An Asynchronous Based Approach to Improve Concurrency Control in Mobile Web Servers

Sudipan Mishra, Siddharth Sarasvati, Shashank Srivastava, Xumin Liu
Department of Computer Science, Rochester Institute of Technology, USA

***Abstract*—The recent boom in mobile computing has created a juncture where it is now possible to host web services on a mobile web server. However, there remain challenges due to limitations in available mobile hardware and software capable of managing resource intensive applications. This paper addresses the issues related specifically to concurrency control improvement in mobile web servers. We propose a Dynamic Resource Management Strategy (DRMS), which uses an asynchronous based approach to reduce the number of discarded incoming client requests by a mobile web server. The DRMS strategy includes identifying the incoming *heavy requests* from the clients, which require relatively heavy system resources on the mobile server to generate a response, and adding those requests to a heap for them to be processed asynchronously. Based on I-Jetty we designed and developed Dynamo, an implementation of DRMS, for Android-based mobile devices. We conducted an experimental study on Dynamo. Our results demonstrated the effectiveness of DRMS with the improvement in Dynamo's concurrency control management as compared to the same in I-Jetty.**

***Index Terms*—Mobile Web Services, Web Services, Mobile Web Servers, Concurrency Control, Jetty, Asynchronous Handling.**

I. Introduction

In 1990, Time Berners-Lee invented the World Wide Web and the world's first web server, CERN httpd, was born. More than twenty years later most of our applications are driven by web services hosted on variety of web servers [15]. The spread of web applications has significantly altered the Web landscape [7]. Leading Internet Technology (IT) companies such as Google, Amazon, Facebook and Twitter provide a wide array of web services such as Google Product Search, Amazon eCommerce Application Programming Interface (API), FaceBook API and Twitter API that use Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) protocols supported on web servers to provide a great web experience for the clients all over the world. Meanwhile, the past years have witnessed a boom in wireless technologies [12]. Recent wireless broadband technologies, such as Wi-Fi, WiMaX, UWB and 3G/UMTS, 4G, are bringing the promise of large bandwidth everywhere. This has significantly encouraged the usage of mobile devices across the globe. We are going through a mobile evolution where the line between mobile and traditional computing is getting thinner. New generation smartphones carry more processing and memory power than the traditional servers did ten years ago. With the recent innovations in the mobile device industry, the consumers are moving towards a post-PC world, where a mobile device is capable of performing most of the essential functions of a traditional computer [6], [11] and with accelerated internet adoption around the world, increasing accessibility of communication services such as maps, instant messaging, social networking, online shopping and news on mobile devices, and improvements in mobile device features such as screen size, internal processing power, and data speed transfers, more consumers today prefer a mobile device over a traditional desktop [5], [9].

Although first developed by Nokia in 2007 [1], the concept of mobile web servers is still new [13]. A mobile web server is a software similar to Apache HTTP Server, or Apache Jetty Server, that provides a mobile device, such as a smart phone or a tablet, the capability, to host web sites, web applications, and web services. Hosting web services on a mobile server will bring flexibility, convenience, and potential to the consumers. This will revolutionize the way we communicate, share data, and gather information. Mobile web servers face unlimited possibilities in the areas of enterprise mobile data access and sharing for lucrative markets such as the military, music and health care, to name only a few. Engineering mobile devices around the world as mobile web servers would change the internet landscape. It would fundamentally alter the way we access web services such as maps with geo-location information, documents, medical records and news, etc. Mass transit systems could use a mobile server hosting a geo-location service providing real-time location, scheduling and route information to commuters. A mobile service such as this would be beneficial to supply chain management systems, where shipped products could be accurately tracked across the world. In the military, solders in the battlefield would be able to use smartphones with mobile servers to share information, communicate and gather intelligence from other soldiers in the vicinity, without depending on satellite communication which may have a certain latency. Skilled people like doctors and nurses can also use a mobile web server in remote areas, in emergency and disaster situations, to obtain and share patient information.

Yet, mobile devices are still undergoing major changes and thus, still lack resource capabilities. In addition, battery life is limited in a mobile server. In addition to a slower processing power (as compared to a desktop), less memory and limited battery life, the concurrency control management on a mobile server also faces several issues. For example, because of its limited thread pool and memory capacity, if the number of concurrent clients accessing a web service hosted on a mobile

[1] http://research.nokia.com/page/231

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States

DOI 10.4108/icst.collaboratecom.2012.250427

server increases, the device is unable to handle these multiple requests. As a result, the mobile web server discards the client requests. With better concurrency control, it is possible to decrease the number of discarded requests and thus increase the reliability and availability of the mobile servers. Motivated by the new innovations in mobile technologies and by the potential benefits of a mobile server, this paper outlines ideas and strategies to address the concurrency issues of a mobile web server.

