**Compute Offloading using JavaScript in Cloudlet Environment**

# Introduction

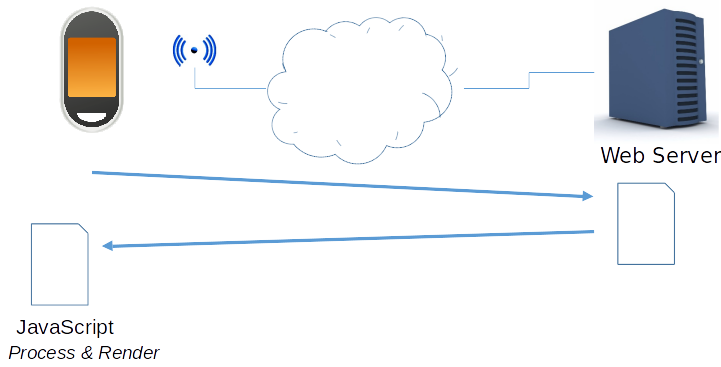
We are trying to analyze the feasibility of using Cloudlets to speed up standalone JavaScript mobile applications. We have taken an existing application and split it to run on offload mode (computation happens on Cloudlet, rendering happens on client), and also purely on the client side. A prototype library has also been written to dynamically provision the application to run on the Cloudlet if a connection to Cloudlet is detected.

# Available options

We had various options to split up the code so as to run in client-server mode. The options are discussed below:

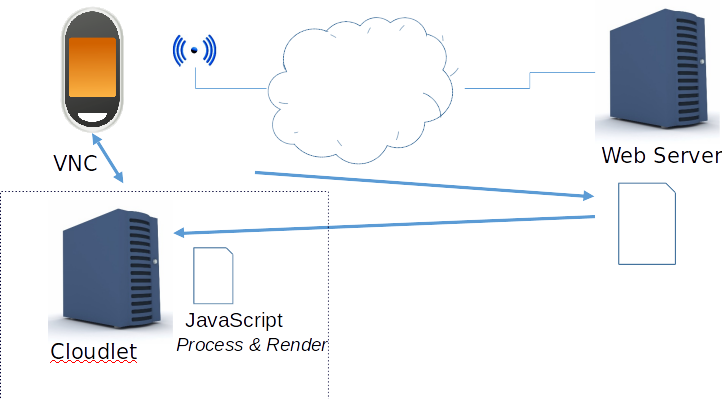
1. **Processing on Client**

All the processing happens on the client only. The client would fetch the required files from the webserver and do all the work related to the application on self.



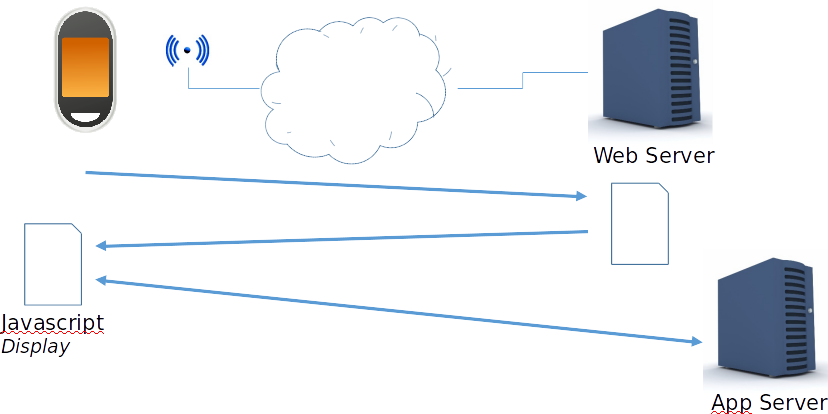
1. **Virtual Display on Client**

All the processing happens on the Cloudlet. The web server sends the files to the cloudlet instead of the client. The Cloudlet does all the processing, including the rendering. The display is then transferred to the client using a Virtual Network Connection



