Server Architecture - Thread vs. Events

Naman Aggarwal and Prabhuraj Reddy
School of Computing
National University of Singapore
naman@u.nus.edu prabhuraj@u.nus.edu

1. Abstract

More and more companies have started providing their services online which not has not only increased the number of internet users but has also impacted the web logic making it more complex. Simple static HTML pages are replaced by the dynamic pages consisting of many internal web objects which might be downloaded from different servers to the client. This poses the challenge to scale the web server to serve the demands of growing user community and add the complex web logic in the server. For this purpose a lot of web server architecture has been proposed, the leading among them is thread based architecture and event based architecture. In this paper we will be discussing what makes the “well-conditioned server” and how thread and event based approach are working in this direction. We will discuss the pros and cons of both the approach. We will also present a new thread package Capriccio which advocates the use of this package in building web servers which overcomes shortcomings of the threaded approach. We will then discuss a new approach TAME which advocates the use of event based approach for building web servers. SEDA is a design approach which takes both the thread based and event based technique to provide the web services. Last but not least we will be discussing a novel approach to predict the performance of web services. We will also discuss the experiments conducted by different authors and provide our comments and suggestions.

2. Introduction

The internet today is growing at a rapid scale. This has presented a new problem in the area of computer science of scalability. A server on an internet should be able to support millions of users accessing the service providing them response that is robust and always available. A large number of concurrent user sessions leads to higher I/O and network requests and thus putting a lot of pressure on the resources. According to published statistics there are 1.3 billion active monthly users on Facebook sharing 1 million links every 20 minutes [1]. 100 hours of video is uploaded to YouTube every minute and over 6 billion hours of video is watched every month [2].The Slashdot effect has shown that it is normal for a website to have a 100 fold increase in hits when it becomes popular. We understand that today’s hardware is capable of providing such services and handling a large amount of load, it is the software that is becoming the bottleneck. Therefore, it is important for system architects to provide a better design to handle this increased traffic and maintain the robustness of the services.

Nowadays, servers that provide these services tend to be more general. This is generally because of the three factors. First, the services are becoming more and more complex and dynamic in nature leading to more resource utilization. Secondly, the logic of the services provided have to be changed more frequently due to increased demand, this causes complexity in deployment. Finally, due to cost issues these services are deployed on more general platforms rather than the specific engineered systems. To solve the above issues we need to focus on two things - performance and ease of authorship. A lot of systems and architectures have been proposed to solve this issue; however, they seem to be more aligned towards one of the two aspects.

The problems that these servers have to tackle are something that was not anticipated during the beginning of internet. The enormous growth of internet has provided a new set of challenges to the designers of server systems. The first challenge is to support a large number of concurrent connections. A server must be able to accept requests from a large number of users and respond effectively. Secondly, a server must provide robustness and should be able to respond to all the user queries with minimum response time even when the load increases. Finally, it should be able to handle the varying load with graceful degradation wherever possible.

To tackle the load and increased concurrency we usually see two types of server architectures - event based and threaded based. In a thread based architecture, system invokes (spawns) a new thread to handle each incoming request. This architecture is easy to understand, however, spawning of a new thread involves high context switching costs also there is a limit on number of threads a web server can run. This imposes a limit on how much concurrency a threaded model can handle. In the Event based models we generally have only a single thread that loops forever and processes the events from different queues. These events may be network or disk I/O, database operations or application specific events. In this type of system each task is represented as a finite state and state transitions happen due to these events. Event driven systems can handle a large amount of load with little degradation as the load increases. However, event driven makes the application logic complex as the responsibility of deciding when to process the incoming request and in which order to process finite state machines if there are multiple flows. We need to understand the pros and cons of both the approaches before using it for our application. A lot of new papers such as SEDA [3], Capriccio [4], and TAME [5] have offered to either improve one of the two methods or try to find out a middle path. We will examine and compare those solutions.

Our objective will be to -

- To understand the current problem of scalability of internet services and why a scalable server architecture is required.
- To understand the two approaches of designing server architecture - thread based and event based.
- To discuss pros and cons of both approach.
- To discuss how several papers have tried to solve the problem associated with both the approach.
- We will compare the solutions provided in different papers on several parameters.
- Finally we will present our view point on the techniques discussed.

Analysing the performance of the web services has also become crucial nowadays due to the fact that even the slightest delay in milliseconds in providing the web service can have a huge financial impact for the web service providers. We will discuss webprophet [6], technique for automating performance prediction for web services.

