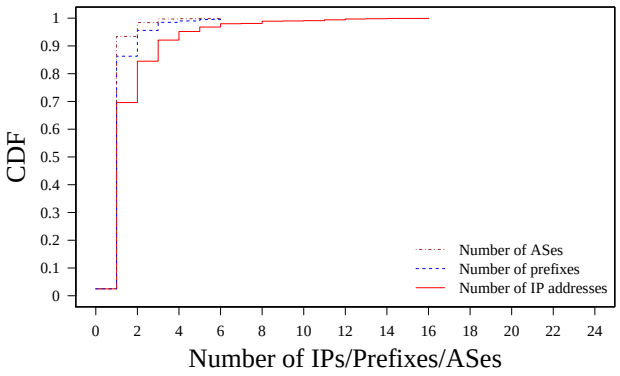


Figure 3: CDF of IP addresses, prefixes, and AS numbers for top-1000 domain names obtained from a single query to the local DNS.

rest. However, multiDNS keeps them for later use.

In the case of a CDN, different DNS servers may provide different sets of IP addresses. Therefore, it is worthwhile querying multiple DNS servers to obtain more IP addresses. multiDNS manages the IP addresses of different DNS servers in different access networks, whenever an interface is activated and the IP address is assigned. It handles the DNS query by validating the availability of a local DNS server in each access network for each interface[3]. If the local DNS is still available at the point of a name translation, a query to that content is made to the local DNS of that particular access network. For each interface, multiDNS receives a list of IP addresses from each access network, and chooses the desired number of IP addresses from every list.

4. multiHTTP

The main task of multiHTTP is to map mHTTP's behavior (*i. e.*, chunked data transfer) to the regular HTTP behavior for web servers and client-side applications (*e. g.*, web browser). mHTTP uses the *byte range request* feature of HTTP specified in RFC2616 [13] to divide the data transfer task (file) into multiple smaller tasks (chunks). mHTTP initiates such a request by adding a *range* field to the header of an HTTP request message including offsets of the first and last bytes of the partial content. If the server supports this operation, it replies with a status code of 206 (on acceptance of the request message) followed by a sequences of bytes. Otherwise, the server replies with a different status code (*e. g.*, 200 OK on success).

mHTTP memory management In mHTTP, a file is associated with a *pool* object, *i. e.*, the central management structure for a connection to exchange states with other connections and to keep track of the file transfer progress. On initialization, mHTTP creates a

[3]The local DNS information is collected when a DHCP request is completed

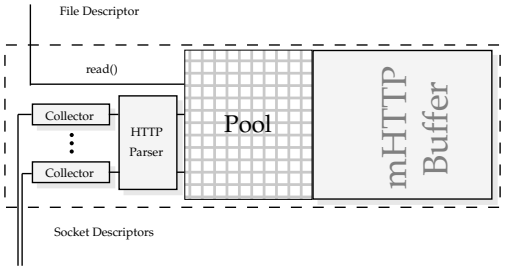


Figure 4: multiHTTP design: a) *mHTTP buffer* stores out-of-order received data b) *HTTP parser* examines and modifies HTTP headers c) *collectors* gather data from individual TCP connection buffers.

ring buffer (*i. e.*, *mHTTP buffer*) in which pool objects can allocate memory for content blocks (chunks).

To drain data from TCP buffers and to move it to the *mHTTP buffer* as frequently as possible, each connection is handled by a dedicated thread called a *collector*. Each thread is responsible for producing requests and managing received data within the *mHTTP buffer*. Moreover, it handles errors and establishes new connections if needed. When a new chunk is requested, the thread notifies its dedicated *pool* to reserve memory within the *mHTTP buffer*. In this way, a *read()* call from the application can read data directly from the *mHTTP buffer*.

Manipulating HTTP headers Once an initial connection is identified as an HTTP connection, multiHTTP creates a *collector* thread with an HTTP parser which examines HTTP messages during the content delivery. The HTTP parser is responsible for four tasks:

- **HTTP request manipulation** The HTTP parser adds the range field to the end of the header with the specified chunk size when the initial HTTP request is sent by the application. When the response message to the initial request arrives back at the application, multiHTTP knows the size of the file and whether or not the server will accept a byte range request for this content.
- **Parsing HTTP headers** The HTTP parser extracts and stores important information from the request and response headers such as availability of content, support for the partial content delivery, content size, and the byte range of the content block.
- **Response header management** In order to allow applications to use mHTTP without modifications, the behavior of mHTTP must be the same as that of a regular HTTP communication from an application's point of view. To this end, the HTTP parser replaces the initial response header (206) with a header that indicates the acceptance

of the request (200). All subsequent headers are discarded by the HTTP parser.

- **Dynamic content filtering** The parser detects dynamic content which cannot be fetched by the mHTTP mechanism. Once such a case is recognized, mHTTP falls back to normal HTTP, transparent to the application.

Additional connections Upon confirmation of the complete delivery of the initial response message, multiHTTP establishes additional TCP connections and *collector* threads, using different IP addresses provided by multiDNS. In order to obtain another IP address, multiHTTP invokes *get_ip()* from multiDNS (see 6 and 7 in Figure 1). The current version of multiDNS hands IP addresses over to multiHTTP in the order that they are retrieved.

Determining what content chunk to request over each connection is another important task of multiHTTP. It keeps track of the requested chunks and decides what chunk and its size to request after the previous chunk on the same connection is completely fetched. The mechanism used for such decisions is described in the next section. .

5. CHUNK SCHEDULER

A critical component of multiHTTP is the chunk scheduler which determines the sequence of chunks and their sizes to fetch on multiple connections. We first study the performance of mHTTP using a simple baseline scheduler. We then propose a more advanced scheduler that efficiently uses the capacity available on each connection.

5.1 Baseline Scheduler

We consider a very simple scheduler that requests chunks of a predefined fixed size over each of the available paths. We study the performance of mHTTP with this scheduler through measurements in Scenario 1, depicted in Figure 8(a) (refer to Section 6 about the configuration of our testbed). In particular, we have an mHTTP client connected via WIFI and LTE connections to two servers in a university campus, downloading 4MB files. We show the completion time of this download for different chunk sizes in Figure 5. We observe that by increasing the chunk size, the performance of mHTTP is improved.

mHTTP introduces a delay each time that a connection performs a range request. This affects the performance of mHTTP, especially when the chunk sizes are small. However, using fixed and large chunk sizes is not recommended as it reduces the responsiveness of mHTTP to changes on the network (*e. g.*, the throughput and the latency of the connections can change over time). Moreover, requesting chunks of a large fixed size

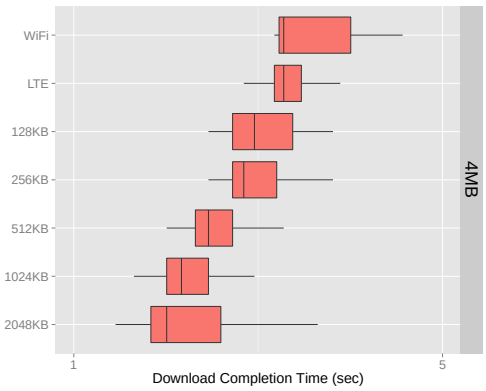


Figure 5: Performance of mHTTP using a scheduler that requests chunks of a predefined fixed size over each of the available paths. We compare the performance of mHTTP with single-path HTTP. We observe that the performance of mHTTP differs for different chunk sizes.

over all connections increases the risk that only the slow connection is active at the end, as the other connection may already have completed the download of its last chunk. This explains why we observe such a high variance in the performance of mHTTP using 2MB chunks.