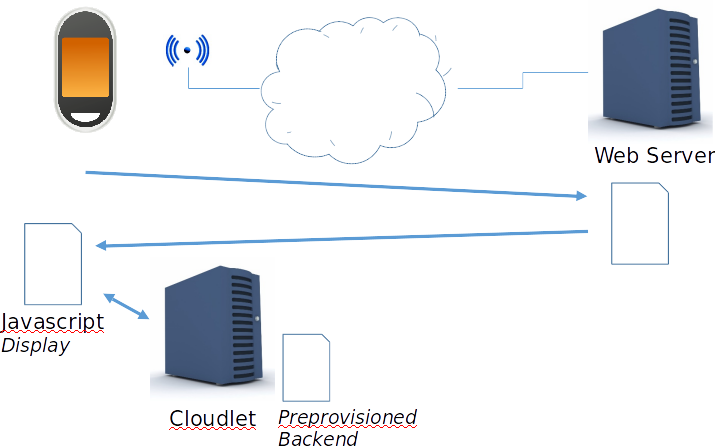
1. **Backend in Cloud**

There is a separate application server in the Cloud which does all the processing. The client communicates with that server for all the application related things.



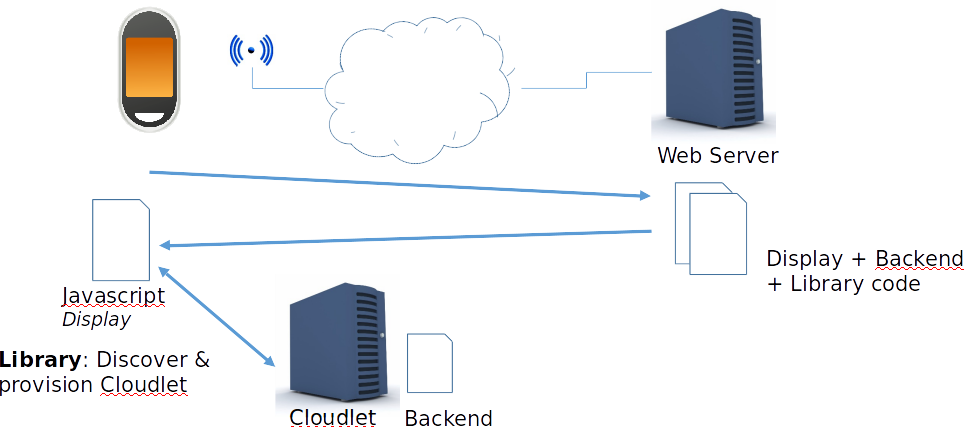
1. **Pre-provisioned Cloudlet backend**

There is a pre-provisioned backend in the Cloudlet. The server transfers the necessary files to the Cloudlet. The processing happens on the Cloudlet while the display happens on the Client.



1. **Dynamically provisioned Cloudlet Backend**

The client discovers the nearest available cloudlet and transfers the necessary code to that Cloudlet. It then does all the communication with that Cloudlet only. For this to happen, a library would be written which would act as a glue between the client and the server side code and decide where the server side code is to be run.



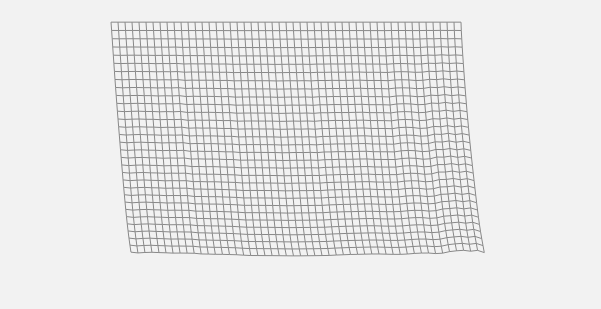
We have evaluated options 1, option 3 and a mix of option 4 and 5 in our experiments.

# Application Selection

In order to find a suitable candidate application for our experiments, we had some constraints which the application had to fulfill:

1. It should be easy to configure the application to increase the computational aspects.
2. It should be easy to configure the application to increase the data size to be transferred over the network.
3. The application should not use GPU for rendering purposes.
4. The non-rendering aspects should be significant in the application.
5. The application should run on mobile devices.

After evaluating applications such as [JavaScript Particle Simulator](http://jsoverson.github.io/JavaScript-Particle-System/), [CannonJs physics engine](http://schteppe.github.io/cannon.js/demos/heightfield.html), [Liquidfun.js](http://google.github.io/liquidfun/), we selected a rather simple, pure JavaScript application: [Tearable Cloth](http://codepen.io/suffick). This application simulates the behavior of a hanging cloth and allows us to move it around and tear it apart.