3. Well-conditioned service

As defined in SEDA, "a service is well-conditioned if it behaves like a simple pipeline, where the depth of the pipeline is determined by the path through the network and the processing stages within the service itself."

In simple words a well-conditioned service should have these two properties -

- *Throughput saturation* -With the increase in load there should be an increase in throughput which saturate when the pipeline is full. That is throughput should not degrade as the number of clients become very large.
- *Graceful degradation* - As the load exceeds the capacity of the service, it should have a linear response time penalty on all the clients. This is sometimes also known as fairness.

It should be kept in mind that these two requirements are very difficult to achieve in the full sense in practical servers. In fact, as the load increases throughput decreases and the response time also degrades exponentially. This gives a sense that the service is down. Hence, while going through the techniques we hope to find a solution that does better than the current system rather than looking for perfection.

4. Threaded Architecture vs. Event based architecture

In this section we will go in the detail of thread based and event based architecture. We will discuss why it is good to have the particular design, what are the critics behind them and the solutions have been suggested by different papers.

4.1 Threads

In thread based architecture, each incoming connection to the server is associated with a separate thread. It’s doing much better than its predecessor multi-process architecture in which each connection to server is associated with each process and are expensive in terms of process creation time, context switch, memory and limited scalability. Whereas threads are lightweight processes which share its process address space, less memory footprint, faster creation/termination. Due to the overhead of creating a new thread for each incoming requests, usually server use thread pools. This thread pools also called as worker pools consists set of threads which either will be processing the incoming requests or waiting for the new requests. Each request to the server is received by the dispatcher thread, which maps each request to one of the worker thread in the worker pool.

In threaded architecture concurrency in the server is taken care by operating system, so programmers just need to code the behaviour of the server. When multiple threads are involved in the processing of requests, operating system overlaps the threads using pre-emptive scheduling which gives the feel of concurrency. When the thread is involved in the blocking I/O operation, scheduler is called causing context switching to the next thread.

In case of threaded architecture, it is very easy to scale the server using multicore processors, as threads can be scheduled to different cores available. In case of single core, we just get the feel of concurrency, but in multicore we can actually get the performance of concurrency. Java Remote method invocation is an example for the threaded architecture where, for each client request a thread is created.

4.1.1 Why threads are good

The argument favouring the threaded server architecture is based on the observation that concurrency in modern server results from concurrent requests that are largely independent while the code that handles the requests is sequential. So threads are best abstractions for the above server properties. Following are some of the arguments favouring threads:

- Way of programming: The control flow is easier to express in the case of threads. The code written in threaded server is more natural.
- Exception handling and state lifetime: Since the code is less modularized it is easier to handle the exception.
- Easily scalable across cores: Operating system usually schedules different threads on different cores to achieve concurrency in system. So in threaded architecture, multiple cores in the system can be easily utilized by scheduling threads on all the cores without having explicit code to achieve this.

4.1.2 Criticism for threads

Threads has also been criticised on some aspects. Some of them are -

- Race conditions: Usually threaded programs used with pre-emptive scheduling which introduces indeterminacy in context switches and interleaving among the multiple threads, so it involves strong race condition hazard.

- Code complexity: Concurrent events introduce race conditions in case of threaded architecture, so appropriate primitives needs to be used to tackle synchronization issues. This introduces complexity in the code.
- Scheduling: As not all threads have same priority, for thread based systems to ensure that all threads get the fair share of execution, scheduling on the threads need to be controlled to certain extent by the implementers of the threaded systems.

4.1.3 Solving issues with threads

As argued in the paper "Why Events are bad idea" [7], that failure of threaded implementation for high concurrency is not because of the threads itself but due to the poor thread implementations. None of the available thread packages are designed for both high concurrency and blocking operations. In this section we present capriccio, a scalable thread package for high concurrency servers. The main goals of the new thread package is to support the existing thread APIs while scaling to hundreds and thousands of threads, and provides flexibility to address the application specific needs.

While solving the problem of threads in the server applications, user level approach is preferred. This is because user level threads have a clear programming model with better semantics. Also the kernel threads are preferred for concurrency via multiple devices, disk requests or CPUs. In this way we are decoupling the threads in programming model from the kernel which is required as there is a variation in interfaces and semantics among the modern kernels. In this way we are able to integrate compiler support into the thread package. Capriccio thread package address the above mentioned goals with three key features -

1. This improved the scalability of the thread operations using user level threads with cooperative scheduling, and taking advantage of new asynchronous I/O operation.
2. Second feature is linked stacks that solve the problem of stack allocation for large number of threads using the dynamic stack growth. In this paper, capriccio applies compile time and runtime optimization in stack allocation for the threads to limit the wasted stack space in efficient and application specific manner.
3. Capriccio includes resource aware scheduler, which takes information about the flow of control within a program in order to make scheduling decisions based on predicted resource usage.