Existing research in the field of mobile web services outlines key components for building a framework for using a mobile device to host mobile Web services. Mizouni, Serhani, Dssouli, Benharref, and Taleb in [8] propose an architecture that deploys Web services on mobile platforms. They were also able to identify and evaluate factors that affect the quality of service of mobile devices such as response time, availability, throughput and scalability. In addition to their proposal of a framework for hosting complex Web services on mobile devices, Hassan, Zhao and Yang in [4] proposed a partitioning framework that provides a high level design about how to delegate heavy-duty tasks to distributed backend servers. Other work in the field is more related to the comparison of performance analysis of SOAP-based Web services and RESTful Web services [2], [10]. A strategy for handling concurrent requests from the clients on a mobile device hosting web services was proposed by Gehlen and Pham in [3].

Our paper proposes an approach in improving the concurrency control on mobile web servers by using an asynchronous approach and *Dynamic Resource Management Strategy (DRMS).* The DRMS provides a mechanism to identify the *heavy requests* from the clients requiring significant processing power, memory, concurrent threads, or processing time to generate a response. Examples of such heavy requests are transcoding a large movie, converting a large document into a different format, or editing a large image file. Once the heavy requests are identified by the DRMS, they are assigned to a heap, where they are processed in an asynchronous manner to generate responses for the clients. From our observations we noticed that I-Jetty adopts a synchronous structure where a thread is in a blocked state along with the requesting client, until the mobile web server generates a response. As a result, with more concurrent client requests, the number of available threads in the thread pool decreases, the memory consumption to process the heavy request also increases and subsequently, client requests get discarded. Thus, we implemented an Asynchronous approach, where the number of threads needed by a servlet is dependent only on the time to generate each response. With this approach, we significantly reduce the number of discarded requests. For our research, we developed an Android Web server based on an open source web container, Jetty and its open source mobile container, I-Jetty. Various tests were performed to evaluate the concurrency control on both Dynamo and I-Jetty and the comparison of the results showed significant improvement in the reduction of discarded requests in Dynamo.

The remainder of the paper is organized as follows: Section II presents related work in the area of mobile web servers; Section III explores heavy requests, DRMS architecture, and asynchronous request handling. Section IV provides an explanation of test results and Section V presents a comparison between Dynamo and I-Jetty. Finally, we conclude with future work in Section VI.

II. Related Work

Significant research has been conducted to determine a framework and to develop the architecture for mobile web services. Researchers in the field have worked to define two parameters; designing a framework to develop mobile web services, and b) comparing SOAP and REST technologies in web services.

Hassan, Zhao and Yang in [4] propose a framework for hosting complex web services on mobile devices. They also proposed a partitioning framework that provides a high level design for how to delegate heavy-duty tasks to the backend servers. The concept of a Context Manager responsible for monitoring and computing the resources available on the mobile device is described in [4]. Such a Context Manager can be extremely useful in monitoring the available resources on a mobile device and to determine whether a specific client request is a heavy request or not. Their proposed architecture is based on the concept of distributed systems, where the heavy tasks are distributed to a backend device, whereas for our research, we focus on optimizing the resource handling on a single mobile device.

Mizouni, Serhani, Dssouli, Benharref, and Taleb in [8] propose an architecture that allows deployment of web services to mobile hosts. They tested the performance of web services hosted on mobile devices and identified and evaluated the different factors affecting the QoS of the web services, such as response time, availability, throughput and scalability. They applied their experiments to both SOAP and RESTful web services. From their evaluations they determined that RESTful web services required a lower memory footprint than SOAP based services. The authors however, do not propose any provision to identify or evaluate requests with heavier payloads from the clients.

Gehlen and Pham in [3] introduced a server building block of a mobile web service based middleware. Their server was integrated in the wireless SOAP in terms of a HTTP server binding to SOAP. They provided a strategy for handling concurrent requests where 'n' number of threads are generated to handle 'n' client requests in parallel, and no more threads are created until the n^{th} thread finishes. Their research was only limited to SOAP based services.

Tergujeff, Haajanen, Leppanen, and Toivonen in [14] discussed a proposal to extend the Service-Oriented Architecture (SOA) to mobile devices. They outlined the challenges associated in developing web services on mobile devices and they offered the extension of SOA to lightweight devices as a solution to these challenges. Their research does not provide any method to identify or evaluate requests based on the type of request or the size of the request.