These results indicate the need to design a more advanced scheduler. In the next subsection, we propose an algorithm that determines and changes the sizes of chunks requested over different paths. The scheduler measures the performance (*i. e.*, throughput and latency) of each of the connections and knows the remaining file size that needs to be downloaded from the servers. It uses all of this information to determine the size of the chunk to be downloaded next over a connection.

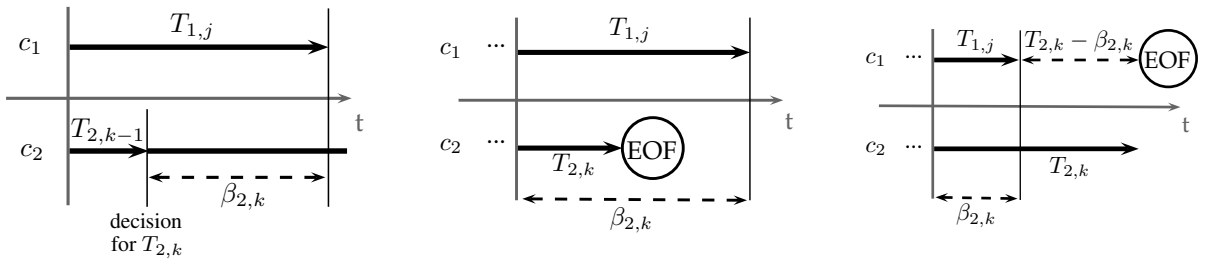
5.2 Advanced Chunk Scheduler

For simplicity, we describe our algorithm when mHTTP uses two connections. The extension to three or more connections is straightforward.

Let c_1 and c_2 be the connections established by mHTTP. We denote by $\overline{R}_i$ and $\overline{rtl}_i$ the estimated throughput and latency (round trip) of c_i, and by $s_{i,j}$ the size of the jth chunk to be transmitted over c_i. The task of the scheduler is to determine $s_{i,j}$ upon the completion of the download of the $(j-1)$-th chunk.

We use the *harmonic mean* to estimate the throughput $\overline{R}_i$ on every socket read operation. As shown in [10], this provides a better throughput estimation than a moving average estimate and mitigates the impact of large outliers due to network variation. Given a series of bandwidth measurements $R_i(t)$, where $t = 0, 1, 2, \cdots, n-1$, the harmonic mean is calculated as:

$$\overline{R}_i = \frac{n+1}{\frac{n}{\overline{R}_i} + \frac{1}{R_i(n+1)}}$$



(a) $\beta_{i,j}$ is the estimate of the time until another connection finishes downloading its current chunk.

(b) Case 1: c_2 can download the remaining bytes in the next request and before c_1 completes its download.

(c) Case 2: c_1 and c_2 need to make one more request each to download the remaining bytes.

Figure 6: Illustration of the scheduler

To estimate the latency, we use an approach similar to that used by TCP to estimate the round trip time, which is performed at the socket API. On every socket read operation on c_i, we measure the current latency rtl_i and estimate $\overline{rtl}_i$ using a weighed moving average:

$$\overline{rtl}_i = 0.8 \cdot \overline{rtl}_i + 0.2 \cdot rtl_i.$$

Further, let $T_{i,j}$ be the time to transmit the j-th chunk over c_i; we have

$$T_{i,j} = s_{i,j}/\overline{R}_i, \tag{1}$$

$T_{i,j}$ is calculated under the assumption that $\overline{R}_i$ does not change during the transmission of the j-th chunk. This assumption is used to simplify the algorithm design. However, our performance evaluation in the next section shows that the scheduling algorithm performs very well in a wireless environment where the quality of the connections changes very quickly.

As we discussed before, mHTTP introduces a delay each time it performs a range request over a connection. This delay equals half of the latency of the connection, *i. e.*, the time that it takes for the HTTP range request to reach the server. Hence, the effective throughput E_i of c_i, during the transmission of the j-th chunk, can be estimated as

$$E_i = \frac{s_{i,j}}{T_{i,j} + 0.5\overline{rtl}_i}$$

Let $T_{i,j} = \alpha_{i,j} \cdot \overline{rtl}_i$. Using eq. (1), we have

$$E_i = \frac{\alpha_{i,j} \cdot \overline{R}_i}{\frac{1}{2} + \alpha_{i,j}}.$$

$\alpha_{i,j}$ should be sufficiently large so that the effective throughput is larger than a certain threshold. In particular, with $\alpha_{i,j} = 20$ we can ensure that $E_i > 0.975\overline{R}_i$. Therefore, we set $T_{i,max} = 20 \cdot \overline{rtl}_i$ as the maximum value for $T_{i,j}$. In the other term, using our algorithm, we do not request chunks larger than $s_{i,max} = 20 \cdot \overline{R}_i \cdot \overline{rtl}_i$, as requesting larger chunks does not bring a significant gain.

Our scheduler needs to request relatively large chunks to minimize the range request overhead. However, requesting large chunks increases the risk that only the slow connection is active at the end. Therefore, a particular attention needs to be paid when assigning the size of the last chunks over each path to guarantee that the downloads of these last chunks complete at *roughly* the same time.

Let L be the size of the file and δ the amount of data that has been already downloaded/requested. Further, let $\beta_{i,j}$ be the time between the j-th scheduling decision for c_i and the estimated time until the other connection completes its current download (see Figure 6 (a)).[4] We consider four cases when determining the size of the next chunk over c_i. We specify the value of $T_{i,j}$ in each of these cases (note that $s_{i,j} = \overline{R}_i \cdot T_{i,j}$). Without loss of generality, we focus on the case where the scheduler wants to determine the next chunk for c_2.

- Case 1 (Figure 6(b)): c_2 can download all remaining $L - \delta$ bytes within a time less than $T_{2,max}$ and before the other connection completes its current chunk download, *i. e.*, $\overline{R}_2 \cdot \beta_{2,j} \geq L - \delta$ and $\overline{R}_2 \cdot T_{2,max} \geq L - \delta$. In this case, $T_{2,j}$ is the time required to download $L - \delta$, *i. e.*, $T_{2,j} = \frac{L-\delta}{\overline{R}_2}$.

