A study of limitations and performance in scalable hosting using mobile devices

NIKLAS RÖNNHOLM
A study of limitations and performance in scalable hosting using mobile devices

DA222X, Masters Thesis in Computer Science
En studie i begränsningar och prestanda för skalbar hosting med hjälp av mobila enheter

DA222X, Exjobbsrapport i Datalogi

Niklas Rönholm
nron@kth.se

Supervisor: Erik Isaksson
Examiner: Johan Hästad

School of Electrical Engineering and Computer Science,
KTH 2018

March 2018
Abstract
At present day, distributed computing is a widely used technique, where volunteers support different computing power needs organizations might have. This thesis sought to benchmark distributed computing performance limited to mobile device support since this type of support is seldom done with mobile devices. This thesis proposes two approaches to harnessing computational power and infrastructure of a group of mobile devices. The problems used for benchmarking are small instances of deep learning training. One requirement posed by the mobile devices’ non-static nature was that this should be possible without any significant prior configuration. The protocol used for communication was HTTP. The reason deep-learning was chosen as the benchmarking problem is due to its versatility and variability. The results showed that this technique can be applied successfully to some types of problem instances, and that the two proposed approaches also favour different problem instances. The highest request rate found for the prototype with a 99% response rate was a 2100% increase in efficiency compared to a regular server. This was under the premise that it was provided just below 2000 mobile devices for only particular problem instances.

Sammanfattning
Chapter 3.5 Test characteristics

3.5.1 Connection loss

3.5.2 Test input

3.5.3 Input & Output data format

Chapter 3.6 Effectiveness formula

3.6.1 How to calculate transfer time of a given number of bytes

3.6.2 How to calculate input time

3.6.3 How to calculate processing time

3.6.4 How to calculate output time

3.6.5 Comparing performance in different platforms

Chapter 4 Results

4.1 Test 1 - Dispatcher/Proxy single device capacity comparison

4.2 Test 2 - 35 device capacity comparison with Normal Server

4.3 Test 3 - Dispatcher & Normal maximum capacity comparison

4.4 Problem specific data

4.5 Connection loss probabilities

4.6 Efficiency comparison

Chapter 5 Discussion

5.1 Dispatcher vs. Normal server

5.2 Proxy vs. Dispatcher

5.3 Connection overhead

5.4 Efficiency model evaluation

5.5 Report weaknesses

5.6 Possible improvements & Future work

5.7 Deep learning & other problems classes

5.8 Using mobile devices vs Regular computers

5.9 Commercialization

Chapter 6 Conclusion
Terminology

**Mobile device** A mobile device, as referred to in this report is a modern smartphone or tablet equipped with Wi-Fi that has an operating system which supports 3rd party applications.

**NAT** Acronym for Network Address Translation, which is used when forwarding traffic from a local host using a local address to the Internet. This is done by creating a temporary translation between the local address space to the global address space (Internet). See section 2.5 for more information.

**DNS** Domain Name Service abbreviated “DNS” is a system made to give a particular IP-address a name (RFC1035).

**Scalable hosting** A technique used to connect multiple servers into a more powerful cluster of servers, hosting the same material. See section 2.4 for more information.

**Dispatcher** A dispatcher is a server used to dispatch clients from itself to the correct node so to avoid handling processing or responses. See section 2.4.3 for more information.

**Reverse proxy** A reverse proxy (Apache website) is an intermediary server used by clients to access a server behind a firewall. The difference from a ‘regular’ proxy (Structure and encapsulation in distributed systems: the proxy principle, Marc Shapiro, 1986) is that it does not forward traffic from clients outwards towards a network, but forwards traffic from clients on a network inwards towards a server.

**Web Service** According to W3C, a web service is software that supports machine-to-machine communication, usually over HTTP. Such machines most often distribute prepared files (such as image files or text files), and to perform calculations as well (services such as “WolframAlpha”). For the purpose of this report, this was referred to as file distributing services and processing services.

**Deep Learning** Deep learning is a term used for describing learning systems using layered computational models capable of abstracted learning of different types. See section 2.2 for more information.

**FLOPs** FLOPs (TechTarget, 2011) is an abbreviation of “floating-point operations per second” which is a common measure of computing capacity when comparing different hardware.
1 Introduction

This section describes the project in terms of subject, purpose, scientific question, limitations, delimitations, scientific contribution and its relevance to society.

1.1 Thesis subject

An approach emerging as an increasingly prevalent concept at present-day is referred to as volunteer computing and is mentioned in a report on a project called Folding@home which was used to simulate biological phenomena by utilizing hundreds of thousands of personal devices (described further in section 2.1.1). Volunteer computing can be described as a form of distributed computing, with the distinction of having computers connected by volunteering owners. Volunteers in the context of volunteer computing and this report are defined by their ambition to support a certain requirement of computing power to reach a common goal.

Today, the quantity of mobile devices on the planet is significantly greater compared to previous decades. This category (mobile devices) are constituted by smartphones in some form (nearly every individual in developed countries has access to one or several devices). These devices are equipped with considerable computing power which is difficult to concentrate. The principal reason for this is their non-stationary positioning and irregular use of multiple gateway networks when swapping Wi-Fi networks. This limits their usefulness and excludes all solutions which require a permanent internet connection as well as those where one would be required to configure one’s gateway router/firewall/NAT. Should volunteers be required to configure their device network every time they relocate, the inconvenience for each volunteer would be too frequent resulting in a significant increase in effort required to volunteer and a subsequent decrease in the number of potential volunteers.

Currently, many NATs prevent clients outside networks to connect to the devices within the network RFC5345. NAT-traversal, which is currently the biggest problem posed by NATs could be avoided if there was a general update to the IP protocol. One of the main benefits of IPv6 (which was described as early as 1995 by the IETF) is its significantly larger address space, which would make the extension of the current address space by the use of NATs obsolete, since one of the main reasons of NATs is to compensate for the lack of addresses, it would be partly redundant. However, since NATs provide network security they cannot easily be phased out in favour of just using IPv6 (as described by an article on an IPv6 website). Also, since many older systems use IPv4, the transition into IPv6 is very slow. Eliminating the data traffic middle-men and unnecessary subnets would enable more peer-to-peer technologies to emerge, and thereby also significantly increase the availability of systems used by volunteers in volunteer computing.
One way to increase utilization in volunteer computing using mobile devices is to overcome their inherent limitations, and thereby harvest the huge amounts of computing power. If connectivity could be improved either through IPv6, NAT-traversal (see section 2.5 for more information on NAT) or simply using direct Internet connections, one could utilize mobile devices even more in peer-to-peer technologies or more advanced volunteer computing. One example would be that web services could be distributed on the clients’ devices, increasing the collective capacity and making them more resilient to DDoS-attacks and more scalable due to the increased infrastructure used.

Should this solution prove to work, one obvious use case of the proposed subject would be something similar to Folding@home, except where the devices used are different. In the case of the given application, one could construct something similar to the mathematical engine “WolframAlpha” which is used for small calculations through a web interface. The difference for the solution proposed in this report would be that the back-end uses many small computers (provided by volunteers) instead of a single large computer that is hosted by the company owning the service. Volunteers could be almost anyone, but the most obvious group would be the users themselves. One special problem which is applicable to volunteer computing is deep learning. Deep learning is applied in varying types of fields, from image compression to speech recognition.

The sustainability aspects in volunteer computing using mobile devices are significant, since one could support a less wasteful trend regarding computer power potential. As the majority of mobile devices are personal and only used by the owner occasionally, there will be moments where its free to use. Consider for example during nighttime when a given user is asleep with the mobile device connected to an electrical outlet and a Wi-Fi access point. In any situation where a volunteer would like to contribute he or she could simply start an app and instantly commit their device. If one could implement this solution within all time zones, one could have a considerable amount of (almost) free computing power on demand during any time of the day.
1.2 Goal & problem formulation

The scientific question of this thesis could be formulated as follows:

*Can mobile devices be used in a seamless manner to assist regular HTTP servers tasked with solving deep learning problems and in so doing increase the collective serving capacity?*

1.3 Limitations

Due to the limited time for research of the tests, the number of devices was limited to thirty-five emulated devices and one real device. This choice was based on the number of functional computers within two computer labs at KTH. Emulated devices work well in all regards except that they might be slightly faster (mostly because of that more powerful host processors have a higher frequency than a real device). This is compensated through the use of real device benchmarks.

According a world leading information technology research and advisory company called Gartner, Android was the most common operating system on smart phones in the world (84.1%) in 2016 [14]. Because of this, the choice of mobile devices in this report was limited to Android devices (smart phones running on Android), meaning no laptops or other mobile devices were used. The reason for this being that laptops are similar enough to produce comparable result, but require resources that are not available.

1.4 Delimitations

In order to produce measurable results, the problem was applied to the task of training neural networks (supervised learning). Read more on the choice of this in section 3.1. Availability of mobile devices (methods of aggregating device volunteers), the synchronization and initialization of a given solution, or security in any extensive form (encryption or data integrity) was not given special focus in this report. The client interface was limited to the HTTP protocol (as it was the most conventional mean of Internet communication). This thesis did not consider mobile Internet, as it would have created expenses and thereby adding complications for volunteers.

Problem instance sizes investigated in this report are limited to small instances with computation times up to approximately one second. Larger problem instances were not investigated, though parallelization is briefly discussed in the report, however not implemented in any way. The main reason for this is that methods of parallelization of any problem classes would enlarge the scope of this report too much.
2 Background

This section contains explanations of concepts, technologies, standards, and much more that are examined in this report. Much of what can be found here relates to distributed systems of computation (both concepts and examples) and distributed devices in general. Facts describing the related subjects, concepts fundamental for the problem statement, as well as techniques used in the thesis method are explained in this section.

2.1 Related works

This section contains works related to this thesis in order to educate the reader on relevant concepts which serve both as sources of inspiration and terminology as well as subject relating boundaries.

