The web browser as distributed application server: towards decentralized web applications in the edge

Kristof Jannes
imec-DistriNet
KU Leuven, Belgium
kristof.jannes@cs.kuleuven.be

Bert Lagaisse
imec-DistriNet
KU Leuven, Belgium
bert.lagaisse@cs.kuleuven.be

Wouter Joosen
imec-DistriNet
KU Leuven, Belgium
wouter.joosen@cs.kuleuven.be

ABSTRACT

Web applications are evolving to a decentralized, client-centric architecture in which browsers need to be able to put the user back in control of his personal data, need to be able to operate in disconnected settings, and need to offload the web server as much as possible.

This paper presents a set of key application scenarios and trends in different business domains that require a more client-centric and data-centric web middleware for decentralized, peer-to-peer web applications in the edge. We define a set of key requirements for data operations in such middleware and motivate them with the application cases.

This paper further discusses the current state and limitations of the browser as platform for peer-to-peer communication and complex decentralized applications with shared data. We conclude with a performance assessment of our first prototype middleware for client-centric and data-centric peer-to-peer web applications.

1 INTRODUCTION

The Web has changed a lot since the original proposal for the World Wide Web thirty years ago. Static web pages using basic HTML evolved to become dynamic, server-side applications and later have become fully functional single-page web applications using JavaScript on the client-side. This enables the offloading of visualization logic and functionality from the server to the client, and thus freeing up resources on the server, enabling better scalability. Moreover, in today's always-connected environment, tunnels and airplanes should not discontinue fluent operations of web applications. Browsers and client-side web technology also offer more and more capabilities to enable fully client-side web applications that can operate in a disconnected setting (e.g. extended local storage with query technology and service workers that support offline operation). As such, web applications are replacing native programs in many places.

However, the basic paradigm of the web and the browser is still server-centric. The key data is stored, served, processed and analyzed on central servers owned by the service provider. As said by Tim Berners-Lee, the founder of the web: over the years, we've lost control of our personal data [2]. To regain control, the web should evolve to a decentralized network, where data can be stored in places under control of the user. To come to a fully decentralized web, living in the edge, browsers need to shift from the client-server paradigm to a peer-to-peer (P2P) approach. This kind of decentralized web application architecture should be supported by both the browser and the client-side web middleware. This client-side platform will be responsible to let the different clients connect to a decentralized fabric of clients in the edge. Moreover, each client-side node will be responsible for key middleware services that were once the roots of the first application servers in the 90's: 1) coordinating consistency over distributed storage, 2) executing data-centric operations and key business logic, and 3) controlling data access.

The contribution of this paper is a case-study driven motivation and analysis of a more client-centric and data-centric web middleware for decentralized web applications in the edge. Based on the three-fold motivation of disconnected situations, offloading web servers and regaining control of personal data, we put forward the need for more advanced client-centric and data-centric web middleware that supports decentralized web applications in a P2P network of browsers. We focus on data-centric operations such as data storage, data replication and synchronization, well-scoped data sharing and access control, as well as secure and privacy-aware data queries and data analysis.

This paper is structured as follows. In Section 2 we first present and discuss a set of key application scenarios and trends that motivate our need for decentralized client-centric web applications. We then define a set of key requirements that must be supported by the underpinning data-centric web middleware. Section 3 presents an analysis of the current state of web browsers to assess to which extent such decentralized web applications can already be supported. In Section 4 we assess a first prototype of our middleware that already supports P2P data synchronization to reduce the load on web servers and operate in disconnected settings. We conclude in Section 6.

2 MOTIVATION AND REQUIREMENTS

Even now that web applications are becoming fully functional and stand-alone programs, they still need a central

server to store and synchronize the user data. This is needed because data is often shared by multiple people, and the latest version needs to be available to all of them. Even applications that only contain data of a single user, often need to synchronize this between all the devices of that user (e.g. laptop, tablet, phone). Storing the data on a central server has several disadvantages. Web applications always need a working internet connection to synchronize the data and share their work with others. For data that is shared across many users, the web server needs enough capacity to ensure prompt synchronization, especially for web applications that allow users to work together interactively. The data on the central server is under control of the provider of the web application. The provider might make security mistakes that leak your data to the internet. Even when the data is secure, the provider has access to it and can look into it at any time.