- Case 2 (Figure 6 (b)): $\overline{R}_2 \cdot \beta_{2,j} < L - \delta$, but there exists a $T_{2,j} \leq T_{2,max}$ such that

  $$T_{2,j} \cdot \overline{R}_2 + (T_{2,j} - \beta_{2,j}) \cdot \overline{R}_1 = L - \delta.$$

  In this case, the scheduler sets

  $$T_{2,j} = \frac{L - \delta + \beta_{2,j} \cdot \overline{R}_1}{\overline{R}_1 + \overline{R}_2} \tag{2}$$

  to guarantee that c_1 and c_2 complete their last chunk download at the same time.

- Case 3: $\frac{L-\delta+\beta_{2,j} \cdot \overline{R}_1}{\overline{R}_1+\overline{R}_2} > T_{2,max}$, hence, at least three more chunks will need to be fetched over c_1 and

[4] $\beta_{i,j}$ is set to infinity if c_i is the only established connection.

c_2 to download the remaining $L-\delta$ bytes. In this case, the maximum allowed time span is assigned in order to keep the request overhead low, *i. e.*,

$$T_{2,j} = T_{2,max}$$

- Case 4: c_2 is a newly established connection ($j = 1$). As the scheduler is not aware of the quality of the connection, it performs a range request for a relatively small chunk in order to probe the path quality. We denote by *initial* the maximum size of the initial chunk that is allowed to be transmitted over a newly established connection; then $s_{2,1} = \min\{initial, L-\delta\}$. The value of *initial* is an input to our scheduler.

Algorithm 1 summarizes our scheduler design.

Algorithm 1 Advanced Chunk Scheduler

```
Init: END ← FALSE
CONSTRAINT ← (β_{i,j} · R̄_i ≥ L − δ or END)
if j = 1 then
  s_{i,1} ← min{initial, L − δ}
else
  if T_{i,max} · R̄_i ≥ L − δ and CONSTRAINT then
    T_{i,j} ← (L−δ)/R̄_i
  else
    T_{i,j} ← (L−δ+β_{i,j}·R̄_k)/(R̄_i+R̄_k)
    if T_{i,j} ≤ T_{i,max} then
      END ← TRUE
    else
      T_{i,j} ← T_{i,max}
    end if
  end if
end if
```

5.3 Path Management

Path management is a component of multiHTTP that determines which end-points are used for a new connection, meaning which client interface and server IP address should be used. In case of a web page downloads, path management decides which connection is currently idle and thus can be reused to fetch a new object from the same server.

One of the interfaces is labeled as the primary interface (*e. g.*, WIFI). Initially the path manager creates a new connection by performing a range request over the primary interface. The size of the object becomes known to mHTTP upon receiving the HTTP response from the server. After that the path manager decides whether or not it is worth establishing a new connection over one of the unused interfaces (*e. g.*, LTE). The path manager also ensures that mHTTP uses the estimated best performing idle connection first.

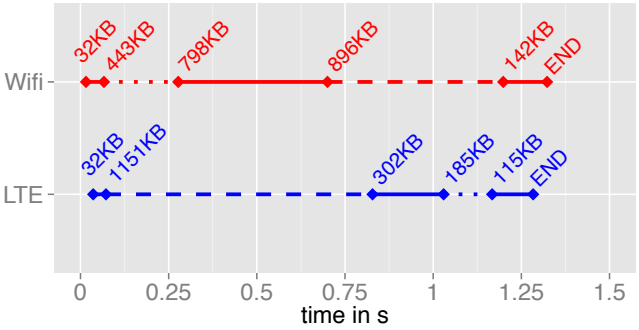


Figure 7: Evolution of chunk sizes for a 4MB file requested by our advanced scheduler over two connections.

5.4 An Illustrative Example

To give more insight into how our algorithm works, we show the evolution of chunk sizes requested over two connections in Figure 7. We use the same testbed configuration as in Figure 5: an mHTTP client connected through WIFI and LTE connections to two servers in a university campus, downloading a 4MB file. We show the results when we use an initial chunk size of 32KB.

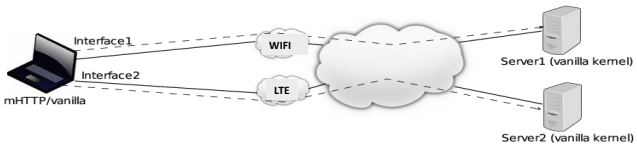
As we observe, mHTTP established the primary connection over the WIFI interface to Server 1 and performs a range request for a 32KB data chunk. As soon as mHTTP learns about the file size, by receiving the HTTP response over the first connection, it establishes a second connection over the LTE interface to Server 2. As it does not know the quality of this newly established connection yet, similarly to the primary connection, it initially requests a 32KB data chunk to probe the path. For each connection after the completion of the initial chunk, the scheduler determines the chunk size that should be requested next.

We observe that the scheduler requests large chunks in the middle of the download and smaller chunks towards the end. Hence, it reduces the range request overhead, by requesting large chunks whenever possible, while ensuring that both connections complete their last chunk download almost at the same time.

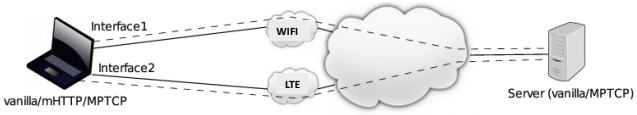
A comparison with Figure 5 indicates that mHTTP with the proposed scheduler performs almost as well as or better than mHTTP with a fixed large chunk size (*e. g.*, 1MB and 2MB). Note that if we want to use the baseline scheduler, then beforehand for any setting, and for any file size, we individually need to determine the best performing fixed chunk size. Moreover, the quality of connections might vary over time and so does the best performing chunk size.

6. EVALUATION

We evaluate the performance of mHTTP coupled with the advanced scheduler in two scenarios as illustrated



(a) Scenario 1



(b) Scenario 2

Figure 8: We study the performance of mHTTP with the advanced scheduler in different scenarios.

in Figure 8. As the default setting, we use mHTTP with a 32KB initial chunk size and with WIFI as the primary interface. In Section 6.1, we provide an overhead analysis for mHTTP. In Sections 6.2 and 6.3, we compare the performance of mHTTP with single-path HTTP and MPTCP. Section 6.4 studies the effect of the initial chunk size on the performance of mHTTP. Section 6.5 investigates the impact of the choice of the primary interface.

We use download completion time as the performance metric in our evaluation. Our measurements are made over 30 rounds where each round consists of a randomized sequence of configuration to account for traffic dependencies and/or correlation from time to time and from size to size. We present the median, $25-75\%$ percentiles (boxes), and dispersion (lines, $5-95\%$ percentiles).