2.1.1 Folding@home

Folding@home (“Folding at home”) is a computer system described in a paper published 2009 (Folding@home: Lessons from eight years of volunteer distributed computing, Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq and Vijay S Pande [7]) designed for statistical calculation of particle trajectories (for particles such as atoms and molecules) in biological systems. The distribution of workload served more to run several simulations concurrently rather than running one simulation faster. It was built to be executed distributed on about 400 000 devices with a total of about 4.8 PetaFLOPS as of 2009. Folding@home is executed with an array of different servers used to define, distribute and coordinate the jobs to volunteers’ devices and return the computational results to be stored in a database. The distribution eliminates difficulties with temperature, electricity, physical space and above all; cost. Back in 2009, this system was the fastest in terms of FLOPS and the bottleneck for scaling up the number of users and achieving an even higher FLOPS number was backend server limitations.

2.1.2 Internet of things

The term Internet of Things is attributed to have been coined by Kevin Ashton in 1999 [15] and are later described more in detail by a paper written in 2010 (The Internet of Things: A survey, Luigi Atzori, Antonio Iera and Giacomo Morabito [16]) as the term becomes more used. He argues that the future will be characterized by letting machines and devices communicate more and interact autonomously with their networked environment. This will force technology into becoming more efficient. Furthermore, there have since been papers discussing this new paradigm that have been more and more highlighted as it comes more into existence. One paper discusses how different fields of knowledge will be synergetically linked and how the development of it should be approached [16]. Fields that are theorized to be improved upon is all from healthcare to social...
2.1.3 Peer-to-peer technology

Peer-to-peer is the concept of sharing resources within a network built entirely out of peers (described in A Survey of peer-to-peer Content Distribution Technologies, Stephanos Androutsellis-Theotokis and Diomidis Spinellis, 2004 [17]). It is adaptive when it comes to operation, and can serve great numbers of clients and maintain processing power and connectivity. A survey done 2003 (A Survey of peer-to-peer Security Issues, Dan S. Wallach [18]) announced security aspects with the technology. Peer-to-peer networks are vulnerable to bad peers in cases where peers are not vetted or controlled. This can lead to attacks primarily on the application level, where the peer sends out bad data, hides data, or similar. Also, attacks on the network level can occur. This leads to that connections between peers can be manipulated, most often that a peer has the ability to enable attackers into the network. The risk of an attacker penetrating the network increases for each node that is connected, since each new node is a potential point of entry into the network. Peer-to-peer systems can either be regular distributed systems that coordinate and function with just the peers, and then there are systems described as hybrid peer-to-peer systems (Comparing Hybrid peer-to-peer Systems, Beverly Yang and Hector Garcia-Molina, 2001 [19]). Hybrid peer-to-peer systems still are distributed in some sense, but also have a centralized component or functionality.

2.1.4 Mobile service platform: A middleware for nomadic mobile service provisioning

One research paper that ties closely to this report was one describing case studies done on so-called nomadic mobile services (Mobile Service Platform: A middleware for nomadic mobile service provisioning, Aart van Halteren and Pravin Pawar, 2006 [20]). This paper defines a nomadic device as a hand-held or otherwise mobile device, which an Internet connection through wireless network. This device is also expected to roam between wireless networks, hence its name. Such a device must have a seamless connection between service and client and must consider limitations set by its environment. Such a system must be scalable, (meaning that devices must be able to be added without effort). The paper also proposes a “Mobile Service Platform” to support this type of service.

2.2 Deep learning

The subject for computations in this report was deep learning, a subject belonging to AI. Deep learning (Neural Networks. A comprehensive Foundation, Simon Haykin, 1994 [21]) is known for requiring huge amounts of computing
power. Deep learning is only used as workload for the system proposed in this project, and is thereby not critical for its success. As mentioned earlier (in section 1.1 Thesis subject) the applications are wide, which should be reason enough for choosing deep learning as a computational problem.

### 2.2.1 Multilayered feedforward neural networks

Multilayered feedforward neural networks \[21\] consist of three types of layers. First, one input layer which receives its input from outside the model. Following are one or several hidden layers are present, which receives their input from the input layer and forwards it through the subsequent hidden layers on to the final (output) layer. The last layer (called the output layer) delivers the final output for the network. Between these layers are weighted connections, which help to transform the input into some other output. Such a model is called a multilayered perceptron (see figure 2.2.1). A perceptron with no hidden layers is simply referred to as a perceptron, or single-layer perceptron.

![Multilayered feedforward neural networks](image)

Figure 1: 1: Input layer, 2: Weighted connections, 3: Hidden layer 1, 4: Hidden layer \(n\), 5: Output layer

The learning aspect of these structures originates from the back-propagation algorithm (or error correcting rule). One can refer to these different phases as the forward pass and the backward pass. The forward pass is when the network is calculating the output from a given input and the backward pass is when the network adjusts. Adjustments are obtained from the difference between the produced output and the expected output).

An important aspect of each layer’s neurons, is their production of output based
on what their nonlinear activation function tells them how to behave. This is done by combining the previous layers' output with their respective weight vector taken from the weight matrix. One common nonlinear function is:

\[ y_j = \left(1 + e^{-v_j}\right)^{-1} \]  

where \( v_j \) is the weighted sum of the previous output for the \( j \)th node. \( y_j \) is the \( j \)th neuron’s output.

\( v_j \) is defined as:

\[ v_j = \sum_{i=0}^{n} w_{ij}(n)y_i(n) \]  

Using the final layer’s output combined with the expected output (for the given input), one can define the numerical error used in the backward pass.

The change in weights between nodes \( i \) to \( j \) is defined by:

\[ \Delta W_{ij}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial v_j(n)} y_i(n) \]  

\[ \varepsilon(n) = \frac{1}{2} \sum_{j} e_j^2(n) \]  

\[ e_j(n) = d_j(n) - y_j(n) \]  

The learning constant is denoted by \( \eta \), a given data point is denoted by \( n \) and the learning rate is denoted by \( \varepsilon \). \( e_j \) is the error in node \( j \), which is calculated by taking the expected output \( d_j \) and subtracting the observed output. This is then subtracted to achieve “learning”.

### 2.3 Parallel computing

Parallel computing is the notion of computing a problem on different processors at the same time. Some computational problems are embarrassingly parallel; which implies they easily can be divided into components that can be executed concurrently (The art of multiprocessor programming, Maurice Herlihy and Nir Shavit, 2011 [22]).

#### 2.3.1 Distributed computing

Distributed computing is a form of parallel computing (Distributed systems: concepts and design, George F. Coulouris, Tim Kindberg and Jean Dollimore [23]) where the parallel elements are using a network to communicate. Three common characteristics are: absence of exact common timing, failure-independence,
and problems with timing in the task it is set to solve. Distributed computing over a network requires each node to feature independent memory and hardware resources.

2.3.2 Amdahl’s law & Gustafson’s law

Amdahl’s law and Gustafson’s law describes the expected speedup of parallelization when using different numbers of processors on problems that are differently parallelizable.

Amdahl’s law:

\[
S = \frac{1}{1 - p + \frac{p}{n}}
\]

The variable \( p \) is the fraction of the total job that can be concurrently executed. The variable \( n \) is the number of processors executing the calculations in parallel. How many times faster the calculation becomes compared to if one processor would do all work sequentially is determined by the variable \( S \). The higher speedup \( S \), the better. Amdahl’s law was proposed in 1967 [22], and in 1988 a new law was proposed (Reevaluating Amdahl’s law, John L Gustafson [24]) called Gustafson’s law, which instead says:

\[
S = n + (1 - n) \times s
\]

The reasoning behind this being that \( n \) (processor count) and \( p \) (fraction of the total jobs that can be concurrently executed) are correlated.

2.4 Scalable hosting & Load balancing

The concept of running multiple servers that serve the same content has been proposed more than 20 years ago. In an article published in 1994 (A scalable HTTP server: The NCSA prototype, Eric Dean Katz, Michelle Butler and Robert McGrath [25]) about a method used at National Center for Supercomputing Applications for scalable web servers was theorized. This implementation is based on using a DNS that uses a round-robin scheduling algorithm to distribute requests among the server cluster and provides scalability and increased load capacity for the system. Load balancing is the distribution of workload over multiple servers or agents and is used to optimize the utilization of server clusters (used in scalable hosting). Below are common techniques used for web servers.

Cloudflare [26] is one of the most modern load balancing services and its main purpose is to distribute content over geographically distant infrastructure with high performance data centers. It also provides DNS services, reduces latency, and load balances to increase the performance of another web service.
2.4.1 Client-based load balancing

Client-based load balancing (Dynamic load balancing on web-server systems, Valeria Cardellini, Michele Colajanni and Philip S Yu [27]) is defined by placing the logic for load balancing within the clients requesting entity, such as a web browser or similar. Its advantages are that there is no server overhead, eliminating load balancing-related bottlenecks for the host and making the cost fall upon the client instead of the host. Its disadvantage is that it is not as dynamically applicable as some other techniques.

2.4.2 DNS-based load balancing

DNS-based load balancing [27] is defined by placing the logic for load balancing in the domain name look-up server all clients use for translating the domain name into an IP-address. Advantages with this technique is transparency and the elimination of load balancing-related bottlenecks for the host. Its disadvantage is that it only provides a primitive control over the load balancing.

2.4.3 Dispatcher-based load balancing

Dispatched-based load balancing [27] is defined by placing the logic for load balancing in a server for the entities that the load is to be balanced upon. This server then redirects all the data traffic. The advantage is that the host gains complete control over the load balancing. Its disadvantage is that it might be a bottleneck when load increases on the dispatcher.

2.4.4 Server-based load balancing

Server-based load balancing [27] is defined by placing the logic for load balancing in the server cluster itself, and giving all host server entities the capability of load balancing. The advantage of this technique is the distribution of control to the host servers. Its disadvantages are that both server overhead and latency is increased.

2.5 Network address translator

The Network address translator (abbreviated NAT) (described by RFC2663 [28]) is responsible for translating between local IP addresses with ports and external ports for the gateways external IP address. This enables clients within a local network connected to Internet via this gateway to access the Internet. The NAT has great significance in communication between local area networks since they might obstruct communications between peers. This is mainly if the NAT is incorrectly configured, which puts different requirements on different possible solutions to the thesis problem.
2.5.1 NAT variants

The RFC3489 [29] defined four NAT variations for using STUN defined in the same standard, although they apply for all communication through the NAT. These define what rules the NAT applies to its mappings when traffic is sent through the NAT. The RFC3489 has been obsoleted by RFC5389 [31], but the NAT variations still apply since they are not defined but only described by RFC3489.