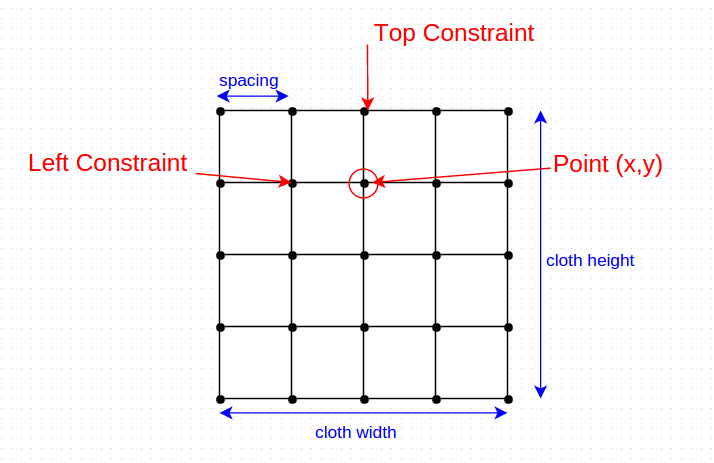
*Tearable Cloth application*

The application allows for various configurable parameters:

1. **Physics accuracy**
2. **Cloth height**
3. **Cloth width**
4. Spacing
5. Gravity
6. Tear Size
7. Mouse Cut
8. Mouse Influence

We are interested in bold parameters since they effect the computation and data size.

# Working of Application



The image above describes how the application actually simulates the behavior of a hanging cloth. A cloth is considered as an array of Points, where each point has its own x and y coordinates. Each point is bound to the point on its immediate top and left, which would be called as Top Constraint and Left Constraint respectively henceforth. The distance between the points is defined by spacing and it is used to calculate the coordinates of the next points. The total number of points in a cloth is (cloth height + 1)\*(cloth width + 1). The force of gravity is added downwards to each point.

To simulate the behavior, the cloth is updated continuously. During an update, new position of each point is calculated based on the position of its constraint points and a constant factor. For every frame, the update function is run physics accuracy times. For example, if physics accuracy is set as 5, the update is run 5 times before the frame is rendered. Each update builds upon the position set by the previous update.

If the distance between two points increases the tear distance specified, the connection between the points is broken and the cloth appears to have been torn apart. The tear also happens when right mouse button is pressed on the cloth or the mouse is moved with right mouse button pressed. Pressing the left mouse button and moving it over the cloth will move the cloth and simulate the behavior accordingly.

# Partitioning the application

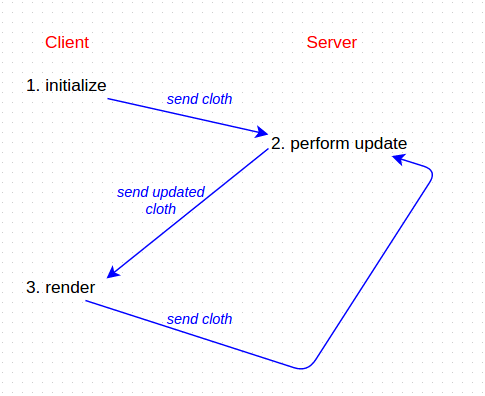
The application was purely JavaScript meant to run on client side only. It started showing signs of lagging when accuracy was increased beyond 20.

The first task was to partition the application such that all the processing happens on the server and the client handles only the rendering aspects.

There were two approaches to handle this:

1. **State resides at client**

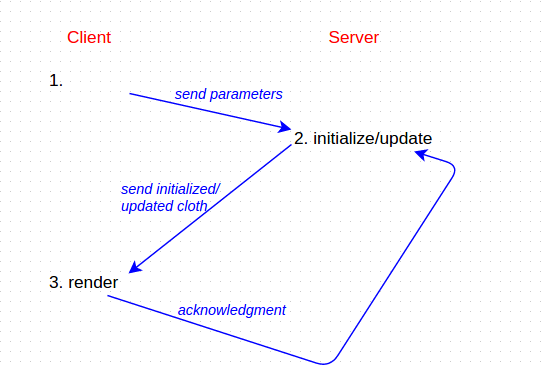
The cloth is initialized at the client side. The client sends the full cloth to the server; the server updates it and sends it back to the client. This happens continuously. At any given point in time, the client has full information about the state of the cloth.