We use servers residing in a university campus and a client equipped with LTE and WIFI network interfaces. The servers are connected to the Internet via a 1Gbps Ethernet interface (*i. e.*, the bottleneck is not at the server). The client device is equipped with two wireless interfaces (WIFI and LTE) that respectively connect to a WIFI network and a cellular network.

TCP *Cubic* [15] is used as the congestion controller at the servers. To provide a fair comparison between MPTCP and mHTTP, we use uncoupled congestion control with Cubic for MPTCP. Uncoupled Cubic represents the case where regular TCP Cubic is used on the subflows. We set the maximum receive buffer to 6MB to avoid potential performance degradation to MPTCP [8]. The proposed standard MPTCP, which uses coupled congestion control, would perform quite a bit worse than the version of MPTCP used in our study [9].

6.1 Overhead Analysis of mHTTP

mHTTP experiences a performance degradation each time a connection performs a range request. We evaluate this degradation by measuring the download completion time of a file over a single path connection using mHTTP and HTTP. We present the results for different types of connections (*i. e.*, WIFI as well as LTE) and for different file sizes. The results are depicted in Figure 9. We observe that the overhead is negligible and increases the download time less than 5% in most cases. In particular, for file sizes smaller than the initial chunk size, the overhead is zero as the file is downloaded within a single request. For larger file sizes, there is a small degradation due to sending multiple range requests over the connection. However, as our scheduler tends to request large chunks, this degradation is small.

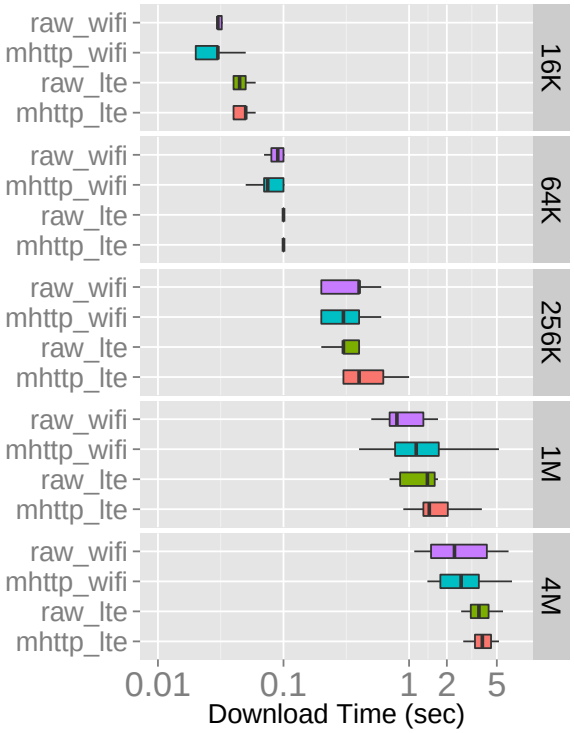


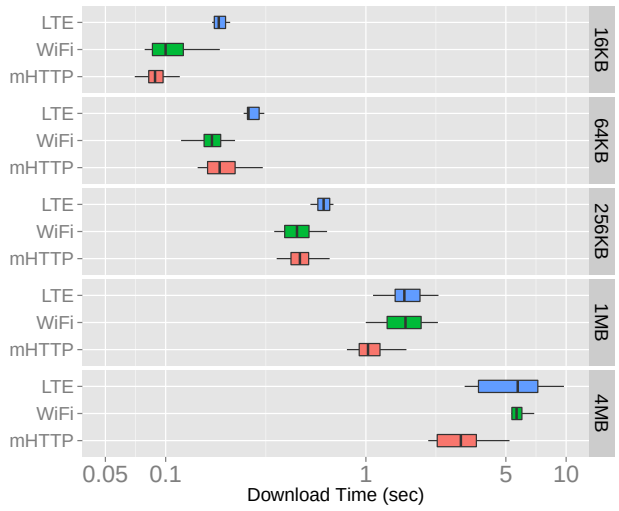
Figure 9: Overhead analysis: HTTP vs. mHTTP over a single connection.

6.2 mHTTP vs Single-path HTTP

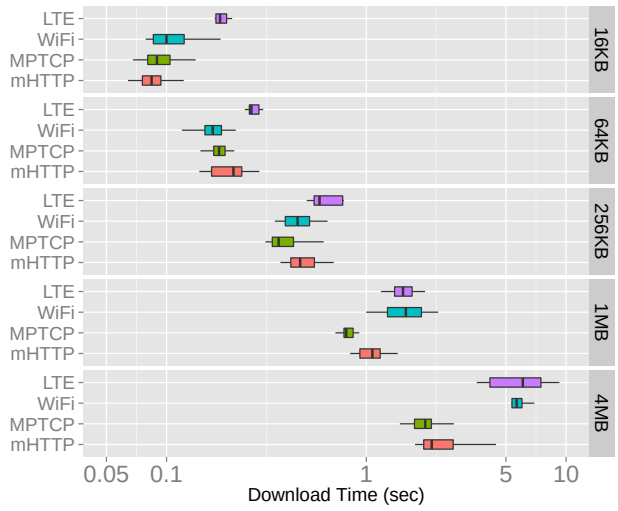
Figure 10(a) compares the performance of single-path HTTP and mHTTP requests carried out on the 2-server scenario illustrated in Figure 8(a). Measurements are conducted for various file sizes, *i. e.*, 16KB, 64KB, 256KB, 1MB, and 4MB.

We see that the link characteristics of LTE and WIFI are different from those we observed in Figure 5. This is because these two sets of measurements are performed on different dates. However, we observe that mHTTP with the advanced scheduling provides a relatively greater gain for 4MB file downloads than mHTTP with baseline scheduling using large chunks.

Moreover, we observe that for small files, mHTTP



(a) Scenario 1: single-path HTTP vs. mHTTP



(b) Scenario 2: single-path HTTP vs. MPTCP vs. mHTTP

Figure 10: Performance comparison of mHTTP with the advanced scheduler vs single-path HTTP and MPTCP.

performs as well as HTTP over the best path and that it provides a significant gain in case of relatively large file downloads (*i. e.*, 1MB and 4MB). The results show that mHTTP with advanced scheduling performs efficiently.

6.3 mHTTP vs. MPTCP

Now, we compare the performance of mHTTP and MPTCP. The experiment is performed in the single server scenario illustrated in Figure 8(b). We also show the performance of single-path HTTP over WIFI and LTE connections.

To avoid confusion, it is important to note that mHTTP and MPTCP do not interfere with each other in our measurements. More precisely, MPTCP is disabled when mHTTP is operating and mHTTP is not present while the MPTCP experiments are carried out.

The results are presented in Figure 10(a). We observe that MPTCP performs slightly better than mHTTP. However, the difference is small, especially for 4MB file downloads. Besides, MPTCP uses uncoupled Cubic which is not part of the MPTCP standard. The proposed standard MPTCP, which uses coupled congestion control, would perform quite worse than what we observe here [9]. Furthermore, as we discussed before, MPTCP requires modifications at both the server and client, while mHTTP provides similar performance with changes that are applied at the client side only.