**Full-cone NAT** Full-cone NAT [29] operates by mapping an internal host IP address and port to the same external host IP address and port over time. All external hosts can send packets through this mapping at any time.

**Restricted-cone NAT** Restricted-cone NAT [29] operates by mapping an internal host IP address and port temporarily. An external host can send packets to an internal host on the condition that the internal host has sent a packet to the external hosts IP address before.

**Port-restricted cone NAT** Port-restricted cone NAT [29] operates by having an internal hosts IP address and port are mapped to the same external IP address and port. An external host can send packets to an internal host on the condition that the internal host has sent a packet to the external hosts IP address and port before.

**Symmetric NAT** A Symmetric NAT [29] operates by mapping an internal hosts IP address and port along with any destination IP address and port the internal host sends packets to with the same external IP address and port. An external host can only send packets to an internal host on the condition that the internal host has sent a packet to the external hosts IP address and port before.

2.5.2 NAT traversal

The NAT can sometimes prevent direct connections over the Internet, depending on the type of NAT and connection. Below are some relevant chosen techniques to penetrate the NAT.

**Static address assignment** Static Address assignment [28] is the basic method which is done manually, where the network administrator creates static NAT mappings. This is independent from any NAT type.
Universal Plug and Play (UPnP)  UPnP (defined by the Internet Gateway Device standard 2 [31]) was defined by The Open Connectivity Foundation [32] and is a protocol specification for Internet Gateway Devices made to accommodate UPnP. This allows clients to dynamically request for ports to be opened. One prerequisite for this traversal method is that the gateway supports UPnP.

Traversal Using Relays around NAT (TURN) TURN (RFC5766 [33]) uses a relay to enable communication between two peers behind symmetric NATs. It supports both UDP and TCP. The relay can be seen as a temporary Proxy Server. TURN only works for symmetric NATs.

Session Traversal Utilities for NAT (STUN) STUN [30] is a protocol which is used as a tool to gain connectivity information for NAT-traversal as well as creating and maintaining connectivity through the NAT. STUN works with full cone NAT, restricted cone NAT, and port restricted cone NAT.

Interactive Connectivity Establishment (ICE) ICE [3] is a peer connection technique suite, that is designed to find (if possible) a way to connect two peers behind NATs. It uses both STUN and TURN to achieve this.

2.6 Load generation – httperf

Httperf [34] is a HTTP load generation tool that is founded on ideas proposed in a report written about web server benchmarking by Gaurav Banga and Peter Druschel [35]. It states that ordinary techniques only achieve server loads that are insufficient. More specifically, they do not have request spikes and peaks that overload the server in any realistic way similar to a massive client base. It suggests that one should implement HTTP test mechanisms so as to avoid pitfalls in TCP, such as the exponential back-off mechanism. This can be done using timeouts, so to close sockets that did not connect and thereby freeing up resources for the emulated client. It also emphasizes packet loss and congestion, and proposes techniques to counter it as well.

2.7 GPU vs CPU for computing deep learning

If one should utilize mobile device GPUs instead of mobile device CPUs, the computations would be much faster for this particular problem (Large-scale deep unsupervised learning using graphics processors, Rajat Raina, Anand Madhavan and Andrew Y Ng, 2009 [36]). This is because the matrix computations involved in deep learning would be more suited for GPUs (GPUs are traditionally preferred before CPUs for deep learning). Possibly, this could even further boost the performance of the system.
2.8 Security

Security is an aspect that is most relevant for almost all computer systems. This thesis investigates solutions where a lot of distributed devices would be connected together, as well as trusting them to deliver correct responses. In a real world application that relates to the question posed by this thesis, identity verification of volunteers and continuously updated and signed software should be used, since an attacker could attempt to fake volunteers, or penetrate the server in order to achieve some malicious goal. However, since this is an entire topic for itself and would have consumed too much time if it had been included, it is not part of the method or any results produced in this thesis.
3 Method

The goal of this report was to answer the question: “Can mobile devices be used in a seamless manner to assist regular HTTP servers tasked with solving deep learning problems and in so doing increase the collective serving capacity?” In order to answer this, one or more prototypes utilizing this technique had to be constructed and then tested. The method describes two prototypes capable of doing this, and a series of performance tests for both prototypes. The choice that led up to selecting deep learning training as a problem for benchmark is also described. The method of the thesis (which was designed to be as pragmatic and empiric as possible) was divided into four main steps:

1. **Choice of server problem (section 3.1)** – The distributed computing technique investigated in this thesis was comprised of data serving and computational problems. This defined boundaries for both scope, goal and the method altogether. Metrics were clearly defined so as to enable a benchmark.

2. **Design of prototypes (section 3.2)** – The most practical and direct approaches to volunteer computing using mobile devices were formulated and theoretically evaluated.

3. **Performance test of prototypes (section 3.3)** – The chosen prototypes were exposed to a series of tests designed to measure their serving capacity and performance. The purpose of this was to determine the efficiency of different setups of the prototype solutions. All test sessions tested one prototype setup with one problem instance type at a time. The tests’ goals, parameters and procedures are formulated in more detail below.

4. **Effectiveness formula (section 3.6)** – After results were recorded and compiled, an effectiveness formula was defined. This formula was then used to predict the effectiveness of other problem instances in order to more clearly see efficiency patterns.

After step 4 had been performed, sufficient results were expected to have been produced in order to answer the scientific question.

3.1 The choice of server problem

Optimally, the problem was required to be easy to quantify the number of operations for, given a certain formula or similar. It was also required to have little or no variation regarding input format. The initial problem was limited to data delivery (such as image hosting), but then refocused to matrix arithmetic which also included processing. Matrix arithmetic however, did not appear proportional processing-wise in comparison to its amount of data transfer. An optimal fit for this problem profile was supervised deep learning training. Deep learning training problem instances can be both big and small, although only
smaller instances were examined in this report, due to the limited capacity for node hardware limitations.

3.2 Design of prototypes

This section describes the properties and components of two different prototype designs used for deep learning training through the use of volunteer computing and explains the design choices behind them.

Main problem: Router & NAT penetration One of the limitations posed by using mobile devices for accepting requests in this thesis was absence of mobile Internet, which limited the devices to Wi-Fi based Internet. Since a Wi-Fi access point in general connects to a local network which in turn connects to the Internet through a router, this posed a problem for the mobile devices in terms of connectivity. In order to serve data through the router one was required to penetrate it to allow data passing through. The NAT in the router does in most cases allow outgoing connections, but not always allow arbitrary incoming connections (see section: 2.5.1 NAT variants). This problem can be solved using two different approaches. One where mobile devices are restricted by the NAT from accepting connections from outside the NAT and one where mobile devices can accept incoming connections through NAT traversal. This is what prototype A and prototype B (described below) are founded upon. This in turn creates characteristics which make them suitable for different environments.

3.2.1 Prototype A: Reverse proxy solution

Assuming that the mobile device resided behind a router/NAT which prohibited all incoming connection attempts directed towards it, some form of intermediary was required to connect a client with a mobile device. This intermediary can be regarded as a reverse proxy server (see word list on section Terminology) and as a dispatcher based load balancing server (see section Dispatcher-based load balancing, although not to be confused with Prototype B).

Differently from prototype B, prototype A is capable of managing the responses of all device nodes (since all requests and responses pass through it), which enables benefits such as response validation, connection loss recovery and coordination of request parallelization. Generally, this solution is more advanced and is preferable where stability and quality is required.

This solution (henceforth referred to as the proxy solution) works by connecting mobile devices to the reverse Proxy Server, which uses the mobile devices as an extension of its back-end. Clients connect to the proxy as a regular HTTP server through a domain name. This enables the server to appear as a traditional server.

The benefit from utilizing a reverse Proxy Server is its ability to resend any failed request sent to a mobile device as well as utilizing mobile devices residing
behind restrictive NATs. This is exclusive to this prototype.

![Diagram](image)

Figure 2: 0: Router/NAT protected LAN, 1: Proxy Server, 2: Mobile devices, 3: Persistent connection, 4: Request from client, 5: Communication over persistent connection, 6: Return response to client

**Proxy server design** The Proxy Server (described in figure 2) allocates a network server socket on a public port for all mobile devices. Mobile devices establish a dedicated TCP connection through their respective NATs using this socket. The connection is kept alive so that the Proxy Server can query the mobile device for processing requests indefinitely when the Proxy Server receives requests from clients. When a client HTTP request is received, it selects a mobile device which can serve the request. Preferably, the fastest possible device with an intact connection should be selected. The Proxy Server relays the request to the mobile device which replied with a response back to the Proxy Server. The response data is then sent to the client as soon as the Proxy Server receives it. Since the proxy communicates through only one socket for multiple simultaneous requests, they must be sent in order for each mobile device. In some cases, this can cause congestion since they have to be queued.

**Mobile device design** Any request should be answered in the shortest period of time possible. The mobile device connects to the Proxy Server on its public network socket and awaits requests made by the Proxy Server. Due to characteristics which benefit from having a connection open (slow start, additive increase/multiplicative decrease) for the TCP protocol (RFC5681) as well
as the proxy’s inability to initiate connections to clients since they are protected by NATs, the connection is kept opened and reused instead of closed.

Computing in parallel The Proxy Server design also allows for parallel computing, since it handles all requests and replies of all mobile devices. This would be in the form of dividing the work and compiling the responses, which would make requests finish faster. The Proxy Server would be under increased stress due to the added work of coordinating the parallelization, which would decrease its capacity (number of requests per second) but it would be able to handle bigger problem instances requested by clients. Read more on parallelization in section 2.3 Parallel computing.