Let us see how the above three features enables to achieve concurrency using capriccio thread package.

User Level Threads

User level threads create a layer of abstraction between the applications and kernel, which enables to take advantage of new features in the kernel without doing changes to the application code. For example capriccio uses asynchronous I/O mechanism to achieve the scalability of threads. These threads can be tailored based on the application needs unlike kernel threads which are generic for all applications. Capriccio is built on top of Edgar Toernig's co-routine library [8] which provides extremely fast context switch. Capriccio has been tested along with 2 other thread packages i.e., LinuxThreads and NPTL. During the testing it's observed that Capriccio has much faster thread creation and context switching. It also has a faster uncontended mutex locking because no kernel crossing involved.

Linked stack Management

Usually thread packages chose to allocate enough stack space for each thread statically for example LinuxThreads allocate 2MB space for each thread though most threads consume only few kilobytes of memory. This is the main hurdle when we start scaling up the servers. So Capriccio adopted dynamic stack allocation based on the demand. Below are the few features which capriccio adopted to optimize memory usage of stacks.

- The program is represented as the weighted call graph, where each nodes are functions weighted by the maximum amount of stack space that function alone consumes and edges between them indicates that function call the other directly.
- The call graph helps to place a bound on the stack space consumed by the thread.
- To achieve this dynamic stack allocation, they have inserted few checkpoints in the program code which always checks whether there is enough stack space available to reach the next checkpoint without stack overflow.
- Capriccio even provides two tuning parameters i.e. MaxPath, MinChunck which provides applications to have trade-off between the execution time and wasted stack memory based on its requirement.

Resource Aware Scheduling

The important feature that any server architecture should provide is the scheduling its tasks based on the application's needs. Capriccio provides similar application specific scheduling for threaded applications as it uses cooperative threading model in which application is viewed as the sequence of stages. So having the knowledge about each stage and resource used at each stage, enabling dynamic scheduling. Blocking graph is used for making scheduling decisions. In blocking graphs, each node represents location in the program that is blocked and edge exists between the two blocked nodes. Using the blocking graph, they keep track of resource utilization and whether the resource usage is at its level. They hold information about the resources used on its outgoing edges, thus predicting the impact on each resource, whether to schedule threads from that node or not. Based on the above information all nodes are dynamically prioritized. Resources considered for scheduling are CPU, memory and file descriptors.

Based on our discussion of the important features of the capriccio thread package i.e. user level thread implementation, linked stacks, and resource aware scheduling, it looks very viable solution for building the scalable concurrent servers.

4.2 Events

Event driven architecture is suggested by many authors for highly concurrent systems. Conventional systems are built around the concept of call stacks. When the caller calls the function, it waits for the return value from the callee. It continues its execution when it gets the return value and the context is restored [9]. Event driven architecture does not use the concept of call stack. It relies on more expressive styles with basic primitives called events. This obviously decouples the caller from the callee and so the caller should know who will handle the response event. At this point it should be clear that the decoupling of caller from the response makes the system to do other tasks till the asynchronous operation gets complete and a complete event is fired. Events can occur from inside the system or from external stimuli. These events can be consumed by other entities and leads to change in state.

In an event driven approach, the server typically has small number of threads (some system just use one) which loops forever processing the events. As stated earlier these events can either be generated by the OS or the application itself. These events correspond to the network and the disk I/O, timers or are application specific events. There is generally a queue for the events. The processing of each task is represented as a finite state machine in the system. Each event thus leads to a state transition in that Finite state machine. A system thus runs a number of these Finite State Machines with each representing either a single request or a flow of execution in the system.

This type of architecture has been implemented in a lot of systems such as NodeJS [10], Flash [11], and JAWS [12] etc. Taking Flash as an example, the component responds to events such as socket connection or access to file system. There is a main server (dispatcher) which dispatches events to the components (Finite State Machines), implemented as library calls.

4.2.1 The case for events

Following are the arguments in favour of using event based architecture.