Figure 11 illustrates the fraction of traffic transmitted over WIFI and LTE using mHTTP and MPTCP. We observe that MPTCP transmits a larger fraction of traffic over the LTE connection, especially for small file sizes (*e. g.*, 256KB), which explains why the performance of MPTCP here is better. The difference is negligible for 4MB downloads.

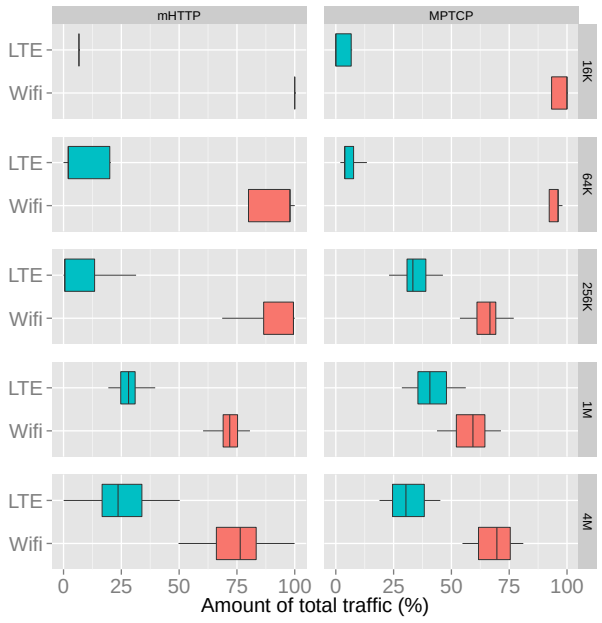


Figure 11: Fraction of traffic transmitted over WIFI and LTE for mHTTP and MPTCP.

6.4 The Effect of Initial Chunk Size

In the previous section, we used an initial chunk size of 32KB. In this section, we study the effect of the initial chunk size on the performance of mHTTP. The results are found in Figure 12, where we consider 3 different initial chunk sizes, *i. e.*, 16KB, 32KB, and 64KB. The measurement is conducted for various file sizes, *i. e.*, 16KB, 64KB, 256KB, 1MB, and 4MB.

We observe that a chunk size of 64KB, yields to the

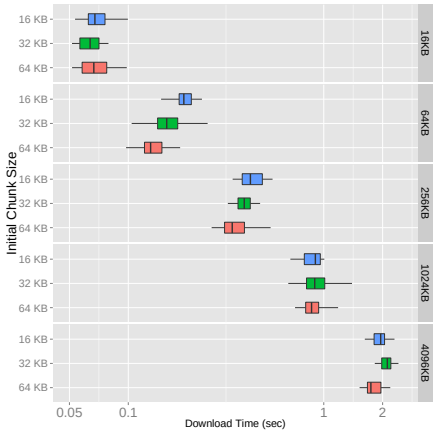
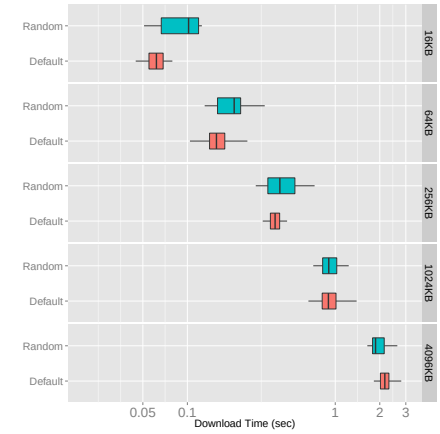
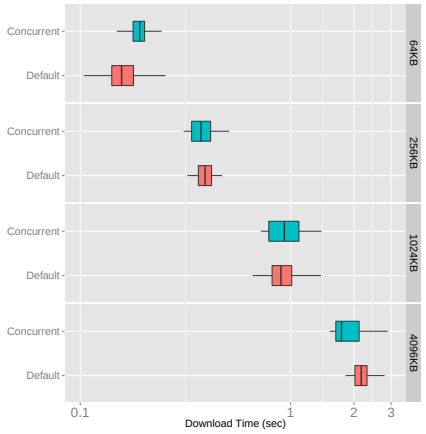


Figure 12: The difference made by the size of an initial chunk



(a) Random-first vs. WIFI-first (default)



(b) Concurrent vs. WIFI-first (default)

Figure 13: The impact of the initial interface (initial chunk size: 32KB)

best performance. This is especially true in case of 64KB file downloads, as initial chunk sizes of 16KB and 32KB introduce at least 2 requests, while only one request is required with an initial chunk size of 64KB. This is not surprising; using larger chunks at the beginning can reduce the total number of requested chunks over a connection and hence reduce the range request overhead. However, it is important to mention that we do not have prior information about the quality of a connection, and that we can learn about the quality of a connection only after the reception of the first chunk. If the performance over a connection is poor, then a large initial chunk can be harmful as it will have to be downloaded over that connection.

Consequently, we set the initial chunk size to the intermediate value of 32KB as a compromise between the benefits and the risks. This is, however, an arbitrary choice and further study is needed to determine the optimal initial chunk size.

6.5 Implication of the Primary Connection

Now, we study the effect of our choice of the primary interface on the performance of mHTTP. The default configuration of mHTTP is to establish a WIFI connection first on the basis that LTE may introduce an additional economic cost to users. However, it is of interest to users and developers how such a decision affect the performance of mHTTP. For the evaluation, we compare this default method (WIFI-first) with two other methods. The study is performed for mHTTP with 32KB as the initial chunk size.

The first comparison is made with the random-first method and the result is illustrated in Figure 13 (a). The measurement is again carried out by downloading 5 different files with various sizes. According to the result, the impact of the random primary connection is literally random. It mainly affects the performance of small file downloads. For large files, the difference is not significant.

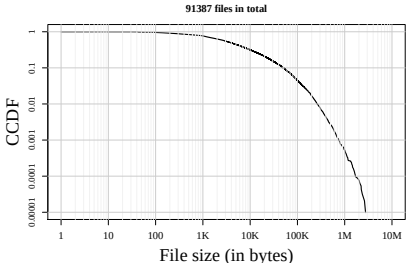
The second comparison is made with the concurrent method, *i. e.*, starting multiple connections at the same time. In this case, mHTTP establishes both connections simultaneously, performing a range request over these connections. Therefore, we save one round trip time compared to our default setting in which we establish the second connection only after receiving the range request response, and hence the information about the file size, over the primary connection. This experiment is conducted only to see the effect of such an approach and to compare it with the default method. In fact, the concurrent start of multiple interfaces may cause a superfluous request and connection establishment.