Aijaz, Ali, Chaudhary, and Walke in [2] performed a comparison of REST based mobile web services (MobWS) provisioning with a similar SOAP architecture in terms of HTTP payload. They also presented a resource oriented approach for optimizing HTTP payload. However, they provide no methodology for identifying or handling any requests based on their size. They also do not provide a methodology for identifying or handling heavy client requests.

While research has been done in providing a framework for hosting web services on lightweight devices and comparing RESTful and SOAP services, very few of them [4], [8] address the concurrency issues pertaining to a mobile device. Due to limited processing power, memory, and limited thread pool to run simultaneous requests, concurrency handling remains one of the major challenges in deploying an effective, efficient and fully functional web server on a mobile device. This paper addresses the issues related to handling concurrent incoming client requests, proposes strategies to reduce the number of rejected requests from the clients, and implements the proposed strategies in Dynamo to evaluate and confirm the performance benefits and improvement in concurrency control through the DRMS and Asynchronous approach.

III. DRMS Architecture

DRMS strategy involves taking the heavy requests and adding them to a queue where they will be handled in an asynchronous manner. The DRMS Architecture consists of identifying the heavy requests and assigning them to a heap where they can be processed asynchronously. Once the response is generated, the server sends it to the client. The Figure 1 outlines a summarized DRMS strategy. Heavy requests are the requests from the client that need significant amount of resources on the mobile server to generate a response.

Heavy requests can be identified by calculating the size of the response payload produced by the server with the cost associated with computing needed to generate such a response. Once the heavy request is identified from response payload size and computational cost, the request is then assigned to a heap where it is processed using an asynchronous approach. Non-heavy requests are processed in a synchronous manner. Once the requests are processed, the server then sends the generated response back to the client.

The DRMS architecture consists of six core components included in the server: a) Request Divider, b) Heap, c) Resource Manager, d) Asynchronous Request Processor, e) Synchronous Request Processor and f) Response Generator.

The remainder of this section describes the process of identifying a heavy request, the functionality of the six core components of the DRMS and how asynchronous request handling works.

A. Identifying Heavy Request

As heavy requests consume substantial resources to generate responses, addressing them without a concurrency management strategy adversely affects the performance of the mobile web server. In processing the heavy requests in synchronous manner, the threads used for handling the incoming request are in a blocked state until a corresponding response is generated. This results in a scarcity of available threads in the thread-pool necessary to address other incoming concurrent requests resulting in the discarding of requests from the mobile web server. Thus, evaluating, identifying and handling the heavy requests in an asynchronous manner in a heap is necessary for improving the concurrency control and performance of the mobile web server, thereby, reducing the number of discarded incoming requests.

Two important factors define heavy request: response payload size, R_{pz}, and computational cost, C_c. The product of response payload size and computational cost can determine heavy request.

The response payload size is defined by the size of the response after processing the request payload. A typical heavy request - such as conversion of a video file into a new format - involves a higher payload size than average responses. This requires more processing power and memory resources than mobile devices typically have and constrains the ability of a mobile device to generate a response. Thus, isolating requests that result in high response payload size is justifiable.

Computational cost is also a factor that determines whether a request is heavy or not. Computational cost is the cost to generate the response for a given request. Computational cost is determined by the amount and type of resources consumed by the mobile device to generate a response. Typical resources are hardware-based resources such as memory, processing power, or software-based such as geolocation API or movie streaming API. As described in [4], certain requests such as document conversion or image processing and conversion to adapt to the screen of a mobile device, usually requires more resources than requests to generate recommendations based on user preferences or location. Aijaz, Ali, Chaudhary, and Walke in [1] also describe how requests from REST-based services have less computational cost than SOAP-based services with higher payload.

We determine the average for the response payload size by evaluating a sample size of average-sized requests. The average payload size is given by,

$$AR_{pz} = \frac{\sum_{i=1}^{n} R_{pz}}{n}$$

Once the average payload size is determined, and on receiving a set of N new requests, we normalize the response payload size of the new requests by taking a standard deviation of their response payload sizes. We take standard deviation of the response payload sizes in order to determine the variation in the response payload size of N new requests from the average-sized requests. A set of small or average requests will have a lower standard deviation than the requests required from heavier response payloads. The average response payload size