- Highly Scalable: In an event based system multiple I/O operations overlap due to the use of non-blocking I/O. This enables parallelism without requiring parallelism at the CPU level. This gives the illusion of multi-threaded concurrency even though only a single thread is deployed. Use of only a single thread for concurrent operations make event based system highly scalable as fewer resources are consumed.
- Robust to load: The event based systems prove to be robust to load with throughput remaining constant for a large range of load. With increase in number of tasks the throughput increases till the pipeline fills. Additional tasks after that are absorbed by the server's queue and throughput remains saturated with little degradation.
- Better visibility of concurrency to application: Events and event handlers make the asynchronous behaviour obvious which is favoured by many developers. This makes the distinction between I/O and CPU operations very clear to the developer.
- Application specific tuning: Since the scheduling of the event is left to the application itself, a lot of fine tuning can be done specific to the needs of application making application more robust.
- Portability: A thread based system requires a thread package either implemented as OS kernel level or at the user level [13]. An event based system requires a system call such as select() for event scheduling. While select() is portable across all the platforms, thread packages are generally not portable.

4.2.2 Criticism for events

A lot of authors criticize the event driven approach due to the following reasons -

- Callback Soup [14]: As discussed before the event driven architecture has a single thread and uses asynchronous non-blocking I/O. This means that whenever a client wants to do an expensive time taking action it should do it in background by registering a callback and continue with the rest of operations. The number of callbacks increases with the number of such operations. So even if it is one logical task, it is separated as number of callbacks depending on number of time consuming operations. This makes the code visually very hard to understand and is known as callback soup.
- Stack Ripping: As now a single task has to be separated into different callbacks, this creates another problem called stack ripping. Since the callbacks are independent of each other so the local variables kept in the call stack are lost on returning. Since all the callbacks are part of a single task we need to preserve the state. Hence, the stacks need to be manually reconstructed in the heap area and has to be passed from callback to callback. Compared to thread programming this task makes programming complex. This also reduces the readability of code and also makes debugging system very difficult.
- Difficult to program: Due to the issues of callback soup and stack ripping the programming becomes very difficult in an event driven architecture. The code is highly fragmented, some of which are not logical. A programmer also has to think about the manual stack management which is an added task.
- Does not take advantage of multi-core processors: In event driven architecture, there is usually a single dispatcher. Hence all the handlers must be run in sequence as they share the common memory. If we try to run them in parallel (in case of multi-core) there is a possibility of memory corruption. This is not an issue with threads, as the threaded architecture has shared memory protection in place even if it is running in a single CPU. Hence, a multi-threaded server can be made to run on multi-core CPU and take its advantage without any changes, whereas, event based system won't be able to take advantage of CPU parallelism.
- Non-ending Listener: This problem is actually due to the manual stack management. Once a handler has registered for events, it will keep on listening till a call is made to unregister.

There is no process of automatic unregistering a handler and this can lead to unnecessary wastage of resources.

- Event-Processing Threads can block: Event driven architecture is based on the assumption that event-handling threads do not block and uses non-blocking I/O mechanisms. This assumption is not completely true. Work done in “Scalable kernel performance for internet servers under realistic loads” [15] has shown that event-processing threads can block due to page faults, garbage collection or interrupts.

4.2.3 Solving issues with events

In this section we will discuss how several researchers have proposed to solve the issues related with event driven approach. Most of the people try to solve the complex programming issues of event and the need for manual stack management. We also see a proposed solution for multi-processer event programming.

Multiprocessor event programming

As discussed earlier event driven architecture cannot take advantage of the multiprocessor system due to shared memory and single thread. There is a solution proposed in “Event-driven Programming for Robust Software” [16] that tries to address this problem.

Their solution is to modify the non-blocking I/O library used by the event driven systems to provide an effective method of running threads on multiple CPUs and avoiding the synchronization issues at the same time. As a part of proof of concept they modified the libasync library. The method needs the programmer to assign each callback a colour. The system then ensures that no two callbacks having the same colour would run in parallel. This allows the programmer to add parallelism to the code and take advantage of multi-core CPUs. It should be noted here that it is programmer’s responsibility to take advantage of the inherent parallelism in this system. This method is also backward compatible as we can just assign a default different colour to each callback.