Figure 13 (b) depicts the result of this experiment. Note that the 16KB file is not used in this measurement since it is smaller than the default initial chunk size. The result is approximately the same as with the random vs. default experiment and does not show any additional gain of starting with multiple interfaces concurrently.

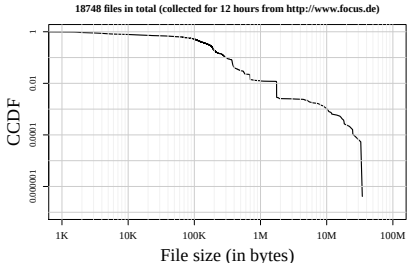
The results of this section show that the choice of the primary connection has little effect on the performance of mHTTP for single file downloads.

7. WEBSITE DOWNLOAD

So far, our evaluation of mHTTP has been carried out based on the download of a single object (file). Outcomes of the experiments have shown that the benefit of using mHTTP increases as the size of the object is larger. However, a recent traffic measurement study [16] and our own analysis (Figure 15) show that the majority of web objects are smaller than 1MB. Figure 15(a) is the Complementary Cumulative Distribution Function



(a) Alexa top-1000 domains



(b) *http://www.focus.de/*

Figure 15: CCDFs of file sizes

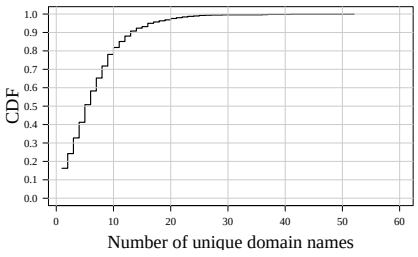


Figure 16: No. of linked domains in CDF (Alexa top-1000 domains)

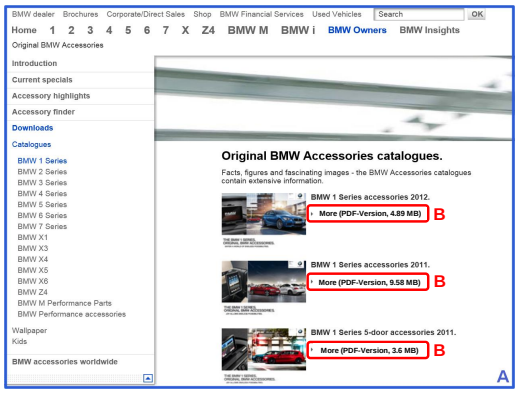


Figure 14: An example web page that links to large files

(CCDF) of embedded object sizes in the index pages of the Alexa top-1000 domains and Figure 15(b) is that of web crawling results from the German online news site *focus.de*. This phenomenon of small file domination inspires us to look at the gain or loss (due to the overhead) of downloading web pages from well-known web sites (**real world experiments**).

One might raise the question why files larger than 2MB are not observed in Alexa top-1000 domains, yet are observed in *focus.de*. Indeed, the answer to this question is important for understanding the benefit of using mHTTP. To explain this, we use a screenshot of a web page from a well-known automobile manufacturer's website as an example (Figure 14). During the page rendering on a web browser, embedded images and text (*i. e.*, small content) are automatically downloaded (*i. e.*, the blue box marked as A in the figure), but large files are often downloaded upon user's click activity (*i. e.*, links within red boxes marked as B in the figure). Thus, Figure 15(a) represents the file transfer during the web browsing, where Figure 15(b) embraces the portion of large files from which mHTTP mainly gains benefits.

7.1 Functional Extension

Accessing a web page (*i. e.*, a HTML file) typically triggers subsequent downloads of multiple linked web objects, *e. g.*, images (jpg, gif, png), cascading style sheet (css), and java script (js) files. Thus, we implement a new feature in mHTTP that enables it to handle multiple web objects that are subsequently requested after fetching an HTML page.

Our major effort in developing this functional extension is to minimize buffer re-initialization overhead since the accumulation of small losses may largely affect the overall performance if a web page is comprised of many small files.

Usually, modern GUI web browsers and download managers send HTTP-requests using multiple concurrent processes (aka, multithreading) in order to better deal with multiple web objects embedded in a single web page. To this end, we do not use any vendor-specific optimization methods in our measurements. Thus, objects hosted at the same domain are requested one after another over the same connection, *i. e.*, the basic operation of wget when *–page-requisites* or *-p* option is set. It is important to point out that an HTML page may embed files from different domains, thus the client (the web browser) establishes several connections (*i. e.*, several mHTTP instances) for accessing a web page. Figure 16 shows that more than 75 % of the web sites among *Alexa* top-1000 domains embed at least two unique domains in their front pages.

7.2 Categorization of Web Pages

The performance of downloading web pages using mHTTP is significantly affected by the size and number of embedded files. Therefore, the performance evaluation of mHTTP on random web pages cannot be seen as the general result. Consequently, we divide web pages into three different categories according to the size of the embedded objects and evaluate mHTTP over each category separately. The three categories are many small files (MSF), some large files (SLF), and many large files (MLF). The categorization is mainly based on the fraction of large files (>1MB) embedded in the page. The categorization scheme and the number of samples are listed in Table 1.

For our experiments, these pages are manually verified and selected to meet certain requirements. First, the content of a web page should not change too frequently since our measurement is performed repeatedly in different time periods. Second, the majority of embedded objects in the web page should not be dynamically generated on every access request. Third, the majority of web objects on the web server must be accessible through HTTP, *i. e.*, no HTTPS-only servers, without authentification. As a result, we select 24 web pages in total.

	MSF	SLF	MLF
% of large files (> 1 MB)	0 %	< 1 %	> 1 %
Median(% of large files)	0 %	0.59 %	2.5 %
No. of samples (pages)	9	6	9

Table 1: Web page categorization scheme

7.3 Performance Evaluation of mHTTP

Unlike our previous experiments that were performed in testbeds in the US, the web workload experiment is conducted in a German university campus using a laptop equipped with an Ethernet interface and a WIFI interface connected to the Internet via two different access networks. The Ethernet link and the WIFI link are respectively throttled to 15Mbps and 10Mbps. We use mHTTP with an initial chunk size of 32KB.

Figure 17(a) and (b) illustrate the performance gain and loss of mHTTP in web page downloads compared to HTTP over the better performing interface (Ethernet in our experiment). From the results we observe that there is essentially no gain for websites in the MSF category, a gain of around 5% for those in the SLF category, and around 18% for websites in the MLF category. The lack of a performance gain for the MSF category is not surprising since web pages that belong to this category do not embed any large files.

Figure 17(b) depicts the performance gain achieved for each individual website in each category. The web page, alias *A* in the MSF category in Figure 17(b) shows good performance even without any large files. On closer investigation, we find that the web page embeds many files that are smaller than 1MB, but much larger than the initial chunk size.