Motivating scenarios. Many web applications can benefit from a more decentralized and P2P approach. Multi-player games in particularly benefit from the decreased latency by communicating directly between browsers. Next to low-latency benefits for games, many enterprise level applications can benefit from the high availability and confidentiality that a decentralized web application can bring. This section will present three such enterprise level applications as motivating examples for the use of a more decentralized web, next to the current centralized approach. The examples are based on real life case studies from our applied research projects with industry.

The first application, eWhiteboard, is a shared whiteboard which can be used by participants at a business meeting. All people are in the same room, using their laptop or tablets to access a web application where they can draw and write ideas during their meeting. Everyone is immediately able to see what others draw and can actively participate in the discussion. Since they are all in one room, there is no reason to use a central server which is under control of the company that created the whiteboard web application. Instead, all communication can happen P2P between the users' browsers over a local ad-hoc network. This even makes it possible to hold such a meeting in cars, trains or even airplanes, since there is no reliance anymore on the internet. The second benefit is that the ideas, which might be of strategical importance to the company, never leak to third parties who are not present at the meeting. All data is stored locally in the browser and is only synchronized between the participants of the meeting. Even the company of the whiteboard application cannot access this data. However, each participant might want to share the data with colleagues and store it to a company server.

The second company is eDesigners. It provides a multi-tenant web application for graphical templates. Templates can be edited by several users at the same time, even when offline. The base of the templates is provided by eDesigners itself, while the customer companies make their own customization's on top of it. The base template of eDesigners is accessible to all payed users. The customization's that a company made on top of it can only be accessed by employees of the same company or department within the company.

The third example is about eWorkforce, a company that provides technicians to install network devices for different telecom operators at their customers' premises. The company has two kinds of employees: the help desk operators in the office that accept customer calls and plan technical interventions by technicians; and technicians on the road that go from customer to customer to install or repair network infrastructure. The technicians need to check their work plan, enter used materials and indicate the status of a particular intervention. Since they are always on the road, sometimes working in cellars, internet is not always available. Yet, they must still be able to complete their jobs and synchronize their devices with the back-office once they are back online. When multiple technicians are working on the same job, they can synchronize their devices with each other. This way, no used material is accidentally entered twice, or worse, forgotten because one employee thought another had already entered it. The operators in the back-office normally have an internet connection to the main server. In case of network disruptions, a large company cannot tolerate to shut down business for several hours. Instead they should be able to continue working as usual. To prevent conflicts between the operators, e.g. assigning the same technician to different jobs at the same time, they can still synchronize their updates to each other. The LAN network in the office will most likely be intact. They can keep up-to-date with the latest changes using P2P communication between the operators' devices.

Analysis of the requirements. In this paragraph we present a set of key requirements and features for decentralized, client-centric web middleware. In our opinion, the basic architecture of such middleware should be fully based on standard browser technology and its JavaScript programming environment, and should not involve any plugins or add-ins to the browser. The basic, core functionality should focus on data-centric operations such as data storage, data replication and synchronization, data sharing as well as secure and privacy-aware data queries and data analysis. The middleware should first be able to synchronize updates promptly to all other clients and solving conflicts in the data automatically. These synchronizations might happen via a central server in the back-office, or using direct browser-to-browser communication. The latter is especially important for privacy sensitive SaaS applications used by any company that doesn't always want its data on a third-party server. At last, there should even be a possibility to connect multiple clients over a local ad-hoc P2P network to share updates in offline

situations (e.g. airplanes). As such, we distinguish a first set of basic requirements and features:

(1) Operate continuously and without disruptions in a disconnected situation using local storage.
(2) Support P2P synchronization between users when the server is not available or overloaded. E.g. two designers working on the same template in an airplane, both using their tablet, should be able to synchronize.
(3) Efficient data synchronization on mobile connections.
(4) Synchronization of data items with interactive timing constraints. E.g. when a first user edits the color of an item on the whiteboard, another user sitting next to him should receive this update promptly within acceptable timing according to usability guidelines [8].