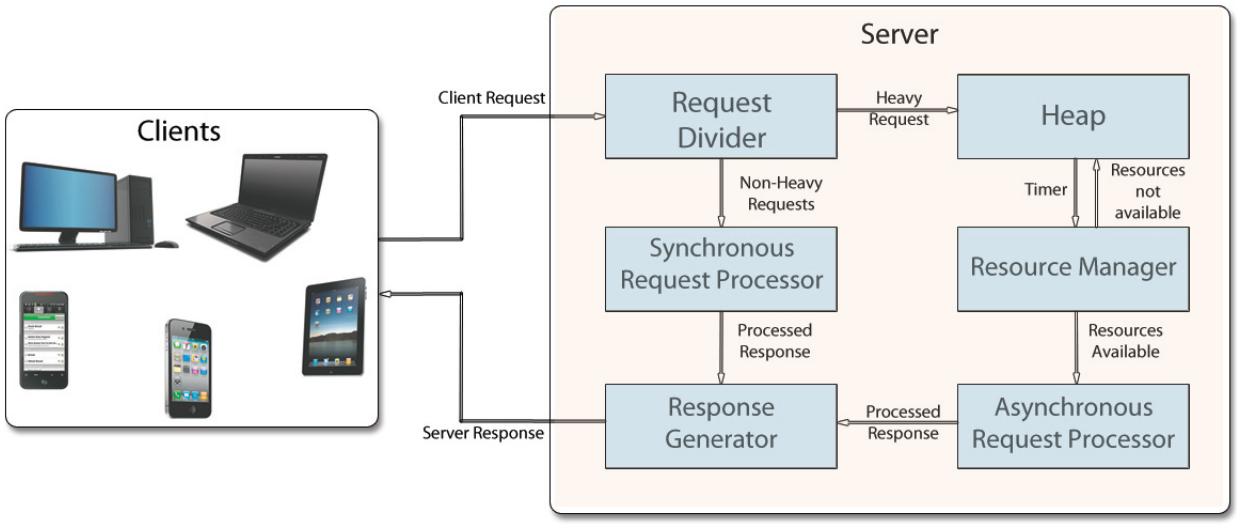

Figure 1. DRMS Architecture

is given by,

$$\sigma_{R_{pz}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(R_{pz_i} - AR_{pz})^2}$$

Similarly, we determine the average for the computational cost by evaluating a sample size of requests with average computational costs. The average computational cost is given by,

$$AC_c = \frac{\sum_{i=1}^{n} C_c}{n}$$

Once the average computational cost is determined, and on receiving a set of N new requests, we normalize the computational cost of the new requests by taking a standard deviation of their computational costs. This allows us to determine how much variation exists in the computational costs associated in processing N new requests and from that of average-sized requests. A set of requests with higher computational cost will have a higher standard deviation, which can help us in identifying heavy requests. The average computational cost is given by,

$$\sigma_{C_c} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(C_{c_i} - AC_c)^2}$$

We then compute heavy requests in standard operating condition where the mobile device is running normal services. Since there is a processing time associated in running services, we include a small Δ t in processing those services. For most cases, (and for our research), we assumed Δ t to be zero. Thus, heavy request is given by,

$$H_{Req} = \sigma_{R_{pz}} * (\sigma_{C_c} + \Delta t)$$

with $\Delta t > 0$, and where Δt = time taken by mobile device to process the requests.

If the heavy request is higher than a certain threshold H_{Req}, we classify that as a heavy request. The threshold may vary depending on certain condition such as device type, version of the operating system, and also session type.

B. DRMS

Our DRMS strategy consists of the following components that are implemented within the mobile web server:

1) *Request Divider:* The Request Divider is responsible for handling incoming requests and identifying the heavy requests from the non-heavy requests. Once the heavy requests are identified, the Request Divider forwards them to the Heap. The Request Divider then sends the remaining non-heavy requests to the Synchronous Request Processor for further handling.
2) *Heap:* The Heap is generally a priority queue where the heavy requests are added. The priority queue follows the First-In-First-Out strategy where a new heavy request is added to the back of the queue, and the heavy request at the front of the queue is scheduled to process first. Each of the heavy requests is assigned a timer, often in milliseconds, so as not to keep a heavy request in the queue for a substantial amount of time. The Heap coordinates with the Resource Manager in determining when to dequeue a heavy request.
3) *Resource Manager:* The Resource Manager identifies whether there are enough resources (memory, threads, and processing power, etc.) available to process the heavy request at the front of the priority queue in the

Heap. If there are enough resources, then the heavy request at the front of the queue is dequeued and sent to the Asynchronous Request Processor for further processing. If however, resources are not available, the heavy request is not dequeued from the Heap. In a worst case scenario, if no resources are available after a certain time threshold, the heavy requests in the Heap need to be discarded.