3.2.2 Prototype B: Redirect/Dispatcher solution

Due to its requirement of having enabled NAT traversal capabilities (See NAT traversal on page 17) for the mobile devices’ networks and therefore only handles redirects it is much simpler but also much faster. This server solution can only use a subset of the mobile devices the Proxy solution can (the mobile devices currently capable of traversing their respective NATs). When an HTTP request is received by a Dispatcher Server (described in figure 3) it is redirected to an available mobile device chosen by a load balancing technique. The redirect method used in this solution is a temporary HTTP redirect (302). The mobile device then serves the client on its own, instead of using a Proxy to handle any response data. The use of redirects eliminates the server’s ability to recover from connection loss (since the connection to the server is already terminated) in favour of eliminating the bottleneck of relaying all response data. The server in turn can use this freed up capacity to redirect a higher number of requests instead, which makes it faster than the proxy solution. Since it shares no resources with device nodes, except for synchronization (which is insignificant in comparison, see section 6.3 Connection overhead), the boost in raw request capacity is massive. Provided that the connection does not experience connection problems, this solution outperforms the proxy solution.
Figure 3: 0: Router/NAT protected LAN, 1: Dispatcher Server, 2: Mobile devices, 3: Ready-announcement is sent to server, 4: Request from client, 5: Request is redirected through to mobile devices and answered directly

**Mobile device servers** The mobile devices utilized by the dispatcher solution are similar to the mobile devices utilized by the proxy implementation in terms of connection initialization. They differ in methods of receiving and responding to requests. Since they need to be able to communicate using the HTTP protocol, received requests are formatted as HTTP requests instead of plainly being sent over a TCP connection. This requires the mobile devices to implement their own HTTP servers locally on the mobile devices, so that it can accept redirected connections from clients. This relieves the Dispatcher Server but puts greater demands on the mobile devices due to the requirement of public accessibility when being relayed a request.

**Mobile device exposure** In order to make the mobile device publicly accessible from their Wi-Fi networks, any NAT traversal technique mentioned in section 2.5.2 NAT traversal (STUN, UPnP, etc.) is required. Any real implementation of this was not included in the scope of this thesis project, due to the limited impact it would have on the results.

### 3.2.3 Prototype optimization

This section proposes technical optimizations which most likely would be present in a real implementation of either prototype A or prototype B. For the purpose
of limiting the number of results and its near insignificant impact in a test environment they were omitted from the implementations. The following optimizations are intended for the Proxy solution, but could be used as well for Dispatcher solutions modified to retain a connection between mobile devices and Dispatcher servers.

**Polling algorithm** One method that can be used to mitigate connectivity issues is a polling algorithm, that sporadically tests a device for connection problems. Since these problems are temporary, the server can be expected to increase its knowledge of which devices are connected and which are not. If a device is found to be unconnected, it is not used until it had been confirmed to be working. Should any device surpass the poll failure limit, it will be rendered unusable and disconnected permanently. An example of a polling algorithm is provided below:

```python
DISCONNECT_LIMIT = <chosen_by_the_admin>
failed_connections[ num_of_nodes ]

for every device in devices
    response = poll device
    if not(response):
        failed_connections[ device ] ++
        if( failed_connections[ device ] > DISCONNECT_LIMIT):
            disconnect( device )
    }else if( failed_connections[ device ] > 0){
        failed_connections[ device ] = 0
    }
}
```

Figure 4: Example polling algorithm

**Load balancing (First available)** The following algorithm makes sure that the first available mobile device connected to the proxy is chosen. If it cannot find a device that is available, it sleeps and performs a exponential back-off (increases the sleep time) and subsequently tries again. If the timeout reaches its maximum value, it discards the request and terminates the connection.

The method of determining if the device is busy is different for each prototype solution. For the reverse proxy solution, one can easily see that the device is free to be used whenever it has returned its solution. Regarding the dispatcher solution, one has to rely on statistical guesswork based on average return times (or a retained connection to the dispatcher server, if present).

Following is the load balancing algorithm for the proxy written in pseudo code (timeouts are measured in milliseconds):
MAX_TIMEOUT = 1000
timeout = 100

while( true )
    for every device in devices
        if(device.busy){
            skip
        }else{
            device.busy = true
            process_request
            device.busy = false
            return
        }

    sleep( timeout )
timeout += 100

    if( timeout > MAX_TIMEOUT ){
        discard_request
        close connection
        return
    }

Figure 5: Example load balancing algorithm

3.3 Performance test of prototypes

This section describes all the different tests, what they measured, what variables and parameters are included in them, as well as some test setup details.

All tests were performed in a group of computer labs at KTH's main campus at Valhallavägen (Royal Institute of Technology) at nighttime when there were no other activity on any computer in the labs in order to minimize sources of interference.

Three metrics that measures how well clients can be served were used. The results were then used to determine the utility and quality of the different server solutions:

**Request rate** This is the primary metric which describes how many requests the system can serve during a certain period of time. A high request rate is beneficial.

**Response time** Response time is the time during which a given request is served. Response time is constituted by transmission times and computation times (the time it takes to receive and solve the given problem instance and return the response). The balance of these two depend on
the size of request/response and the amount of the computations a request requires. A high response time is bad if the client expects responses within a certain amount of time.

**Request loss** Request loss is the rate of which requests are unexpectedly lost, unanswered or their response is interrupted. Request loss is a big factor in a systems stability. If the request loss is too high the system could be considered unusable. Thus, a low request loss rate is beneficial.

The goal of this thesis was to improve at least one of these metrics with any of the given prototypes.

### 3.3.1 Test procedure

There are three tests described below (section 3.4). The purpose of each test is to measure the serving performance on different scales of the Proxy solution, the Dispatcher solution and normal server. The normal server acts as a baseline for evaluating solution performance and runs the same request software and hardware as the Proxy and Dispatcher servers and the same processing software as the device nodes. Depending on limitations and result relevance, each test involves relevant server solutions (proxy solution, dispatcher solution and/or normal solution).

Each of the three tests in the test suite served is designed to measure performance of different setups based on their respective server solution (different number of nodes). Each server solution is tried with eight input types in order to see how the system performs under differently sized loads which have different requirements of data transfer and processing. Each input type is sent to the server with varying rates and each rate is continuously sent for 15 seconds so as to get a mean response rate (during which no other requests with other input types are made). This way the performance of each server is measured for different types of input sent at varying rates. From this, performance limits can be found for each server setup. The request rates are roughly chosen at first, and when response drops are detected, more granular ones are chosen. This means that some values below a given request rate which were fully responded to can be assumed to be fully responded to as well. The reader should not assume anything regarding any parts showing response rate drops, since it could be due to temporary problems. Every unique test (server setup, input type and request rate) is only performed once, due to the large amount of time required. The tool used for sending requests is httperf with version 0.9.0.
Memory: 32GB  
Processor: i7-6700 3.4GHz 8-core

<table>
<thead>
<tr>
<th>Memory:</th>
<th>32GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>i7-6700 3.4GHz 8-core</td>
</tr>
</tbody>
</table>

Table 1: Client host hardware specification

The tests yield results showing each server types capacity of responding to requests (model training) for different setups of the systems.

### 3.3.2 Implementations & Technology choice

The following libraries, frameworks and implementations were used in the performance tests (see section 3.3 Performance test of prototypes).

**Proxy & Dispatcher Server framework**  The Proxy Server implementation and the Dispatcher Server implementation used a Java application with the Jetty library to connect to the mobile devices which was chosen because it is lightweight.

**Deep learning library (Neuroph)**  The library which was used for training deep learning models the tests is called Neuroph. This is a small and lightweight library that is fast and properly scaled for the magnitude of problems reasonable for HTTP servers and for the scope of the thesis. Neuroph is also Java-based, which enables Android devices to run it.

**Platform & Programming language**  The implementation of the prototypes in this project utilized groups of Android devices as mobile device nodes, due to its prevalent usage as a smart phone OS in the world (see section 1.3 Limitations). The programming language for writing Android software was Java. Java was also used in the Proxy Server and Dispatcher Server as well as the Normal Server.

**Android emulator**  The Android emulator was used to emulate the behavior of Android devices without requiring physical devices. This made large scale testing much easier financially and practically. They were emulated on Desktop computers running on the Linux operating system (which is more easily customized than other operating systems) in close proximity within the network (so to minimize problems with network connection that could affect the results).

The exact hardware of each emulator host machines and server host were:

### Table 2: Emulator host hardware specification

<table>
<thead>
<tr>
<th>Memory</th>
<th>32GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>i7-6700 3.4GHz 8-core</td>
</tr>
<tr>
<td>Android Version</td>
<td>7.0</td>
</tr>
<tr>
<td>Android API</td>
<td>Galaxy_Nexus_API_24</td>
</tr>
<tr>
<td>Emulator version</td>
<td>26.0.3.0 (build id 3965150)</td>
</tr>
</tbody>
</table>

Real android device

The real Android device was utilized as a performance baseline for the emulated devices in determining performance differences. The exact hardware of the real device used was:

<table>
<thead>
<tr>
<th>Model name</th>
<th>SHIELD Tablet K1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build number</td>
<td>MRA58K.49349_766.5399</td>
</tr>
<tr>
<td>Android version</td>
<td>6.0.1</td>
</tr>
<tr>
<td>Wi-Fi frequency bands</td>
<td>Capable of both 5GHz and 2.4 GHz</td>
</tr>
<tr>
<td>Wi-Fi version</td>
<td>B0_6.10.10.4</td>
</tr>
<tr>
<td>Wi-Fi reception</td>
<td>Full or Very Good</td>
</tr>
</tbody>
</table>

Table 3: Real device hardware specification

### 3.4 Test suite

Below are the three different tests described. In addition to the given server prototypes (Dispatcher- & Proxy server) that is tested the “Normal” (previously described in section 3.3.1) server is also included.

#### 3.4.1 Test 1 – Single device & Return time

The purpose of this test was to determine to what degree a single real device performs (regarding the metrics described in section 3.3) so to calibrate the request rates for test 2 and 3 for any offset posed by using emulated devices instead of real devices. Except for response rate, processing times and return times during normal loads were also recorded.