State of the art frameworks have already support for these basic requirements. However, there are also a set of more advanced requirements related to privacy-aware and secure data sharing and data analysis:

(1) *User-centric access control with selective sharing and synchronization of data items.* Each user should be able to determine who can see what from the data the user owns. In case of the eDesigners application for concurrent editing of templates, a designer needs to be able to select which templates are shared with who.
(2) *Support for distributed select queries.* Users might request specific data objects of another user using a typical select query. When a certain user shares a drawing with another user, the drawing's document identifier appears in a list of shared documents. The other user will query the drawing document using the data object identifier from the first user.
(3) *Decentralized data processing and analysis.* In case of the collaborative whiteboard application, the developer of the application, who is not controlling the data on an application server, might want to know the following statistics: how many drawings is the average user storing, how many drawings does a user create per month, with how many people does an average user collaborate on a drawing, how complex and large is an average drawing, what kind of colors and forms are mostly used in a drawing, ...

Based on these requirements we now analyze the current state of the browser as platform for decentralized, client-centric and data-centric web applications.

3 STATE OF THE BROWSER

This section will go into the technologies that are present in browsers to enable a decentralized, offline web. It first covers WebRTC, which enables browsers to directly communicate with each other. Then we explain the JavaScript threading model based on the event loop and the possibility to use multiple threads. We explain how web applications can be used offline. We end with the security model of the browser.

WebRTC. WebRTC (Web Real Time Communications) [1] enables direct, P2P communication between browsers. Communication is coordinated by the exchange of control messages over a separate signaling channel. This has serious consequences, because two browsers that want to connect to each other directly already need an indirect connection to each other. The most user friendly option is to use a central server for this. The browsers can send the control messages to each other using normal HTTP requests or WebSockets to the server, which will forward the message to the right browser. Once the WebRTC connection is setup, the signaling channel is no longer needed, and the browsers can communicate with each other directly. The requirement of having a central server for setting up the connection is a problem in an offline situation, where you want to setup a local P2P network. As a solution, it is possible to do the signaling manually. One can use QR-codes to encode the control messages. When initiating a P2P connection, you let the web application generate a new QR-code which is shown on the screen. The other party that you want to connect to scans this code with his device. The web application on his device generates a QR-code as a response. You scan the QR-code and the connection can be established. You can now repeat the process for all devices you want to connect with. An existing WebRTC connection can also be used as a signaling channel. So once you connect to a device, you can also setup a P2P connection to its peers.

Next to the need of a signaling channel, Network Address Translation (NAT) and firewalls pose additional problems. NAT is used to overcome the shortage of IPv4 addresses. NAT creates a local network where each client on that network has its own local IP address. The NAT-box has a public IP address and forwards all requests coming from a client to the internet using its public IP address and remembers which client made the request. This means that the IP address that a client knows is not its public IP address, but only a local IP behind the NAT-box. Browsers using WebRTC on a different network cannot connect to you using your local IP address. Session Traversal Utilities for NAT (STUN) is used by a browser to discover its public IP address. Again, a central server is needed (STUN-server) to setup a P2P connection. Firewalls can make it impossible to setup a real P2P connection, essentially because the browser needs to accept connections from outside on some random port, which is not always permitted. The solution for this is using Traversal Using Relays around NAT (TURN), which will essentially put a relay-server in between the P2P communication. This is not a real browser-to-browser communication and therefore has almost none of its advantages. It is provided as an option to make WebRTC connections more reliable. As long as

you have access to a TURN-server you can setup a WebRTC connection, possibly via a TURN-relay.

The JavaScript event loop. JavaScript was once made for building interactive and more complex user interfaces than were possible using plain HTML. It uses only a single thread with one event loop and a task queue to hold tasks, which get executed one by one. That single main thread is responsible for everything the web application needs to do: drawing the page on the screen, executing the JavaScript code of the web application and background data-synchronization to peers. With the evolution of JavaScript, new features are added and multiple queues exists for the event loop to choose from with different priorities. One such queue is the microtask queue, which handles Promises. A Promise is a JavaScript primitive to allow you to write clear concurrent code, without relying on callbacks. The browser will always first empty the microtask queue, and only then return working on the main task queue. This can delay the execution of tasks for a long time when too many Promises are generated, especially when those Promises also create new Promises. This leads to temporary starvation of the task queue.