The concurrency control method provided by this technique also avoids deadlock, since a callback can have only a single colour. Even if we allow multi coloured callbacks we can still prevent deadlocks from happening as the colours are pre declared by the programmer. The proof of concept library presented by the paper has a single queue for all the callbacks. Each kernel thread de-queues a callback that can run within the colour constraints and executes that callback. The experiments done by the authors of the paper shows that event driven parallel system is 1.62, 2.18 and 2.55 times as fast as the uniprocessor system on two, three and four CPUs respectively.

Making sense of events - TAME

“Events can make sense” [5] has proposed a new system to solve the problems discussed earlier that are associated with event based architecture. They call this system TAME. Authors have presented a new programming paradigm/syntax and claim that it is easy to learn and sufficient to build and learn real systems. The objectives listed in the paper solves three main problems of the event based system -

1. Tame provides an event based high level API that solves the problem of stack ripping. This also makes the program easy to understand and debug.
2. It gives an automated memory management scheme that frees the programmer from manual garbage collection.
3. Incorporates both thread and events in the same program taking advantages of both the approaches.

TAME has provided a powerful abstraction that aims to capture the expressivity of the events while maintaining the code readability and modularity. TAME makes the shift very easy as one can just use the ported libraries and source to source translation provided by the system.

TAME Semantics (Flexibility, Readability and Ease of programming)

TAME makes it simple to represent the concurrency problem just like the threads in terms of programming constructs. For the purpose of this TAME introduces abstractions that make it easy to represent the problem. The tame abstractions are as follows -

- event<> : This represents a basic event.
- event<T> : This represents an event with a single trigger value T. This value is set when the event occurs. An event can have multiple trigger values.
- trigger(v): This is a method that triggers the event. Its signature is void and passes zero or more results v to whoever expecting this event.
- mkevent: This allocates an event of type event<T>.
- twait{statements; }: This executes all the statements inside the block and then waits until all the events created by statements using mkevent have triggered.
- rendezvous<I>: To make the wait more flexible a twait can explicitly define a rendezvous object. This make the twait to wait on events associated with rendezvous. Each event associate itself with one rendezvous.

These semantics make it very simple to represent the concurrent problem and make it more readable. Authors claim that TAME version is actually very close to the thread version in terms of lines of code and readability. Making events as first class and distinction between the eventIDs and trigger values, brings flexibility, safety and composability in TAME.

Safe Local Variables (Preventing stack ripping)

TAME has a notion of safe variables which are the variables whose values are preserved across wait points. A programmer can specify the local variables as safe by declaring them under the tvars{} block preserving their value in the heap allocated stack. This helps to solve the problem of stack ripping.

Memory Management and Automatic Stack Management

To solve the problem of responsibility of stack management on programmer, TAME hides most of the event memory management from programmers. For this TAME uses the concept of closures borrowed from functional languages like Haskell and LISP. TAME uses the reference count to de-allocate the closures or rendezvous if they are used explicitly. The reference counting scheme guarantees the following -

- A closure lives until the control exits for the last time.
- A closure lives until the events created in the function have triggered.
- Before rendezvous r is de-allocated the events associated with it must trigger at least once.

To give the above guarantees TAME uses two type of references - strong references which are conventional reference count; and weak references which allow the access to object only till if hasn't been deallocated. The following reference counts kept -

- Strong reference is added to the closure when entering the function first time which is removed when function exits for the last time.
- All the events created holds the strong reference to the corresponding closure.
- Events associated with rendezvous keeps weak references to it and vice versa.

Debugging

As discussed earlier TAME is source to source translation, it also provides the debugging capabilities. Debuggers and compilers point to the line of code in the original TAME input code file in case of some issues and no need to go into the generated source code. TAME allows the programmer to set the breakpoint at a function and then trace the execution unit a block point.

5. Taking a middle path between Threads and Events - SEDA

Though a lot of system has proposed taking a middle path between the threads and events Staged Event-driven architecture i.e. SEDA paper [3] is the most cited. In SEDA, an application consists of network of event driven stages connected via queues. Each stage in SEDA is self-contained application component consisting of event handler, event queue and unbounded thread pool which vary its size based on the dynamic load balancing. At each stage, incoming requests are assigned a thread, which enables to identify the load at specific stage which is very much required for the load balancing. Whereas in case of threaded architecture, for client request, single thread is assigned this makes it difficult to identify internal stage bottleneck. As queues are inserted between each stage, it improves the modularity and lets each stage maintains the load balancing.

The important feature of SEDA is dynamic resource controller which has two components to it. First, control the size of the thread pool of each stage, so as to not allocate many threads to the stage and allocate enough to achieve the concurrency requirement. Second is the batching controller, which controls the number of events processed for each event handler invocation within the stage.