The emulator software used by regular computers emulate real devices closely regarding functionality, memory and process restrictions, and less so regarding processing speed. The most central effects of this is computation speed and data transfer rates are faster. Results from this test were used to calibrate the other tests to a degree where these differences can be disregarded.
A proxy server with one connected device compared to one dispatcher device

A control solution (a Normal Server with similar software) was tested for comparison. The return time of the dispatcher solution was measured by adding the duration for the dispatcher solution to serve a redirection response to the duration for the mobile device node to serve the response.

### 3.4.2 Test 2 – 35 emulated proxy nodes & Dispatcher performance extrapolation

The purpose of this test was to measure and compare the serving capacity of the prototypes with access to a maximum of 35 mobile devices which is a moderately big but yet manageable number of devices. The number of emulated nodes was limited by the number of computers in the computer labs used (Sport & Spel in the D-building at KTH). The Dispatcher server is tested only on its capacity to redirect requests since it is the bottleneck and not its connected mobile devices. This is because the devices are using independent connections and hardware and is independent of the dispatcher in regarding performance (which is not the case for the Proxy server). By extrapolating successful request rates measured from test 1 with proportional rates for 35 devices (one gets the max performance proportional to the combined emulated proxy nodes). The Proxy server which uses 35 emulated devices is also tested with request rates limited by the same extrapolation (except here the successful request rates for the proxy device is used instead). This mitigates any performance increase which comes from using emulated devices in the results.

<table>
<thead>
<tr>
<th>Server Test Setups</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher Server</td>
<td>0 (x35 extrapolation of test 1)</td>
</tr>
<tr>
<td>Proxy Server</td>
<td>35 Emulated Devices</td>
</tr>
<tr>
<td>Normal Server</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 5: 35 emulated proxy devices (and proxy server) compared to extrapolated data from test 1 for the dispatcher server and a normal server

### 3.4.3 Test 3 – Dispatcher maximum capacity

The purpose of this test was to determine any upper limits to Dispatcher server efficiency (when the server has access to an infinite number of devices) compared to a normal server. This test was performed without any connected nodes, and
because of this, only the Dispatcher server could be tested because it does not require any connected devices (and is tested through extrapolation as in Test 2). The Proxy server is not independent of its devices, and therefore could not be included in this particular test, since any large enough number of devices cannot be acquired or emulated for a proportional test.

<table>
<thead>
<tr>
<th>Server Test Setups</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher Server</td>
<td>-</td>
</tr>
<tr>
<td>Normal Server</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 6: One dispatcher server with an undefined number of devices compared to a normal server

3.5 Test characteristics

Below are some deviations and characteristics explained.

3.5.1 Connection loss

Connection loss occurs when a mobile device (for any reason) is suddenly disconnected while still serving clients, which in turn causes a lower response rate for the solution. It cannot be emulated in any practical or reliable way (since its occurrence can vary depending on multiple factors), although it can be expressed mathematically so to enable usage recommendations to mitigate it.

Consider the situation where one mobile device continuously connects to the system, and is repeatedly is sent a continuous stream of requests. Upon connection failure, a new node will connect. This process is similar to the poisson point process. All disconnect events occur constantly at an average rate and are independent of each other. With these assumptions, we can assume the exponential distribution to model the time before connection failure in the system for any independent request.

The given distribution is then used to calculate the risk of connection loss for sessions of varying length by using its cumulative density function.

Using the cumulative distribution function for the exponential distribution we can calculate the probability of connection loss, given the average connection disconnect rate $\lambda$ and the request length $x$.

Having a low rate parameter (lambda) is optimal. This rate parameter is in the given case synonymous with the average time to failure.

$$f(\lambda, x) = 1 - e^{-\lambda x}$$ (7)
The function $f$ (equation 7) gives the probability of connection loss, given that a request is made, and then disrupted because the device goes offline.

3.5.2 Test input

This section describes the input data format for training deep learning models.

Input size (Training data) The first parameter used to vary data transfer amount was training data (number of training cases) which produce different amounts of data depending on the size of the model. This was used to train the deep learning models and induce different transfer times. Both small and big input data was used in order to scale upstream data transfer amounts and transmission time. It was denoted by either 1 for a greater amount of input data (100 training sets) or 2 for a smaller amount of input data (10 training sets).

Processing (epoch count) Processing amount was decided by a multiplier variable for the training data. By training the model several times with the same data, processing time is increased by a factor given by the epoch count and with a given input data amount. It was varied in two quantities; small and large. This scales processing time without affecting data transfer amounts. It is denoted by either A for a greater number of epochs (20 epochs) or B for a smaller number of epochs (1 epoch).

Output size (model size) Output size was determined by the size and structure of the model used to solve the deep learning problem instance. This was returned after the problem was solved with the output. It was varied in two quantities; small and large in order to scale downstream bandwidth and transmission time. The notation was either X for a larger model size (12) and Y for a smaller model size (5). This parameter indirectly affects the impact of the other two parameters (by increasing the amount of computations needed for every epoch and increasing the amount of output data required to describe the model). A large model size (assuming this also provides a larger input and output size) significantly increases the input size, and correspondingly smaller when using a smaller model size. A larger model size also increases processing time as well as a decreases processing time for smaller model sizes.

Irregularities Some alterations were made to the permutations produced by the variations of the parameters in order to diversify the results with some interesting combinations. BY1 was given an increased number of training cases (from 100 to 230) which is an number arbitrarily chosen number higher than the doubled original value. BX2 was changed to have a massive model size (from 12 to 30) also arbitrarily chosen above its doubled original value. Preliminary tests were also made to make sure the return time was around 1000 milliseconds.

See table 4 below for the exact constants used for the problem.
Table 7: This table shows the exact constants used for the problems

<table>
<thead>
<tr>
<th>problem type</th>
<th>iterations</th>
<th>model size</th>
<th>training cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX1</td>
<td>20</td>
<td>12</td>
<td>100</td>
</tr>
<tr>
<td>AX2</td>
<td>20</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>AY1</td>
<td>20</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>AY2</td>
<td>20</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>BX1</td>
<td>1</td>
<td>12</td>
<td>100</td>
</tr>
<tr>
<td>BX2</td>
<td>1</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>BY1</td>
<td>1</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>BY2</td>
<td>1</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

3.5.3 Input & Output data format

Each problem instance (input data) was formatted as text (since the HTTP protocol is used) when sent to a given server as well as the output data. Figure 6 shows the input data for the problem instance BY2 which is one out of eight problems. As can be seen there, data is divided into rows and delimited by a “hashtag” (#). The first row contains the number of nodes for each layer (so that an empty neural net can be initialized). The second row contains the number of epochs for the training. All subsequent rows are training cases. Because it is supervised learning, it contains a number of input values (determined by the node count of the first layer in the model) and output values (determined by the last layers node count). All training case numbers are randomized and range between 100 and 999 (so that they span the maximum number of different values and have a consistent textual length). Though, the precision of values used for input and expected output within each training case is arbitrary in comparison with the output data.

Figure 6: This is an example of input data, which trains a model with an input layer, 3 hidden layers, and an output layer (all with 5 nodes each). It contains 10 training sets.

Figure 7 shows the output data format. The output data is constituted by the
weights of the deep learning model. The higher precision for the double values used to represent the smaller is the risk for errors in the model. Since full decimal representation could be achieved, all weights retained their full decimal values in binary form encoded with Base64 instead of being sent as human readable text. This leaves it upon future experiments or implementations to lower this precision to gain efficiency. Each weight is represented by its double value, given by 8 bytes encoded in Base64, which converts it to 12 characters of text.

Model:

```
I: 0.5963266668489329 -0.5773568080328152 -0.36854180790775104 0.34076640289355725 -0.06290187448811977
I: -0.41353072719320293 -0.455336095367327 -0.585386443387341 -0.01664047681119974 0.36602832725505846
I: 0.321071333585645 -0.639709537941502 -0.308146195616982 -0.1839233933234 -0.415109689768585
I: 0.020377164720286154 0.5432799638583166 -0.01664047681119974 -0.160903269854802976 -0.160001025622708
I: 0.5467894904933375 -0.31595028990404617 0.03279581732201955 0.186534652245802 -0.07901124079093491
H: 0.3731691289677329 -0.3982026519576713 -0.3284479854840154 -0.3264519724413262 -0.411175640923374
H: -0.41353632826201617 -0.32243865852918693 -0.0570085356544836 -0.2532161339312141 -0.477623581105356
H: 0.4536792326261517 0.1664479446406166 -0.0483261771486894 -0.367697410306853 -0.2177378052244048
H: 0.00070566589630016 -0.422739151614693 -0.130597746181667 -0.612244131766589 -0.0023908638327504
H: 0.5460910328364828 0.7072932326440158 -0.22385971136476834 0.25660816758529860 -0.20113321797792194
H: 0.4609747650179258 -0.414496750543393 0.6683043121519749 -0.31918175644542305 -0.330849124123023
H: 0.742697944165856 -0.29174538656521503 -0.6452001657401564 0.12989675792000205 -0.5204144841874077
H: 0.4381654413300817 -0.7501789330779204 -0.4561183092943743 -0.583790277897788 -0.248862166459303
H: -0.4517704768707604 -0.0542551980406533 -0.326190497015604 -0.3403285803803481 -6.692161567239705
H: -0.4587692320755325 -0.3897999453941209 -0.94751813585823776 -0.120917725353949 -0.260345998905228
H: -0.0275486013985872 0.56301674420247799 -0.0614825778759490 -0.236054673220845 -0.444442412900445
H: 0.39460559364959407 -0.03501180431101015 -0.62047064778393 -0.0500019778992256 -0.15699081913967
H: -0.51174308608205357 -0.23105801665122078 -0.4990457485065727 -0.3105645074674623 -0.148000197301324
H: 0.53658761074434357 -0.5584901312932 -0.229623079898208 -0.0479783990292083 -0.132871547946119
H: 0.07715558810708654 -0.6718609394088787 0.3253688821452199 -0.56514295050917639 -0.677220147949892
```