4) *Asynchronous Request Processor:* The Asynchronous Request Processor is responsible for handling the Heavy Request using an asynchronous approach. By using an asynchronous approach, the threads required to handle the request are only necessary at the time of generating each response. Thus, a thread is no longer in a blocked state while the server is waiting to generate a response for the client. This reduces the required number of threads in processing heavy requests and thus allows the server to handle additional incoming requests without discarding them. Once a request is processed, the Asynchronous Request Processor forwards it to the Response Generator to generate a response for the client.
5) *Synchronous Request Processor:* The Synchronous Request Processor is responsible for handling the non-heavy requests using a synchronous approach. We used synchronous approach in handling non-heavy requests since such requests don't require significant computational resource costs, and time in generating a response. Once a request is processed, it is sent to the Response Generator by the Synchronous Request Processor.
6) *Response Generator:* The Response Generator is responsible for parsing the response and sending it back to the client. If necessary, a response may need to be aggregated from multiple incoming processed requests from Asynchronous Request Processor and Synchronous Request Processor for instance, if the server needs to handle a combination of small and heavy requests to generate a response.

C. Asynchronous Request Handling

Asynchronous servlets are mainly used when waiting for non-IO events and resources. Many of the web-services are required to wait at a certain point, especially during the processing of a request. Examples of such scenarios are:

1) a web service or application waiting for a resource such as database connection or synchronized thread before processing a request
2) a web service or application that needs to wait for a certain event, such as stock price update, or reply of a message in a chat application
3) a web service or application waiting for a response from another remote remote web service

Older versions of the servlet API (versions of, or earlier than, 2.5) only support synchronous call handling. As a result, any wait time required by the web server is accomplished by blocking, whereas, threads responsible for handling the requests are allocated and held during the wait time. Consequently, all the resources including stack memory, application context, and kernel thread are held by these threads during the wait time. This mechanism of holding dedicated threads over a wait time results in waste of valuable resources, and this is especially a concern for mobile devices where the resources are already limited and scarce. To improve the quality of service and achieve better scalability, the waiting of resources needs to be done in an asynchronous manner, where the threads are not blocked during the entire wait time.

To demonstrate the benefits of asynchronous waiting over synchronous waiting, let us consider a web application that needs to remotely access a web service. The web service can be either a SOAP-based or a RESTful service. Under normal conditions, a remote web service usually takes up to few hundred milliseconds to generate a response for the request. An example is eBay's RESTful web service that provides a list of auctions matching a certain query [2]. This web service typically takes up to 350 milliseconds to generate a response, although the time to process the request locally and generate a response is less than 100 milliseconds.

In order to handle 2000 requests per second for such a web service in a synchronous manner, with every request taking 300ms, the web server would need 2000*(300+30)/2000 = 330 threads and close to 165MB of stack memory to process the request. This may lead to thread scarcity and starvation, especially if the server has a limited thread pool, as usually is the case with mobile web servers. Thread starvation and scarcity would lead to performance degradation in the web service, where the server may become slower or even unresponsive. However, handling such requests in an asynchronous manner, the server wouldn't need to block a thread while waiting to generate a response. If an asynchronous call would cost 20ms, then the web server would need 2000*(20+30)/2000 = 50 threads and 25MB of stack memory. From this example, using asynchronous waiting resulted in a reduction of 85% of required resources and 140MB of stack memory. These extra resources could be used to handle additional concurrent requests, which would lead to reduction in discarded requests.

IV. Experimental Study

In this section, we describe our experimental set up, the architecture of Jetty and our experiment that shows how asynchronous waiting approach in Jetty can be used to improve web services.

A. Experimental Set up

For our experiment we use Jetty web server that runs a servlet using eBay API both synchronously and asynchronously. We installed Jetty on a Mac OS X Lion operating system with a 2 GHz Intel Core i7 processor and 8 GB 1333 MHz DDR3 RAM. The web application [3] was set up on Eclipse running Jetty as its web server. The client was a PC

[2]http://docs.codehaus.org/plugins/viewsource/eBay

[3]http://repo1.maven.org/maven2/org/mortbay/jetty/example-async-rest-webapp/7.0.2.v20100331/

running Windows 7 32-bit operating system with 2.9 GHz Intel Core i7 processor and 4 GB RAM. Google Chrome was used as the browser on the client.

The eBay web application was used because it is a very simple API that allows the client to enter one or more product key words to search for and returns those results back to the clients. The ability to search for multiple product key words allows us to test the effect of the size of the client request on Jetty's synchronous and asynchronous handling.

B. Jetty

Jetty is an open source project that provides a HTTP server, a HTTP client and a javax.servlet container, written in Java language. All the components are full-featured, based on standards, use a small foot print, are embeddable, scalable and provide asynchronous operations. Jetty finds its application in many products and projects such as in Google App Engine, GWT framework, and I-Jetty, etc. An overview of Jetty's architecture is shown in figure 2.