¬

The drawback with this approach is that at every update, the full information for the cloth is being sent over the network. This is a lot of data transfer given that update happens continuously.

1. **State resides at server**

The cloth is initialized on the server. The client only sends the parameters necessary to initialize the cloth and the server initializes and updates the cloth from then on. The client only receives the information necessary to render the cloth. The full state resides with the server.



This is the approach that has been taken and would be discussed further on.

# Challenges in Partitioning

## Recursive data structure

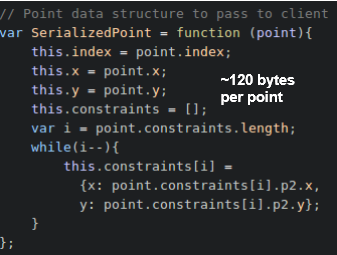
The data structure being used to represent a point in the original application was recursive in nature. What this means is that each point had its constraint point stored in the form of [self, constraintPoint]. It is not possible to pass this data structure over the network because the default serialization mechanisms in JavaScript (JSON.stringify()) are not able to handle recursive structures. The data structure had to be changed to remove the recursive nature and now it stored the constraint points in the order [constraintPoint1, constraintPoint2].

## Stack Overflow

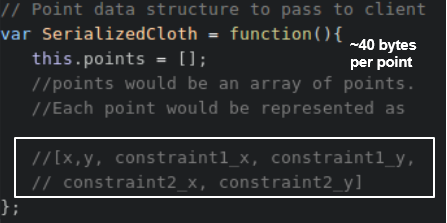
Changing the data structure to remove the recursive nature did not solve the problem completely. Since each constraint point was still a object reference, serializing a point meant serializing its constraint points also. The constraint points had their own constraint points and it easily caused stack overflow for a cloth of dimensions 10x10 also. This was removed by sending a completely different data structure over to the client which had information relevant to rendering only. So each point had its own coordinates along with the coordinates of the constraint points sent over to the client.

## Data Size

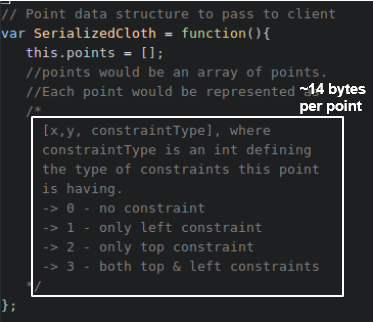
After solving the above two mentioned problems, the application was working in client-server mode. However, the data size being transferred per point was very large. The below image shows the data structure being passed and how much size did it took for each point



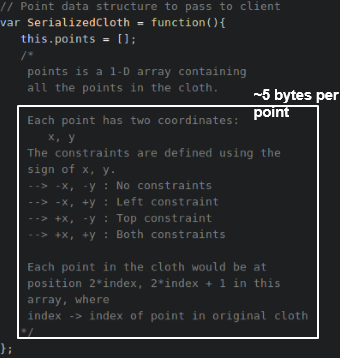
The data structure was then changed to remove the JSON structure (key, value) and passed information in the form of an array only. Each point was an array of coordinates and the cloth itself was represented as an array of points. From this point onwards, we started using MsgPack library to transfer data in binary format over the network rather than in the form of a string.



Since we were dealing with a specific application, the client gets the coordinates of all the points when it receives a cloth. Embedding the coordinates of the constraint points within a point was duplicating the information. The new data structure took care of that and each point now only contained information about the type of constraints it had. The actual constraint coordinates were identified at the client side.



This was way better than from where we started. However, there were still scope of improvement. The additional information to specify the type of constraint could be embedded in the point itself. Since there can be only four types of constraints, we could take advantage of the +/- signs of the coordinate points to embed this information.



This was our final data structure. It can be seen that we started from 120 bytes per point and reached to 5 bytes per point and still preserved the original nature of the application.

# Results

Having partitioned the application and made it work in the client-server mode, it was time to take the final set of readings. End-to-end latency was measured as part of the readings. Please note that we had readings taken at every data structure but the graphs shown are from the final set of readings.