WebWorkers [6] are separate threads in the browser with their own event loop. They can therefore run in parallel with the main thread. The main thread is still the only thread that can update the user interface. This allows to do heavy computations on a separate worker thread, so the user interface stays responsive, which leads to a better user experience. The only possible communication between the main thread and a worker is using message passing. There is no shared memory between the two threads. While workers also run JavaScript, not all APIs are available. For example the WebRTC API is missing in workers and can only be used from the main thread. This means if one wants to do P2P synchronization with other browsers, this synchronization needs to happen on the main thread. One can offload the main thread by only sending and receiving the messages on the main thread and offloading the processing of it to one or more worker threads. But still some part of the main thread will be consumed. There are plans to allow WebRTC in workers in the next version of WebRTC [3].

WebAssembly threads. WebAssembly [9] (Wasm) is a new binary instruction format for the web. One can write programs in high-level languages like C++/Rust and compile them to Wasm, which can be executed on the web. The WebAssembly Community Group is currently standardizing a new feature called WebAssembly threads. Under the hood, they are still using WebWorkers but they can make use of the same shared memory.

ServiceWorkers [10] run in the background of the browser and can be used to provide rich offline experiences, periodic background sync and push notifications. A ServiceWorker acts like a proxy and can intercept requests from your page. One use case is to save requested pages in the cache. Later on, when the ServiceWorker detects that there is no internet connection, it can serve those cached pages to the user. This allows users to open a page even when they are offline. They can then use WebRTC to interact with other browsers and share data, without the need for a central server (within the boundaries of the current WebRTC implementation).

Security model of the browser. The current security policy in the browser is based on the Same-Origin Policy. This allows all scripts from the same origin (combination of scheme, host name and port) to access the same data. Scripts coming from another origin have their own data storage and have no access to those of other origins. This security model works fine when handling local, personal data. However, if we want to enable a more decentralized, client-centric web with shared data, where each client has their own local copy, more security measures are needed.

First of all, data can be shared between multiple origins: for example between a data storage provider and a service provider. In the current client-server model, such information would be requested from the server by the service provider and be authenticated via e.g. OAuth [5]. In the decentralized approach, that information might already be present in the browser, but stored under a different origin. Current implementations of the browser allow communication between scripts of multiple origins via message passing. This can be used by the service provider to locally (in the browser) request the required data from the storage provider, which can provide the correct information from the local copy, or download it from another storage location (be it another browser or a server). Important here is that the storage provider should have a way to verify the request coming from the service provider, even in a disconnected situation.

Next to data sharing between different origins on the same device, data can be shared between multiple devices, possibly owned by different users via WebRTC. Again the same story applies, independently from a central component, there should be a way to verify if the request should be full-filled or denied. A possible solution would be to use capabilities, which are signed claims that can be verified without contacting the authority that created it using cryptographic primitives. The browser-to-browser communication itself using WebRTC is encrypted by default. In fact, using it without encryption is not possible from within the browser.

4 PRELIMINARY EVALUATION

In this first preliminary evaluation we want to assess the impact on performance of shifting from a server-centric approach to a client-centric approach. A server-centric approach hosts the main copy of the data on the server and all clients synchronize with this main copy. The threading

model of the server-side technology is optimized to handle many concurrent updates with the different clients. So is the database behind it. A client-centric P2P approach however needs to handle the many concurrent updates from the other clients using the threading model and P2P communication model of the browser as discussed in Section 3. In this section we thus assess if the decentralized client-centric P2P approach can offload the server and achieve faster synchronization for various scales of data sets and users.

We have started implementing a middleware which uses state-based CRDTs [11] for the synchronization. Updates are synchronized by computing deltas dynamically, as is also the case in Legion [12]. The middleware can use WebSockets to synchronize updates between clients via a central server. This is the baseline used by most modern web applications today. It can also be used in a P2P setting using WebRTC to synchronize updates directly from browser-to-browser. A WebSocket connection to the server is used as signaling layer to setup the WebRTC connection.