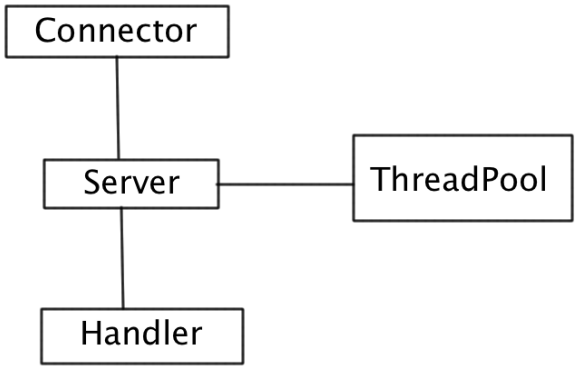

Figure 2. Jetty's Architecture [4]

In Jetty, the server acts as a liaison between the connectors and the handlers using threads from the thread-pool to generate responses. The collection of connectors accept incoming HTTP connection from the clients, parse requests and are responsible in sending responses to the clients. The incoming requests are sent to a collection of handlers for processing and retrieving a response. The handlers use available threads from the thread-pool to process the requests. The handlers may process the request, or forward the request to another handler, or to a servlet for further processing. It may also modify the request before passing it on to another handler. The servlets are mapped to a ServletHandler responsible for generating content. However, a Jetty server can be built by using a collection of connectors and handlers and without using servlets.

C. Experiment

In our experiment, we show the difference between the duration of time a thread is blocked to generate a response using synchronous and asynchronous operations. For our experiment, we use a servlet using Servlet 3.0 API on Jetty that supports asynchronous HTTP clients. Our servlet makes a call to eBay's RESTful web service both synchronously and asynchronously and the eBay's web service responds back with items matching the keywords specified in the servlet. This example demonstrates how the servlet processing is suspended by Jetty using asynchronous waiting approach such that the threads are not blocked during the generation of a response. As a result, fewer threads are used from the server's thread-pool allowing it to handle additional incoming requests from the clients, thereby, reducing the number of discarded requests. This asynchronous technique typically increases the performance by at least ten folds.

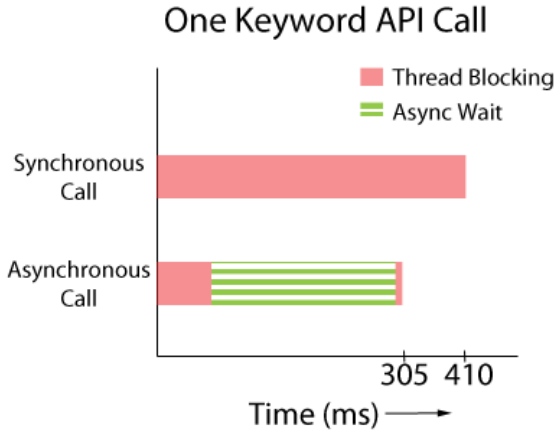

Figure 3. API Call with single keyword.

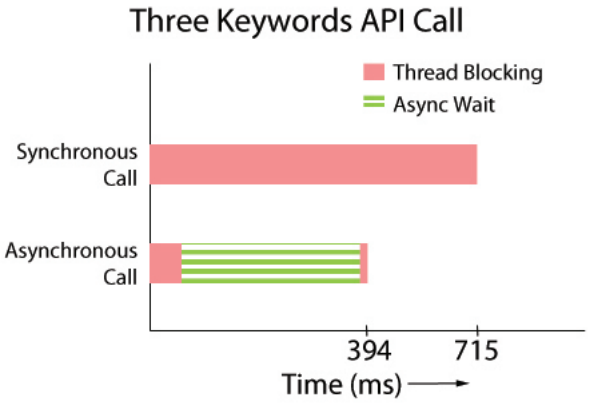

Figure 4. API Call with three keywords.

Figure 3 shows the result of the web service making a call to eBay's web service API using a synchronous and an asynchronous approach. It displays the durations for a thread blocked for a single key word query. The results show that for the synchronous call, a thread is held for 410ms (shown in red color) during the entire wait time to generate a response. If there are 100 threads in a thread-pool, then the maximum amount of requests the server can handle in a synchronous manner will be (1000/410)*100 = 243 requests per second. When the same call is made asynchronously, the thread is held

[4]http://jetty.codehaus.org/jetty/

only for 1.2ms even though the wait time was 305ms (shown in green color). With a thread-pool of 100 threads, using asynchronous calls, the server can handle up to (1000/1.2)*100 = 83,333 requests per second.