The application was configured to run in three different modes:

1. **Offloading to the Cloudlet**

The processing happened on the cloudlet. Please note that we had not done any provisioning at this point. The cloudlet acted as the web-server also. A private wireless network of 100Mbps was used to take the readings, the mobile phone was in charging mode and had a ping application running continuously in the background. The ping application was used in order to prevent the phone’s network driver to go into sleep mode after some period of inactivity.

1. **Offloading to the Cloud**

The processing happened on the cloud. The cloud machine was hosted on Amazon AWS West instance.

1. **No Offloading**

There was no offloading and everything happened on the mobile phone only.

The following settings were used for each of the modes mentioned above:

1. **Increasing Computation**

The physics accuracy was increased at constant steps. The following accuracies were used:

5, 25, 45, 65, 85, 105

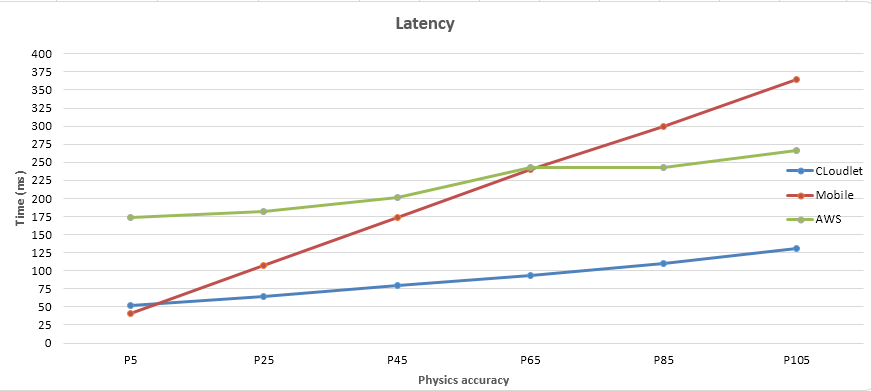
1. **Increasing Data size**

The height and width of the cloth was increased to increase the amount of data flowing over the network. The table shows the parameters used and the corresponding data size:

|  |  |  |
| --- | --- | --- |
| **WIdth** | **Height** | **Data Size (KB)** |
| 80 | 90 | 33 |
| 120 | 182 | 110 |
| 150 | 244 | 188 |
| 210 | 244 | 274 |
| 260 | 245 | 346 |
| 317 | 260 | 452 |

Since the increase in data size also increased the computation, the data size readings were taken for three different accuracies: 5, 25, 45

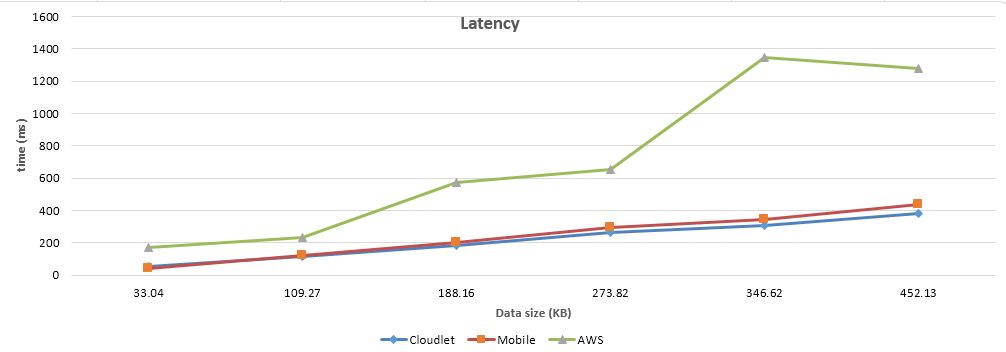
## Computation



The graph shows the results when only the computation aspects were increased. It can be clearly seen that Cloudlet performs way better than both the AWS cloud and the Mobile. There is a spike in the AWS reading which can be attributed to the fluctuation in the bandwidth.