We have implemented the shared whiteboard example using our middleware and tested the performance in both the classical client-server setting, as well as the P2P settings. We decided to use our own middleware to be able to change configurations and collect extensive metrics. In future work, we want to extend that middleware to support our vision of a decentralized, client-centric web.

Benchmark setup. The servers (including signaling and TURN) and browser clients are deployed as separate Docker containers on several VMs in our OpenStack private cloud. A VM has 8 CPUs and 16 GB of RAM and can hold up to 6 client containers. A client container contains a Chromium browser which loads the web application from a web server. The Linux traffic tool (`tc`) is used to artificially increase the latency between the containers to an average of 100 ms. Which resembles the latency of a bad 4G network.

The benchmark is executed with 10 and 30 clients. Each client makes one update per second to the shared data set. The shared data consists of 1000 objects on a canvas, which have properties like the position, size and color of the object. A single run of a benchmark provides us with 10 minutes of data. The 10 minute interval is preceded by a 1 minute warm-up period. Each benchmark is repeated 10 times.

Baseline: client-server. The baseline for comparing the P2P performance is an application that only synchronizes data via the server. We've implemented the shared whiteboard in a classical client-server architecture. The first two columns of Table 1 show the synchronization times and the network usage. The synchronization time is the time it takes for all clients to receive an update made by one client. We show both the median (50th percentile) as well as the 99th percentile of these synchronization times. Not only the average client should have a great performance, but most of the clients [4].

Peer-to-peer setup. Now we disable all synchronization via the server, only P2P synchronization is allowed. The data synchronization server now acts only as signaling layer to setup WebRTC connections between each client. The P2P-network is a fully-connected network where each peer is connected to all other peers. The results are in the last two columns of Table 1. Going from client-server to P2P communication improved the mean synchronization time with about 0.7 seconds. The network usage of each client raised to 1.2 Mbit/s in the large scale scenario with 30 clients, which is still far away from the maximum bandwidth available today. In comparison, the average download-bandwidth on a mobile network today is 27 Mbit/s [13].

Limitations of the P2P solution. While the previous paragraph talked about a full P2P solution, there were still two central servers needed to setup the connection. The actual data synchronization is indeed P2P and messages are sent from one browser directly to the other. But to initiate the connection, there is a signaling layer needed. This is implemented as a central server which is connected to the web applications via a WebSocket. The clients all have a unique ID and can request the list of other clients from the signaling server. The signaling server acts as a relay for control messages. Clients can send WebRTC control messages to the signaling server, combined with the ID of the destination. The signaling server will forward it to the correct client. Next to the central signaling server, a STUN server is needed for clients to discover their real IP address. The signaling server could be replaced by a manual procedure using e.g. QR codes. The STUN server can be removed when all clients are on the same network. This way, even during the setup phase, no central servers are needed. We used the signaling server to automate the tests, and STUN was needed because Docker containers have their own local network on each VM.

Conclusion. Our preliminary evaluation shows that browsers are ready to let the web evolve to use a decentralized, client-centric approach. P2P communication increases the interactivity of updates while the network usage stays low enough to be able to run on a mobile network. The fully

		client-server		peer-to-peer	
	clients	10	30	10	30
Sync. time [s]	50%	1.9	2.3	1.1	1.6
	99%	2.3	3.2	1.8	2.3
Bandwidth [Mbit/s]	server	1.5	8.9	0.0	0.0
	client	0.1	0.3	0.3	1.2

Table 1: Statistics for all benchmarks for the client-server and peer-to-peer situations. Numbers are the average over 10 tests of each 10 minutes with respectively 6000 and 18000 updates made per test.

connected P2P network works great for the scale of the benchmarks done already, but to scale to hundreds or thousands of concurrent clients, more structured P2P-networks will be needed.

5 RELATED WORK

The current client-centric web middleware platforms can be divided into three categories: 1) GUI-focused JavaScript frameworks (e.g. React and Angular), that only focus on local data binding of data with GUI elements, 2) libraries that focus on client-server REST communication (e.g. JQuery) and 3) data-synchronization focused frameworks (e.g. PouchDB, Yjs, Legion).