Figure 7: This is an example of decoded and slightly formatted output data, which contains weights for 5 layers (each row starting with “I:” represents an input node and its weights, each row starting with “N:” is a hidden node and its weights). In a real output, only the weight value data in binary format converted to base64 format would be included.

```
Data transfer sizes
Figure 8 displays the amounts of data that is sent upstream from the client (httperf) to the different solutions and the amounts of data sent back downstream. Given the “model size” value (which is the number of input nodes, layers in the model, and output nodes) together with the number of training cases, input data amount can easily be reproduced. Output data amount scales with the number of weights. A more exact description can be found in section 3.6.3 How to calculate output time.

3.6 Effectiveness formula
When the effectiveness for each input type has been determined, one can construct a formula for effectiveness prediction, which can be used to extrapolate and make guesses about efficiency for untested input types.

When a request containing any input type is sent to a server of any type, one can expect it to need more or less time to complete, depending on how fast it processes training and how fast it reads and writes data to and from the client. The formula is an approximation of how input parameters comprise time demands. It is constructed by using the parameters from the different input types together with the measured time needed per request (which is produced from the inverse of the serving rate for each particular input type).

The three primary factors of time consumption is input-time, processing-time and output-time. The input-time represents the time to receive and parse the HTTP request from the network socket. The processing-time represents the time required to setup the problem, memory allocation for the data structures and the computations themselves. The output-time represents the time to write the HTTP response to the network socket.

The serving rate could be expressed as a linear regression model (where $i$ is the element proportional to the time needed for input, $p$ is proportional to the processing time, and $o$ proportional to the output time) and $a$, $b$ and $c$ are weights. In order to normalize the result, we utilize a multiplier $m$ and overhead $h$ to form the rate $r$:

\[
r = \frac{m}{ai + bp + co + h} \tag{8}
\]

Two parameters used for modelling characteristics of the particular TCP connections (which are used to calculate the input-time $i$ and the output-time $o$ before linear regression is used) are not known (additive increase increment and maximum transfer rate), however all other parameters required to calculate the formula are known.
The first step is that random values for additive increase rate and maximum transfer rate are chosen. This enables the calculation of input-time, processing-time and output-time. When these are calculated, their weights can be calculated using the least-squares method in order to estimate their relative size coefficients.

The slightly less impactful values for overhead and multiplier are then randomly chosen, as with additive increase increment and maximum transfer rate. The last step is calculating a serving rate. This serving rate is then compared to the real serving rate (given the same input parameters), which is used to calculate an error.

By repeating this process, and randomly choosing different values for all the unknown variables (except for the weights), we can minimize this error. The assumption used here is that the better fit produced, the better variables and formula. Any produced variables can be inspected for greater abnormalities, which ultimately gives us a crude but good enough tool to view patterns in serving rates based on different parameters (considering the lack of time and lack of data).

3.6.1 How to calculate transfer time of a given number of bytes

Since transfer is done through the use of TCP connections, they incorporate mechanisms to optimize transfer speeds, by using slow start, additive increase and multiplicative decrease (as described in the background). For both input and output we are using a formula which takes slow start and additive increase into account (though not multiplicative decrease as it is just replaced with a maximum data transfer speed constant). It is a function that takes the number of bytes, the increase parameter and the parameter max transfer rate-parameter and the number of bytes, so that data transfers become more effective as time progresses.

The reason for this is that requests with very big amounts of data have a different average transfer speed, which is hard to reconcile with smaller average transfer speeds, so the model must take this into account when calculating time in order to minimize its error.

The model assumes a linear increase in the transfer rate, with a constant maximum transfer rate instead of a continuously peaking and decreasing transfer rate (the transfer rate would on average match a continually increasing and decreasing transfer rate peaks).

When using additive increase for calculating transfer speed, it requires us to construct a linear function of transfer speed:

\[ f(t) = k \times t \]  

(9)
Where $k$ is the additive increase factor and $t$ is the time. We also define a max transfer speed $d$. The time needed to reach this max transfer speed is $d/k$. The number of transferred bytes $b$ is the integrated version of the function $f$:

$$F(t) = \frac{k \cdot t^2}{2}$$

under the assumption that the number of previously transferred bytes are 0. The amount of time to transfer a certain number of bytes using the additive increase is therefore:

$$t(b) = \sqrt{2 \cdot b/k}$$

The number of bytes to reach the maximum transfer rate is calculated:

$$\sqrt{\frac{2b}{k}} = \frac{d}{k} \Rightarrow b = \frac{d^2}{2k}$$

When you then want to calculate the time needed to transfer all bytes, you either have reached the maximum transfer rate, or you have not. If you have not, $t(b)$ is used. if not, the time to reach the transfer speed limit is added to the time it takes to transfer the remaining bytes at constant speed:

$$\frac{d}{k} + \frac{b - (d^2/2k)}{d} = \frac{b}{d} + \frac{d}{2k}$$

$$t = \begin{cases} \frac{b}{d} + \frac{d}{2k}, & \text{if } b > \frac{d^2}{2k} \\ \sqrt{\frac{2b}{k}}, & \text{if } b \leq \frac{d^2}{2k} \end{cases}$$

3.6.2 How to calculate input time

The input time is based solely on the number of bytes that is transferred upstream to the server and is produced by the method of calculating transfer time for a given number of bytes. The number of bytes is comprised by a description of the model and of training cases. The description starts with a number of integers describing each layer’s node count (notice the digit count variable which is used to count the number of digit bytes), then one integer describing
the epoch count. The training cases in turn are constituted by a number of 3-digit integers. This number of integers is produced by the number of input nodes added to the number of output nodes. Since each integer is 3 digits, this means 3 bytes to represent it. For every number there is a delimiter (either '#' or a blank space). In the following number of bytes for input as a function \( b \) taking a vector \( \vec{v} \) containing each layers node count, an integer \( e \) containing the epoch count, and a matrix \( M \) containing \( r \) rows of training case integers. The function \( d(x) \) gives the number of bytes required to represent any integer \( x \).

\[
b(\vec{v}, e, M) = d(e) + \sum_{i=1}^{n} d(v_i) + \sum_{j=1}^{r} \sum_{i=1}^{v_1 + v_n} d(m_{ij})
\]  

(15)

3.6.3 How to calculate processing time

The amount of processing is determined by the following formula, based on the definition of deep learning forward and backward pass (see section 2.2.1 Multilayered feedforward neural networks). Here, the exact number of CPU operations is irrelevant, since we only want a relative measurement unit of processing. The node error \( e_j(n) \) can be considered constant for each node \( j \), since it is the same number of computations every time. This makes \( \epsilon(n) \) be \( j \) operations, since it scales with the number of nodes in adjacent layers. The weighted sum for the \( j \)-th node \( v_j \) scales with the number of nodes \( i \). The output \( y_i \) for node \( i \) is determined by a previously calculated value, which makes the weight change \( \Delta w \) a constant \( c \) number of operations. This means that for between each layer \( n \) (containing \( j \) nodes) and layer \( n + 1 \) (containing \( i \) nodes), the number of operations can be expressed as an equation when disregarding the small and constant values:

\[
(n - 1) * j * i * c
\]  

(16)

3.6.4 How to calculate output time

The output time is produced by the method of calculating transfer time for a given number of bytes. The number of output bytes is calculated from the number of weights between the \( n \) layers (containing with \( j \) nodes) and their following layer (with \( i \) nodes). Each weight then uses one double value, which is represented by 8 bytes (which in base64 representation needs about 33% bytes extra). The formula for the total number of output bytes (when also including
node counters integers for each layer and a layer count) is:

\[ 1 + \left( \frac{1}{3} \right) \times (n - 1) \times j \times i \times 8 \]  

(17)

### 3.6.5 Comparing performance in different platforms

Given the fitting procedure and the formula we can then calculate the efficiency of both the prototypes and the normal server. Since the dispatcher prototype is expected to be significantly more efficient, it is used as a benchmark against the normal server. The relative performance is produced by dividing the performance of the normal server with performance of the dispatcher node.
4 Results

This section contains graphical data on the performance of the tests performed and efficiency comparisons based on the tests. All graphs which compare response rate and response rate are measured per-second.

Note that some values are interpolated between data points, mostly where the response rate is 100% (under the assumption that all request rates under another request rate which respond with 100% efficiency will also be 100% effective). All request rates must have a 99% response rate in order to be considered a valid capacity for a given test.

4.1 Test 1 - Dispatcher/Proxy single device capacity comparison

This section includes small scale tests results which demonstrates the effectiveness for one mobile node. This is useful for the creation of projections on larger scales and for unit baseline measurement and future development. The eight problem types described in table 7 are used as load.

![Graph showing response rate vs. request rate for Dispatcher and Proxy solutions.]

Figure 9: AX1 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

As can be observed in figure 9, both solutions handled one request successfully, and then became overloaded.
Figure 10: AX2 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 10 reveals slightly better dispatcher performance and peaked when handling 10 requests and the Proxy Server handled 8 requests.

Figure 11: AY1 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 11 shows that the Dispatcher Server handled up to 13 requests and the proxy handled 8 requests.
Figure 12: AY2 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 12 reveals that the Dispatcher Server handled almost 5 times more requests than the Proxy Server (with up to 53 requests compared to 11 requests).

Figure 13: BX1 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

According to figure 13, the Dispatcher handled up to 13 requests when the proxy handled up to 10 requests.
Figure 14: BX2 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 14 reveals that the Dispatcher served up to twice as many requests compared to the Proxy Server (1 and 2 for the proxy and dispatcher, respectively).

Figure 15: BY1 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 15 shows that the Dispatcher served was almost twice as effective as the proxy solution (17 requests compared to 9, respectively).
Figure 16: BY2 input sent to a Dispatcher Server, a Proxy Server setup with only one node connected.

Figure 16 shows that the Dispatcher Server served 79 requests (more than 5 times as many requests) whilst the Proxy Server successfully served nearly 18 requests.

4.2 Test 2 - 35 device capacity comparison with Normal Server

This section demonstrates the effectiveness for the server types using 35 nodes using the same input as for test 1. Note that the following section shows results for the Proxy Server type, which used emulated devices. It has only been tested with the capacity the real device could effectively manage (extrapolated for 35 devices). A few results showed no definite maximum, but are represented at the end of the Proxy Server graph lines. Also included for comparison are the extrapolated dispatcher node performances.