For websites in the SLF and MLF categories, we observe a significant performance gain using mHTTP. This is especially true in the case of SLF where the fraction of large files is less than 1 %, one of the web pages (*c* in Figure 17(b)) shows more than 10 % gain using mHTTP. Considering the small difference between the percentage of large files in SLF (median 0.59 %) and in MLF (median 2.5 %), the difference in the median gain between the two categories is significantly high. In particular, web page *6* embeds around 2.5 % of large files and has an median file size of around 170 KB and exhibits almost 50 % of the performance gain by using mHTTP. This shows again the well-known fact that while there may be many small files, large files are responsible for the bulk of the total value traffic in the Internet. Hence, multipath approaches such as mHTTP can bring significant decreases in download times to users.

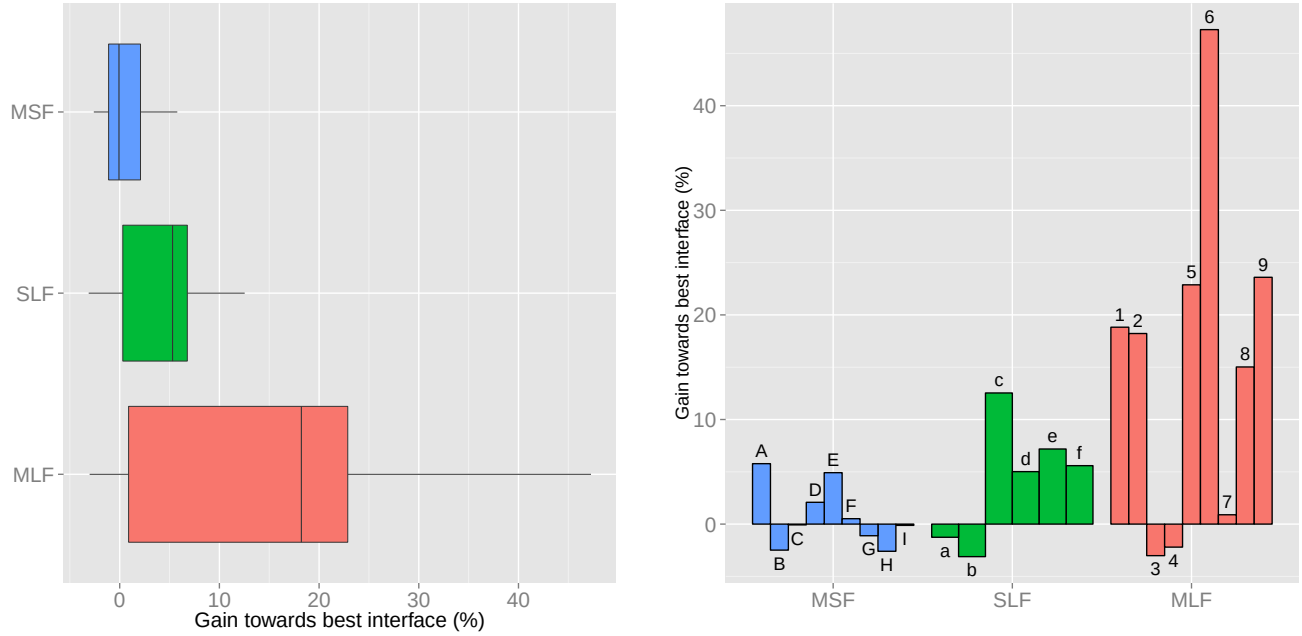
8. DISCUSSION & FUTURE WORK

This section elaborates on limitations of mHTTP and discusses our on-going and future work with regard to its further development.

Verification of the content identity It is, in principle, possible that the two files located by the same URL (Uniform Resource Locator) are hosted on different servers (*e. g.*, by CDN) and are not identical, *e. g.*, due to the locale or time difference. Two verification methods are currently considered. The first method is to ask for a few more bytes than the chunk size in the initial HTTP request and to compare these bytes with the beginning of the next chunk transferred over the second connection. The second method is to use the Entity Tag (ETag) of HTTP. The ETag is a hash value of a web page that is delivered from the server on client's demand. Once the mismatch is recognized, mHTTP can fall back to regular HTTP without interrupting the file transmission.

GUI browser support The current development status of mHTTP is a proof-of-concept prototype. To operate on fancy GUI web browsers, *e. g.*, Firefox, Chrome, or Internet Explorer, mHTTP needs to understand specific behaviors of such GUI web browsers. It is crucial to note that providing an appropriate intelligence for mHTTP is still more cost-beneficial than writing and managing tens of thousands of lines of source code for various web browsers. We are currently working towards implementing a subset of socket API functionalities for measuring the performance of specific web services on the GUI environment, but developing mHTTP to the product level is out of this project's scope. We are planning to release the source code of mHTTP in the form of an open-source project, thus expecting mHTTP to further evolve with help of interested developers and researchers.

HTTPS support Given the fact that many web objects are delivered over HTTPS and it is expected to grow due to the growing privacy concerns, extending capabilities of mHTTP to HTTPS is one of the most important future development tasks. For this, mHTTP needs to establish a transparent communication channel with the Secure Socket Layer (SSL) as is done in a similar way between an application and mHTTP, *i. e.*,



(a) Categories

(b) Individual sites

Figure 17: Performance evaluation of web page downloads

ensuring transparency of mHTTP to SSL.

Comparison In this paper, we mainly compare the performance of mHTTP with that of MPTCP and normal HTTP. Although there are other approaches that aim to increase the performance of the data transfer within the existing Internet infrastructure, comparing mHTTP with these other approaches is not a straightforward task to carry out since many of those solutions are not available to the public or even implemented. Currently, we are working on designing a testbed platform that can be used to compare the performance of different mechanisms, *e. g.*, Google's SPDY protocol [6] and the Download Booster of Samsung Galaxy S5, on a fair comparison environment.

Mobility study In this paper, we mainly focus on the performance enhancement that mHTTP provides in term of download completion time. The robustness is another important issue that should be investigated in our study. In particular, as mHTTP establishes multiple connections over multiple interfaces, it should be able to benefit from this path diversity to mitigate the effect of the changes that occur in the network due to the mobility of the users (*e. g.*, when a user walks away from a WIFI access point or switches from one WIFI network to another). We are currently setting up a testbed to study the performance of mHTTP in such a setting.

9. RELATED WORK

mHTTP is designed to utilize opportunities created by various types of network plurality and diversity, *i. e.*, multiple network interfaces, replicated Internet content, various access technologies, different service providers, and fully and/or partially disjoint paths between end-hosts. Studies on benefits of utilizing network diversity have started more than a decade ago. One of the earliest publications in this research area is the study of Rodriguez and Biersack [26]. The paper explores the concept of the parallel access to replicated content and evaluates benefits of the mechanism based on the simulation atop a web browser.

The support of multiple end-host identifiers, *i. e.*, IP addresses, is a unique feature of the Stream Control Transmission Protocol (SCTP) [17, 29] that makes the concurrent multipath transfer available. However, SCTP requires a modification at existing applications and it is not an easy task to cope with in the TCP dominating Internet.