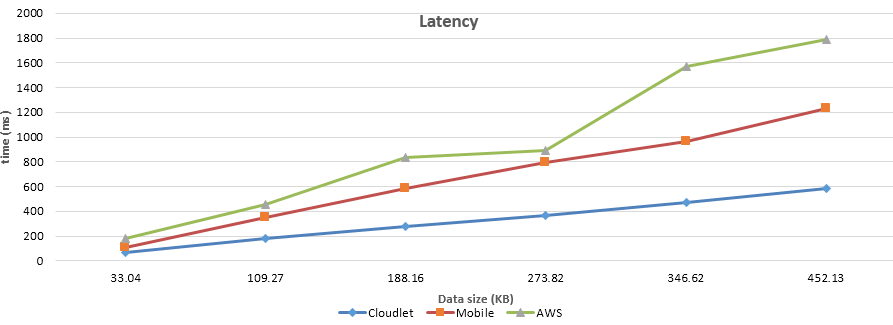
## Data Size

### Accuracy 5



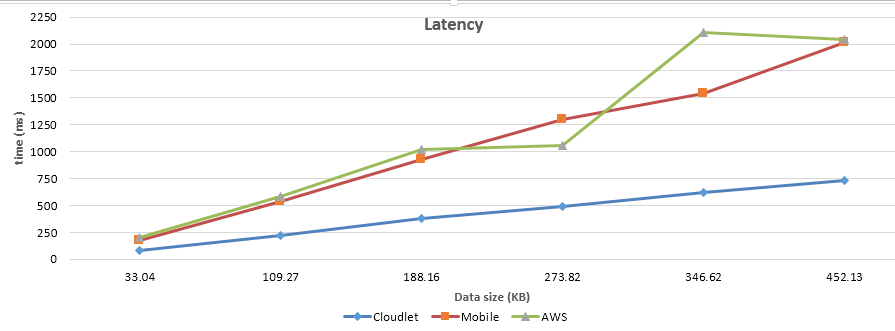
The graph shows the change in end-to-end latency as the data size was increased. It can be seen that Cloudlet performs better than the mobile but the difference between Cloudlet and mobile is not much. AWS performs the worst. The spikes in AWS can be attributed to the fluctuations in the network bandwidth.

### Accuracy 25



The mobile is not able to handle increase in the data size and computation much.

### Accuracy 45



The increase in the end-to-end latency of the cloudlet (600 to 750) is not much as compared to the increase in that of mobile (1200 to 2000). Cloudlet definitely wins over in this case.

# Provisioning

Having done the partitioning and obtaining the readings, we knew that is beneficial to offload the processing to Cloudlet. We wanted to have a library which would dynamically provision the code on the cloudlet and run it from there. The user should not worry about running the application on mobile or on cloudlet. To have this, the following was done:

1. A simple web-server was created which would serve the files to the client. The files would be those which are necessary to run the application on client side along with the library code we have written.
2. Once the client has all the files from the server, the library code would run. It would try to establish connection to the cloudlet. If the connection to the cloudlet is established, it would transfer the files which are necessary to do the server side processing to the cloudlet.

Please note that as of now, the IP of the cloudlet, the files to be sent to the Cloudlet and the file to be run on the Cloudlet is embedded in the html page. The library reads it from there.

1. When the cloudlet receives all the files, it loads the file which is to be run. The client now starts communicating with the cloudlet.
2. The Cloudlet is able to handle connections from multiple clients. It spawns the number of workers equal to the number of CPUs on it. Each worker is responsible for handling all the requests coming from an IP. Requests from a different IP are handled by a different worker. Also, whenever the cloudlet receives a file, it stores the files with a different name. When the connection with the client is lost, the file is deleted. This ensures that each client can run a different version of the server side file.

# Reflections

1. It is easy to split a code to run in client-server mode. What is difficult is to make the code efficient. Since JavaScript in inherently single threaded, it is difficult to leverage the multiple cores of the server to do the processing.
2. Network bandwidth is a critical factor in improving the performance. As we kept on decreasing the data size, the performance kept on improving.

# Future work

The aim of this work was to have a prototype which can ensure us that what we thought of is feasible. Having ensured this, here is a list of future work which can be done:

1. **Dynamic discovery of Cloudlet**

Currently, the IP of the cloudlet is embedded in the html page. We can have a library which would detect the nearest cloudlet and establish connection with it.

1. **Transfer of state**

As of now, the application can run in either of the two modes but it cannot automatically switch modes depending on the connection. If we are able to transfer the entire state of the application to either mobile or cloudlet without the user getting to know about it, it would be great. For example, the application is running on mobile only, it detects a connection to the Cloudlet, it transfers the entire state of the application to the cloudlet and the application is now running from the cloudlet. The user does not get to know that a switch has happened.