Figure 17: AX1 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 17 shows that the Normal Server outperformed the other servers greatly with nearly 195 successfully served requests, compared to 35 requests for both the Proxy and Dispatcher. It should be noted that only values up to 35 are valid, since higher request rates than that has not been shown to proportionally perform well when testing with a real device.

Figure 18: AX2 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 18 shows that the Normal Server outperformed the other servers greatly with nearly 1350 successfully served requests, compared to around 280 requests and 350 requests for the Proxy and Dispatcher respectively.
Figure 19: AY1 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

According to figure 19, the Normal Server outperformed the other servers greatly with nearly 1800 successfully served requests, compared to around 280 requests and 455 requests for the Proxy and Dispatcher respectively.

Figure 20: AY2 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 20 reveals the Normal Server outperformed the other servers greatly with nearly 5000 successfully served requests, compared to around 385 requests and 1855 requests for the Proxy and Dispatcher respectively.
Figure 21: BX1 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 21 shows that the Normal Server outperformed the other servers greatly with nearly 2100 successfully served requests, compared to around 350 requests and 455 requests for the Proxy and Dispatcher respectively.

Figure 22: BX2 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

According to figure 22, the Normal Server outperformed the other servers greatly with nearly 180 successfully served requests, compared to around 8 requests and 70 requests for the Proxy and Dispatcher respectively.
Figure 23: BY1 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 23 shows that the Normal Server outperformed the other servers greatly with around 4500 successfully served requests, compared to nearly 350 and 595 requests for the Proxy and Dispatcher respectively.

Figure 24: BY2 input sent to a Dispatcher Server, a Proxy Server and a Normal Server to measure the maximum capacity. Proxy and Dispatcher are computed with 35 nodes connected.

Figure 24 shows that The Normal Server outperformed the other servers greatly with nearly 5000 successfully served requests, compared to around 420 requests and 2450 requests for the Proxy and Dispatcher respectively.
4.3 Test 3 - Dispatcher & Normal maximum capacity comparison

This section displays the results performed on the Normal and Dispatcher Server. The request rate was maximized, in order to try and show the maximum capacity of each respective solution. Please notice that many results have a gradual efficiency decline, which makes their response rate curves appear more efficient than they are. All mentioned response rates have at least 99% efficiency.

Figure 25: AX1 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

According to figure 25, the Normal Server successfully handled up to 195 requests. The Dispatcher handled up to about 3000 requests.

Figure 26: AX2 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

According to figure 26, the Normal Server successfully handled up to 195 requests. The Dispatcher handled up to about 3000 requests.
Figure 26 shows that the Normal Server successfully handled up to about 1350 requests, the Dispatcher handles up to about 5000 requests.

Figure 27: AY1 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

Figure 27 reveals that the Normal Server successfully handled up to about 1800 requests, the Dispatcher handled up to 3000 requests.

Figure 28: AY2 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

Figure 28 reveals that the Normal Server successfully handled up to about 5000 requests, the Dispatcher handled up to 4000 requests.

According to figure 28, the Normal Server successfully handled up to about 5000 requests, the Dispatcher handled up to 4000 requests.
Figure 29: BX1 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

Figure 29 shows that the Normal Server successfully handled up to about 2100 requests, the Dispatcher handled up to 2000 requests.

Figure 30: BX2 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

Figure 30 reveals that the Normal Server successfully handled up to about 180 requests, the Dispatcher handled up to 4000 requests.
Figure 31: BY1 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

According to figure 31, the Normal Server successfully handled up to about 4700 requests, the Dispatcher handled up to 4000 requests.

Figure 32: BY2 input sent to a Dispatcher Server and a Normal Server with a maximized request rate.

Figure 32 shows that the Normal Server successfully handled up to about 5000 requests, the Dispatcher handled up to 5000 requests.
4.4 Problem specific data

In this section results regarding the execution of the problems are revealed. As can be seen in figure 33 AX1 had a calculation time of around 1000 ms for Dispatcher nodes. BX2 had slightly higher than half that, whilst the others had nearly 10% of AX1. The least demanding problem instance is BY2, with around 2% of AX1. Figure 34 reveals that AX1 had the highest return time, with nearly 1500ms for the Proxy solution and above 1100ms for the Dispatcher solution. BX2 had nearly 1000ms and 700ms for the Proxy and Dispatcher solution, respectively. All other problem instances had less than 200ms for the Dispatcher solution, and between 400ms and around 100ms for the Proxy solution.

Figure 33: Processing time for each problem type when using a dispatcher node. Note that this is fairly representative also for the proxy node since they utilize the same processing hardware and software.
4.5 Connection loss probabilities

Below can be seen the calculated probabilities for interruption (using the formula described in equation 7), i.e. the probability of being spontaneously disconnected from a device that is being queried.

Each row (except the top-most one, and except the left-most value) represents failure probabilities for a given request duration of which each value represents the failure probabilities for a specific online time.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$x=0.001$</th>
<th>$x=0.01$</th>
<th>$x=0.1$</th>
<th>$x=1.0$</th>
<th>$x=2.0$</th>
<th>$x=3.0$</th>
<th>$x=5.0$</th>
<th>$x=10.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.1%</td>
<td>0.01%</td>
<td>0.0033%</td>
<td>0.0017%</td>
<td>0.0008%</td>
<td>0.0003%</td>
<td>0.0002%</td>
<td>0.0001%</td>
</tr>
<tr>
<td>0.1</td>
<td>0.995%</td>
<td>0.1%</td>
<td>0.033%</td>
<td>0.017%</td>
<td>0.0083%</td>
<td>0.0033%</td>
<td>0.0017%</td>
<td>0.0002%</td>
</tr>
<tr>
<td>0.03</td>
<td>9.5%</td>
<td>1.0%</td>
<td>0.33%</td>
<td>0.17%</td>
<td>0.083%</td>
<td>0.033%</td>
<td>0.017%</td>
<td>0.0033%</td>
</tr>
<tr>
<td>0.017</td>
<td>63%</td>
<td>9.5%</td>
<td>3.3%</td>
<td>1.7%</td>
<td>0.83%</td>
<td>0.33%</td>
<td>0.17%</td>
<td>0.027%</td>
</tr>
<tr>
<td>0.0083</td>
<td>86%</td>
<td>6.4%</td>
<td>3.3%</td>
<td>1.7%</td>
<td>0.66%</td>
<td>0.33%</td>
<td>0.17%</td>
<td>0.027%</td>
</tr>
<tr>
<td>0.0033</td>
<td>95%</td>
<td>9.5%</td>
<td>4.9%</td>
<td>2.4%</td>
<td>1.6%</td>
<td>0.5%</td>
<td>0.17%</td>
<td>0.027%</td>
</tr>
<tr>
<td>0.0017</td>
<td>99%</td>
<td>9.9%</td>
<td>15%</td>
<td>8.0%</td>
<td>4.1%</td>
<td>1.7%</td>
<td>0.83%</td>
<td>0.17%</td>
</tr>
<tr>
<td>0.0002</td>
<td>100%</td>
<td>63%</td>
<td>28%</td>
<td>15%</td>
<td>8.0%</td>
<td>3.3%</td>
<td>1.7%</td>
<td>0.17%</td>
</tr>
</tbody>
</table>

Table 8: Total probability of interruption for a given online duration and request duration. The leftmost column describes the average request period $x$ in seconds, the topmost row describes the average interruption rate $\lambda$ (in interruptions per second) for a mobile device. The values are described with up to two significant figures.
4.6 Efficiency comparison

Using data from all the normal server-results and the dispatcher node results (the request rates displayed are those which can be served with at least 99% efficiency). Both are modelled within 30% of their correct performances.

The Dispatcher node was fitted with an overhead value \((o = 17358)\), a multiplier \((m = 1197890)\), a max transfer speed value \((d = 9.8)\) and an additive increase value \((k = 0.70)\). These input, output and processing were balanced with the following values \((a = 27, 0, b = 14, 4\) and \(c = 0, 273)\).

![Dispatcher node performance](image)

Figure 35: A comparison of the real performance and the performance predicted by the model for the Dispatcher node.

The Normal server was fitted with an overhead value \((o = 139, 76)\), a multiplier \((m = 1266391)\), a max transfer speed value \((d = 824)\) and an additive increase value \((k = 74.7)\). These input, output and processing were balanced with the following values \((a = 0, 575, b = 14, 8\) and \(c = 0, 00136)\).
Figure 36: A comparison of the real performance and the performance predicted by the model for the Normal server

When using the respective models to compare the efficiency of the dispatcher and the normal server, the following heatmaps were created (with 1, 10 and 20 epochs, respectively):
Figure 37: Heatmaps with the number of Dispatcher nodes needed to match performance for the Normal server. The gradient bar to the right of each figure display the exact numbers of devices needed.
5 Discussion

This section analyzes and discusses the results in order to explain and simplify them.

5.1 Dispatcher vs. Normal server

When at first comparing the capacity of the normal server and the dispatcher server, the biggest difference in relative performance is for the BX2 and AX1, which are both big input types (AX1 has 10 times the processing of any other type and BX2 has about 10 times more data transfer than any other type). The Dispatcher server handles requests 15 times faster for AX1 input and 22 times faster for BX2 input compared to the normal server. Other input types are just 3 times faster or less for the Dispatcher server. AX1 and BX2 are especially suitable input types when comparing the server elements. One especially notable detail about the three most effective input types (in terms of redirectability) is that they also have the biggest return times (as a consequence of being bigger requests).

The potential speedup is calculated from the number of redirects possible divided by the number of responses by the normal server made during a given amount of time. Multiplying with the number of dispatcher nodes needed to match the normal servers capacity for BX2 input (90) yields just under 2000 devices. As is easily deduced from the observation that when the request is served faster and the time to complete the response converges towards the time required to redirect a client, the potential speed up decreases.

When observing the greymap figures, one can see that requests with smaller model sizes (the model size parameter as defined in section 3.5.2 Test input) generally are more effective than the other input types (when to a dispatcher node), and for any bigger model sizes only with smaller numbers of training cases. If one looks carefully, there are diagonal lines in the greymaps which suggest that bigger model sizes might be more effective. Problem instances with higher number of epochs also seem to be relatively less effective when using the dispatcher.

5.2 Proxy vs. Dispatcher

The effectiveness of the dispatcher solution compared to the proxy at a 99% response rate was the same for AX1 and only 20% as effective for AY2. This is the performance for the proxy solution with only one device, but since there is no clear reason the server would be able to utilize their devices more efficiently when it has a higher number of devices connected, it can be considered inferior. With 35 emulated devices for the proxy solution, all request rates were successfully handled for a request rate 35 times the highest successful request rate for one device except for BX2, where the performance dramatically drops.
This is not surprising, since the amount of data that is transferred is enormous and it is likely congested in the proxy. This is not evidence for either that it can replace a single server or that it is too weak, which means that only larger scaled projects will be able to properly answer this (since a device ratio close to at least 90 devices per server will match the normal server’s performance). The Proxy has greater advantages when it comes to computing in parallel, since it can coordinate the division of work as well as compiling answers. Applications for this could be tasks which require interruption recovery and reliability where computational power is the limiting factor (not data transfer). If one cared only for return time and also implement parallel computing, the proxy would be superior.