Some of these data-centric frameworks support P2P synchronization when a central signaling server is available. These synchronization frameworks help offloading the server and allow disconnected operation. However, prompt synchronization is only supported in small scale scenarios, i.e. tens of users. PouchDB is a JavaScript library to replicate data (as JSON-documents) with a CouchDB server. It doesn't support automatic fine-grained conflict resolution or P2P synchronization. Yjs [7] is a framework for synchronizing different data structures (maps, arrays, ...) using operation-based CRDTs [11]. It supports WebRTC as adapter to synchronize the changes to other clients. Legion [12] is a research prototype for P2P synchronization between web applications.

While the first four basic requirements we defined in Section 2 are currently supported, the last three are still a challenge to achieve truly decentralized, client-centric web applications. None of these frameworks provide extensive access control that help ensure the confidentiality of the data. They also don't provide any help with complex queries that run over multiple browsers.

6 CONCLUSION AND FUTURE WORK

This paper defined a set of key requirements for data operations in a middleware for a decentralized, client-centric web architecture. The move to such decentralized web applications in the edge is needed to allow operation in disconnected situations, offload web servers and regain control of your personal data.

We assessed the current state of the browser and its limitations. WebRTC needs a signaling layer to connect to other browsers and a STUN server is required to circumvent NAT. The event-loop driven threading model of JavaScript is not suited for large scale client-side application servers. However, WebWorkers and Wasm allow multiple execution lines. Indeed, our preliminary evaluation, using WebRTC and WebWorkers showed that interactive, P2P data synchronization is suitable in a browser. Synchronization times were even lower than the client-server variant.

More research is needed to scale up to hundreds or thousands of users interactively accessing the same data set. Next to the scalability, there are still 3 requirements missing before fully decentralized web applications can exist: 1) user-centric access control with selective sharing and synchronization of data items, 2) support for distributed select queries, and 3) decentralized data processing and analysis.

REFERENCES

[1] Bernard Aboba. 2018. *WebRTC Next Version Use Cases*. Working Draft. W3C. https://www.w3.org/TR/2018/WD-webrtc-nv-use-cases-20181211/

[2] Tim Berners-Lee. 2017. Three challenges for the Web, according to its inventor. https://webfoundation.org/2017/03/web-turns-28-letter/

[3] Jan-Ivar Bruaroey, Daniel Burnett, Taylor Brandstetter, Cullen Jennings, Anant Narayanan, Adam Bergkvist, and Bernard Aboba. 2018. *WebRTC 1.0: Real-time Communication Between Browsers*. Candidate Recommendation. W3C. https://www.w3.org/TR/2018/CR-webrtc-20180927/

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41(6). ACM, ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[5] D. Hardt. 2012. *The OAuth 2.0 Authorization Framework*. RFC 6749. https://www.rfc-editor.org/rfc/rfc6749.txt

[6] Ian Hickson. 2015. *Web Workers*. Working Draft. W3C. http://www.w3.org/TR/2015/WD-workers-20150924/

[7] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Engineering the Web in the Big Data Era*. Springer International Publishing, Cham, 675–678.

[8] Jakob Nielsen. 1993. *Usability Engineering*. Nielsen Norman Group. https://www.nngroup.com/books/usability-engineering/

[9] Andreas Rossberg. 2018. *WebAssembly Core Specification*. Working Draft. W3C. https://www.w3.org/TR/2018/WD-wasm-core-1-20180904/

[10] Alex Russell, Marijn Kruisselbrink, Jungkee Song, and Jake Archibald. 2017. *Service Workers 1*. Working Draft. W3C. https://www.w3.org/TR/2017/WD-service-workers-1-20171102/

[11] Marc Shapiro, Nuno Perguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems (Lecture Notes in Computer Science)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[12] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 283–292. https://doi.org/10.1145/3038912.3052673

[13] 2018. Speedtest.net. http://www.speedtest.net/reports/united-states/2018/Mobile/.