Figure 4 shows the results of the API call with a query of three words made by the server synchronously and asynchronously. Here, we observe that for three keywords, additional time is taken to process the request to generate a response. As a result, a thread is held for 715ms, when called synchronously during the time to generate a response. With 100 threads in a thread-pool, the server may only be able to handle up to (1000/715)*100 = 140 requests per second. Alternatively, when called asynchronously, the thread is held for only 1.5ms even though the wait time (in green) was 347.6ms. With 100 threads in a thread-pool, the server will be able to handle up to (1000/348)*100 = 66,667 requests per second.

The stark contrast in the blocking times of the threads and the number of additional requests that can be handled per second show that less incoming client requests will be discarded if the server is using asynchronous mechanism to handle these requests. Thus asynchronous handling of RESTful requests can significantly improve the capacity of a server to process additional requests and to avoid any thread starvation.

V. Test Results & Comparison

For our experiments, Jetty, and its open source Android mobile container, I-Jetty, inspired us. Based on Jetty, we developed our own Android web server, Dynamo (Dynamic Web Server), as our improvement upon I-Jetty, in that it uses asynchronous waiting for handling incoming requests. We implemented the DRMS architecture inside Dynamo, where we handled the calls asynchronously. We tested both of the mobile web servers, I-Jetty and Dynamo, for the number of concurrent requests handled and discarded on a Droid phone running Android 2.1 operating system with 512 MB of RAM and a single core ARM Cortex A8 600 MHz processor. The results from the experiments were then compared and shown in Figure 5A and 5B. The evaluations were done using LoadUI software [5]. Figure 5A shows the our test results with a payload size of 100 KB. When the tests were performed on I-Jetty without DRMS, we observed that with 10,000 incoming client requests per second, the server discarded up to 9000 requests and was only able to respond to 1000 requests. However, when the tests were performed on Dynamo with DRMS architecture, only 4984 requests were discarded out of 10,000 incoming client requests. Thus, with DRMS, for 10,000 incoming requests per second, Dynamo was able to respond to 5 times more incoming client requests.

Figure 5B shows our test results with twice the payload size of 200 KB. With payload size doubled, when the tests were performed on I-Jetty without DRMS, we observed that with 10,000 incoming client requests per second, the server discarded up to 9700 requests and was only able to respond to 300 requests. With Dynamo implementing DRMS architecture, only 5450 requests were discarded out of 10,000 incoming client requests. Thus with DRMS, for 10,000 incoming requests per second, Dynamo was able to respond to 15 times more incoming client requests.

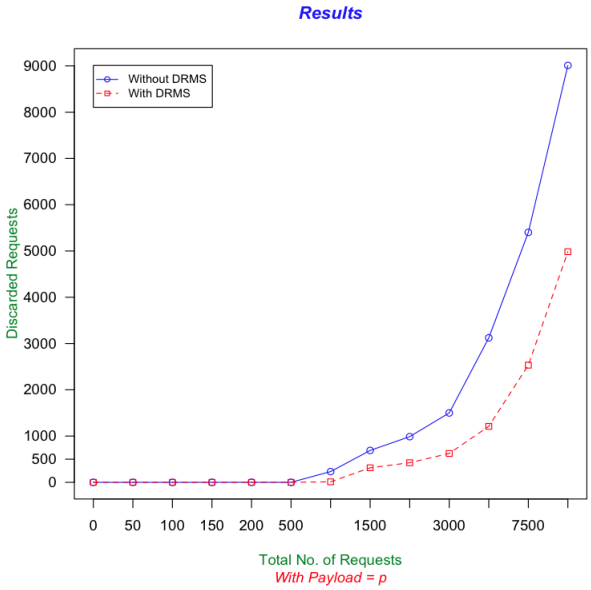

Figure 5A. Results with Payload size p.

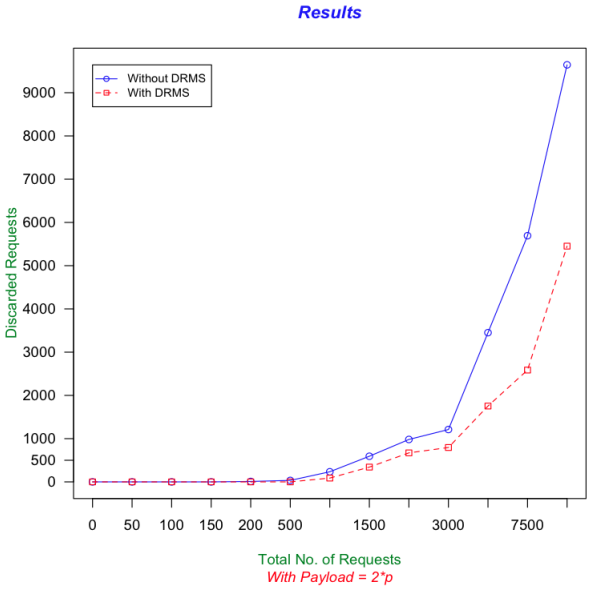

Figure 5B. Results with Payload size 2*p.