5.3 Connection overhead

The process of connecting devices could be a cause for performance issues if the average online time is low. Connecting a device is a matter of performing a TCP connection and transferring data in the size order of 100 bytes. It is less than the smallest input data, and should logically therefore only take milliseconds. Comparing to the order of online times higher than a couple of seconds, this makes the effort of synchronization less than 1 part per thousand of the capacity offered, and can thus be considered a comparatively negligible overhead.

Should the connection be interrupted, the request needs to be resent by the client (or restarted by the proxy). As can be seen in table 8, one should make sure to keep devices online for a minimum of 10 minutes (600 seconds) when having requests with duration 1 second (which was the maximum request duration used in the tests) to decrease the risk of spontaneous connection loss to below 0.16%. The proxy solution can recover from this, although it would sometimes increase response time.

5.4 Efficiency model evaluation

The efficiency model is able to predict the serving rate within 30% of its correct value. Though there are many unknown parameters and variables, this is still a useful tool for making rough estimates, but requires empirical data as well as time for fitting possible values to the unknown variables before every use. The noise that could have disturbed the results is limited to only wi-fi disturbances, as no other resource demanding background-process was executed in either part of the test setup. 30% performance difference being caused by a wi-fi connection could be reasonable, but for the normal server (which use only cabled connections) it is not relevant. Since the models only use results which are limited to a 99% effectiveness, every setups performance are skewed by their respective efficiency declines. Consider for example where one setup serves 10 requests per second with 99% efficiency, and 11 requests per second with 10% efficiency,
while another setup performs 10 requests per second with 99% efficiency, and 20 requests per second with 98% efficiency. Even though the capacity is the same with an efficiency condition, they differ hugely in the number of requests they are able to serve. Any future model should take this into account.

One should also observe the difference in the amount of mobile devices needed to match the performance of a normal server increases with a higher number of epochs. This is most likely due to request timeouts since a mobile device takes longer to complete requests when tasked with multiple simultaneous requests.

5.5 Report weaknesses

One weakness of this thesis is its stable high-performance network connectivity in the tests (due to having the tests performed in a local area network). Considering that all solutions are intended to be used over the Internet, the test results could be considered optimistic since all requests should be faster in the tests compared to a real world application.

This thesis utilized only one real device for a test which results are extrapolated. In a real world application, the device diversity that should be found among the volunteers should reflect consumer popularity more. It is unreasonable that one single device type should closely represent all possible devices (in terms of average processing capacity, communication speed, etc.). To acquire the best possible result, one should have used a set of real devices instead of Emulators altogether. This would have eliminated any discrepancies that may be found in this report.

The version httperf used in this thesis had a maximum request size of around 10kB, which can only yield a certain size of responses. Should another program have been used for load generation, perhaps other relevant results would have showed.

5.6 Possible improvements & Future work

Future work should focus on increasing the practical instance size limit for prototype A and B. This could be done using for instance peer-to-peer communication, which could alleviate the work of dividing the problems and compiling the results. Future work should also consider additional problem instances for deep learning and should also consider other problem classes than deep learning.

The scale of the tests performed within this thesis could have been more substantial which implies having the proxy solution tested with a number greater than 35 devices. The results suggest that there could be more relevant results with as far as up to 10 times more devices.
Should the problems be scaled up, they could be computed using a longer expected response time. Due to the various timeout limits for different implementations of HTTP clients which range between 30 and 300 seconds, the requests could be between 10-100 times bigger. This would probably not be enough for any heavy deep learning applications such as image recognition, but might be applied to problems similar in size to modelling of complex mathematical functions.

With the performance shown in the results, its prerequisites are about 80-90 mobile devices (with performance similar to the Nvidia SHIELD K1 tablet) to match a server with 32GB memory and i7 processor in performance. This can be purchased for approximately 9 000 SEK. This would mean that every mobile device would be worth approximately 100 SEK over the lifespan of a server machine. Of course, this is disregarding infrastructure costs such as internet bandwidth and electricity.

The hardware used for the proxy and Dispatcher Server were relatively powerful in comparison with other hardware. The value of assisting devices varies under the assumption that a low performance hardware has a better performance to cost ratio than high performance. If the price of servers for example would exhibit a quadratic growth in relation to the performance, the value of each assisting device would be higher when achieving higher performance. Should a cheaper server prove to have a better response rate in comparison to the cost of the hardware (when utilizing assisting mobile devices), the comparable value of each mobile device would be higher.

One method of improving this system would be to let the device nodes utilize their GPU hardware to solve the problem instances. As mentioned in section 2.7 GPU vs CPU for computing deep learning there would likely be clear improvement, since they naturally are more adapted to the given kind of computation.

One possible improvement could also be a more efficient encoding for delivering double values over HTTP. Since this is a text based protocol no easy method to compress this exist. Although this would probably incur a small time penalty for encoding/decoding, it might improve performance.

Security is also a point that should be regarded for any practical implementation of this system. Since the system deals in connecting vast numbers of mobile devices to a single server, one should remember that there are ample opportunities for attackers. To mitigate the risks, one could utilize encryption for in-transit data, which would prevent any attacker to easily eavesdrop on any sessions. This is given that any data that is sent is worth protecting, otherwise one should remember that adding encryption to server nodes that are taxed enough already will affect performance. The system is also partly open for review and modification, due to the app being distributed. To solve this, one would need to validate the identities of the nodes before using them to compute...
important answers. The risk of having any malicious attacker in the system could be mitigated by using some kind of identity verification before submitting a node to the system. Overall, one should consider the systems to be an extension of any general HTTP server, which should make the reader consider solutions common with other types HTTP servers.

5.7 Deep learning & other problems classes

There are no obvious reasons why other problems classes would not work with this solution. One qualifying criteria for computing a problem using these solutions is ensuring that the bottleneck is constituted by number of computations and size of downstream data, rather than size of upstream data. As long as those conditions are met, any problem class would probably be qualified, perhaps even more than deep learning. This thesis have not considered how much deep learning can be parallelized, similar to other volunteer computing systems (e.g. Folding@home). One should take this into consideration as it would have the potential to also reduce response time.

The system was only tested using smaller problem instances, which is fitting considering the hardware limitations. This means that it is not likely that it can be a substitute for a conventional server system for solving bigger problem instances.

5.8 Using mobile devices vs Regular computers

This report is exclusively focused on mobile devices. Should this be used in addition to regular computing hardware under the same circumstances, one could easily observe that both computing power, memory, and bandwidth/transfer speed would be greatly increased. The difference otherwise (in terms of NAT issues and connection to the servers) would probably not be that exceptional, as they both are computers connected to the internet through the same gateway.

5.9 Commercialization

In a real world application, hardware probably would be more diverse and could the performance of other devices could only be speculated upon. Devices with other operating systems might be more or less efficient depending on how well the software is adapted to the operating system. This project used Android, but if it would for example have been done with iOS, performance probably would have differed, since they have different OS kernels.

A commercial implementation could upset the balance of other solutions offering server processing power and keep prices down, as well as offering cheaper solutions for organizations requiring hosting capacity with similar use cases as described in this thesis.
6 Conclusion

The scientific question of this thesis was formulated as follows: “Can mobile devices be used in a seamless manner to assist regular HTTP servers tasked with solving deep learning training problems and in so doing increase the collective serving capacity?”.

Of what have been observed, some positive results were observed when calculating deep learning problem instances, although negative results were observed as well. The highest increase in efficiency when using the dispatcher solution was measured up to 2100% (given a 99% response rate for the given request rate), provided just below 2000 volunteer mobile devices. The produced results and subsequent analysis suggest positive results under certain conditions such as scaling more in model size than in input amount or epoch count. If the mobile devices assume the computational burdens of the server, at least 80-90 devices (for the problem instances and hardware used in this thesis) should be used in order to gain capacity compared to using a regular server only. Other capacity proportions may exist for other server types.

A model was constructed which could be used to determine response rate, although it requires calibration with its target device beforehand. The model was able to predict real maximum response rates within 30% of the real maximum response rates. This is given that the response rate is at least 99%. Since different problems/servers have different response rate drop-offs, the absolute max response rates does not match the rates at 99% all the time and could be a cause for the 30% discrepancy.

Response time is most often increased (although the increase could be limited or even replaced with a decrease using parallel computing). The proxy solution should be used for problem classes not encumbered by significant amounts of upstream or downstream data in order to be efficient. The Proxy solution is also recommended for services which require failure recovery. Should the mobile devices be restricted by NATs which does not allow NAT-traversal one cannot use the Dispatcher solution, and must therefore use the Proxy solution. If users require only increased serving capacity the Dispatcher solution can be used given that upstream data is not a limiting factor. This report does provide as good guarantees for the efficiency of the proxy as for the dispatcher because exhaustive benchmarks could not be performed, and should be performed if any further implementation of this technique is to be used.

Any future attempts relating to this research should use real devices, parallel computing and attempt to use GPUs when dealing with deep learning problems, as well as focusing primarily to improve processing, since it seems to be the greatest factor in boosting performance.
References