Kaspar [19, 18] thoroughly studies the path diversity in the Internet and discusses use cases on the transport layer as well as on the application layer. His work and mHTTP have many features in common except that his work is limited on a single client/server scenario and it does not take chunk scheduling into the design consideration. Evensen *et al.* [12] emulate the same technique within the scope of adaptive video streaming services and evaluate the increase of the performance. Their performance evaluation of static segments, *i. e.*, the same segment size, versus dynamic segments is particularly

interesting for us since we also implement a dynamic chunking algorithm to improve the performance of the initial prototype that uses static chunk sizes.

One of the closest siblings of mHTTP (in terms of the philosophy underlying their technical approach) is MPTCP [1, 8] which is an extension of the regular TCP that enables a user to spread its traffic across disjoint paths. Although MPTCP focuses on the path diversity between a single server and a single receiver and requires the modification at both end-hosts, the fundamental idea behind these two protocols is the same. A handful of evaluation studies on MPTCP witness [5, 9, 22, 25] the benefit of utilizing a rich diversity of paths and interfaces.

Socket Intents [27] is a multi access technology that aims to utilize the availability of multiple and heterogeneous network interfaces and to improve the quality of web content delivery based on the pre-configurable decision policies.

10. CONCLUSION

In this paper, we presented the design, implementation, and evaluation of mHTTP, a concurrent HTTP data transfer mechanism based on various types of diversity existing in today's Internet. Furthermore, we described our effort on the enhancement of mHTTP by developing a scheduling algorithm that heuristically determines the chunk sizes during runtime.

Our performance evaluation proved that mHTTP can efficiently aggregate the available bandwidth to an end-device without any changes at server-side system elements and at any application (*i. e.*, modifications only at client-side socket APIs). As a part of the evaluation, we compared the performance of mHTTP (multi-source mechanism) with that of MPTCP (*cf.* single-source mechanism) on a single data-source environment. We observe that mHTTP exhibits similar performance as MPTCP for downloading relatively large objects. Hence, we consider mHTTP to be a viable alternative for MPTCP for HTTP traffic.

Finally, we conducted the real-world experiment on 24 entry pages of well-known web sites by classifying them into 3 different categories based on the portion of large files embedded in the web pages. The purpose of this experiment was to show that mHTTP does not harm the web browsing (a set of small objects) even though mHTTP targets at large file transfers such as mobile applications (apps) and software, file sharing (OCHs), and high definition multimedia files. The result of this experiment was beyond our expectation and showed that mHTTP can gain a high benefit from web browsing even if the web page embeds a very small portion (such as 1 %) of large files.

11. REFERENCES

[1] Architectural Guidelines for Multipath TCP Development.

[2] Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *Proc. IEEE INFOCOM* (2012).

[3] Ager, B., Schneider, F., Kim, J., and Feldmann, A. Revisiting cacheability in times of user generated content. In *Proc. IEEE Global Internet* (2011).

[4] Almesberger, W., et al. Linux Network Traffic Control - Implementation Overview, 1999.

[5] Barre, S., Paasch, C., and Bonaventure, O. Multipath TCP: from Theory to Practice. In *IFIP NETWORKING 2011*. Springer, 2011.

[6] Belshe, M., and Peon, R. SPDY Protocol.

[7] Breslau, L., Cao, P., Fani, L., Phillips, G., and Shenker, S. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM* (1999).

[8] C. Raiciu and C. Paasch and S. Barre and A. Ford and M. Honda and F. Duchene and O. Bonaventure and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP.

[9] Chen, Y.-C., Lim, Y.-S., Gibbens, R. J., Nahum, E. M., Khalili, R., and Towsley, D. A Measurement-based Study of Multipath TCP Performance over Wireless Networks. In *Proc. ACM Internet Measurement Conference* (2013).

[10] Chen, Y.-C., Towsley, D., and Khalili, R. MSPlayer: Multi-Source and multi-Path LeverAged YoutubER. *arXiv preprint arXiv:1406.6772* (2014).

[11] Cohen, B. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems 2003*.

[12] Evensen, K., Kaspar, D., Griwodz, C., Halvorsen, P., Hansen, A., and Engelstad, P. Improving the Performance of Quality-adaptive Video Streaming over Multiple Heterogeneous Access Networks. In *ACM Multimedia Systems* (2011).

[13] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. In *RFC 2616*.

[14] G. Tian and Y. Liu. Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming. In *Proc. ACM CoNEXT* (2012).

[15] Ha, S., Rhee, I., and Xu, L. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42* (2008).

[16] Ihm, S., and Pai, V. S. Towards Understanding Modern Web Traffic. In *Proc. ACM Internet Measurement Conference* (2011).
[17] Iyengar, J. R., Amer, P. D., and Stewart, R. Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Trans. Networking 14* (2006).
[18] Kaspar, D. *Multipath Aggregation of Heterogeneous Access Networks*. PhD thesis, University of Oslo, 2012.
[19] Kaspar, D. Multipath Aggregation of Heterogeneous Access Networks. *ACM SIGMultimedia Records* (2012).
[20] Maier, G., Feldmann, A., Paxson, V., and Allman, M. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proc. ACM Internet Measurement Conference* (2009).
[21] Maierr, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., and Schneider, F. Enriching Network Security Analysis with Time Travel. In *ACM Comp. Comm. Review* (2008).
[22] Nguyen, S. C., and Nguyen, T. M. T. Evaluation of Multipath TCP Load Sharing with Coupled Congestion Control Option in Heterogeneous Networks. In *IEEE Global Information Infrastructure Symposium* (2011).
[23] Poese, I., Frank, B., Ager, B., Smaragdakis, G., and Feldmann, A. Improving Content Delivery using Provider-aided Distance Information. In *Proc. ACM Internet Measurement Conference* (2010).
[24] Popa, L., Ghodsi, A., and Stoica, I. HTTP as the Narrow Waist of the Future Internet. In *ACM HotNets* (2010).
[25] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM* (2011).
[26] Rodriguez, P., and Biersack, E. W. Dynamic Parallel Access to Replicated Content in the Internet. *IEEE/ACM Trans. Networking 10*, 4 (2002).
[27] Schmidt, P. S., Enghardt, T., Khalili, R., and Feldmann, A. Socket Intents: Leveraging Application Awareness for Multi-Access Connectivity. *Proc. ACM CoNEXT* (2013).
[28] Seedorf, J., and Burger, E. Application-Layer Traffic Optimization (ALTO) Problem Statement. In *RFC 5693*.
[29] Stewart, R. Stream Control Transmission Protocol. In *RFC 4960*.
[30] Alexa. `http://www.alexa.com/`.
[31] FlashGet. `http://www.flashget.com/`.
[32] JDownloader. `http://jdownloader.org/`.