From our test results, we were able to obtain significantly better results with DRMS architecture implemented on Dynamo web server. This improvement in results was a result of adding heavy requests into a heap where they were processed in an asynchronous manner. Since the blocking time for each thread on an incoming request is significantly less when the request is processed asynchronously, more threads were available in the thread-pool to handle additional incoming

[5]http://www.loadui.org

requests. Therefore, requests were processed faster on Dynamo than they were when processed synchronous manner on the I-Jetty server.

VI. Conclusion and Future Work

Our objective was to improve the concurrency control so as to minimize the number of discarded requests. To achieve this, we implemented our DRMS strategy where we to took the heavy requests and handled them using an asynchronous approach. We implemented this strategy on our Dynamo Mobile Web Server and from our tests we observed that as the number of requests from the clients increases, the number of discarded requests decreased from at least 90% to at least 50% of the total requests. From our evaluations, we also noted the requests were processed faster in Dynamo than in I-Jetty. This can be explained by the ability of an asynchronous approach to handle the heavy requests allowing the thread pool to be available to handle additional incoming requests rather than being locked on the single current request.

Future work in this topic would include the areas of security, battery consumption, native deployment of Dynamo into other mobile platforms, and addressing portability issues in 3G, LTE and 4G networks. In terms of security, research is needed to improve security on mobile web servers without burdening the overhead of the web services. This may compromise the performance and quality of service. Battery consumption remains a big issue. Research is needed to design smart batteries capable of implementing emerging technologies that are energy aware, efficient and maximize idle power usage. Research can also be done in optimizing the underlying software framework of the mobile web service that can be energy efficient without compromising required performance goals. With distributed systems and cloud computing, it will be possible to develop a framework where the incoming client requests can be divided and distributed over the cloud network using a Map-Reduce system to be dynamically processed in parallel. This would decrease the workload on the lightweight mobile device while improving the quality of services and extend the battery life. Networking and portability issues in 3G, LTE and 4G networks can be resolved by the usage of DynDNS and rendezvous servers, which can work with a mobile server as if it has a static IP, address.

References

[1] F. Aijaz, S. Z. Ali, M. A. Chaudhary, and B. Walke. The resource-oriented mobile web server for long-lived services. In *2010 IEEE 6th International Conference on Wireless and Mobile Computing*, 2010.

[2] Vehicular Technology Conference Fall, editor. *Enabling High Performance Mobile Web Services Provisioning*. Vehicular Technology Conference Fall, September 2009.

[3] G. Gehlen and L. Pham. Realization and performance analysis of a soap server for mobile devices. In *11th European Wireless Conference*, volume 2, pages 791–797, Nicosia, Cyprus, April 2005.

[4] M. Hassan, W. Zhao, and J. Yang. Provisioning web services from resource constrained mobile devices. In *2010 IEEE 3rd International Conference on Cloud Computing*, 2010.

[5] IBM. Ibm study finds consumers prefer a mobile device over the pc, Oct 2008.

[6] M. Lopez. Four ways the post-pc era differs from today, May 2012.

[7] M. Miller. *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online*. Que Publishing Company, 1 edition, 2008.

[8] Rabeb Mizouni, Mohamed Adel Serhani, Rachida Dssouli, Abdelghani Benharref, and Ikbal Taleb. Performance evaluation of mobile web services. In Gianluigi Zavattaro, Ulf Schreier, and Cesare Pautasso, editors, *ECOWS*, pages 184–191. IEEE, 2011.

[9] Earl Oliver. A survey of platforms for mobile networks research. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(4):56–63, February 2009.

[10] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, 2008.

[11] Malladi R. and D. P. Agrawal. Current and future applications of mobile and wireless networks. *Communications of the ACM*, 45(10), 2002.

[12] J.A. Senn. The emergence of m-commerce. *IEEE Computer*, 33(12), 2000.

[13] Satish Srirama, Matthias Jarke, and Wolfgang Prinz. Mobile host: A feasibility analysis of mobile web. In *Service Provisioning, Proc. UMICS 2006, @ CAiSE06*, pages 942–953, 2006.

[14] R. Tergujeff, J. Haajanen, J. Leppanen, and S. Toivonen. Mobile soa: Service orientation on lightweight mobile devices. In IEEE International Conference, editor, *Web Services, ICWS*, pages 1224–1231. IEEE International Conference, 2007.

[15] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17:537–572, May 2008.