6. Discussion on Experiments

The paper published on SEDA, Capriccio and TAME, conducts experiments to verify the correctness and measure the performance of their system. SEDA, being the earliest of the three papers compared its performance with the traditional thread based server Apache and an event based system FLASH. For experiment purposes, authors developed a High Performance HTTP Server implemented in Java based on SEDA’s architecture called Haboob. Done on 4-way SMP 500 MHz Pentium III machines with 2GB of RAM, Haboob proved to most robust and was able to provide greater throughput under heavy load. However, this throughput was not considerably high. A greater advantage of using SEDA was seen in terms of response time which was around 10 times less than FLASH and 30 times less than Apache under heavy load. Haboob also proved to be better than Apache on the fairness index.

Capriccio authors used the similar server configuration to evaluate Apache server with Capriccio thread package, their test web server Knot using Capriccio and Haboob described in SEDA. Authors claim that by using Capriccio’s thread package the performance of Apache server increased by 15%. Knot webserver was able to match the performance of Haboob. Experiments conducted do not talk about the average response time and fairness. However, authors mention that writing a server with Capriccio thread package is fairly simple and it took them just 3 days to develop Knot.

TAME authors modify the Knot server on its architecture and compare with Capriccio’s Knot server. Their experiments shows that though both TAME and Capriccio achieves same throughput the memory consumption of TAME is very less. Also Capriccio’s knot server needs manual tuning to achieve maximal throughput whereas TAME’s server did that automatically.

7. Predicting performance of web services

In the previous section, we evaluated different server architectures to achieve highly scalable, concurrent web server. Here we will examine the way to predict how the web services are performing which is very critical for the web service providers as introduced in WebProphet [6]. A slightest delay in the search results for search engines like Google in the range of 500 milliseconds can affect their revenue up to 20%. Nowadays web services are much more than static pages, which include HTML pages, CSS, JavaScript, images and more which have dependency among themselves. So it is important to take into account all the dynamic objects which make up the page and dependency among themselves while predicting the performance and webprophet is one of such systems. The metric we use to measure the performance of web services is page load time (PLT).Webprophet includes 3 different components

1. <u>Measurement Engine</u>

Measurement engine consists of set of web agents which allows measuring the PLT of web-page. And it includes a centralized controller which is is required for continual operation of the agents and to upload scripts to control the interaction between the agents and web-page in different conditions. The web agents used need to meet few requirements like should be able to interact with the web-page automatically, it should behave like a full-fledged web browser, and it should provide means to set object and DNS cache and provides means to adjust parallel TCP connections.

2. Dependency extractor

This module extracts the dependency among the different dynamic objects in the webpage. If the object X triggers the download of the object Y, then object X becomes the ancestor of the object Y, and object Y descendant of object X. This relationship between the web objects are captured as the parental dependency graph (PDG). PDG is an acyclic directed graph (DAG) where each node represents the web object and each directed link represents the parental relationship. PDG is extracted by letting the web agent interact with the web-page. As our web agent is equipped with the feature of controlling the download speed of each object, it lets to get the timing information and parental dependence which is required for PDG.

3. Performance predictor

Performance predictor module measures the performance of hypothetical scenario using the PLT of the webpage in baseline scenario. Our target is to predict the new PLT based on the delay factors of any object which is making that page. The idea is to design a model which simulates the page load process under new hypothetical scenario.

Performance predictor module consists of the trace analyser and page simulator. Trace analyser extracts the basic object timing information using the packet traces. Then using the PDG, each object are given more timing information about the client delay. Based on the new scenario objects are adjusted to reflect the timing changes in the new scenario. Now based on the available information, page simulator simulates page load process to predict the new PLT value.

Experiments conducted by authors found the technique to be quite accurate with errors less than 16%. The main advantage of webprophet will be to optimise the real word web applications. One such example has been given in paper when they predicted the performance of Yahoo Maps. They found median PLT of 3.987 seconds with 110 objects. Authors concluded that by moving 5 data objects from Yahoo Servers to Akamai CDN the median PLT can be brought down by 40%.

8. Our viewpoint

This project helped us to understand and compare the various architectures for server design. We also looked into predicting the performance of web services and optimizing them based on the results obtained. We feel that for a well architecture server, throughput should not be the only criteria of measurement, but we should also include other factors such as response time and fairness. A lot of authors in this field compare only the throughput while totally ignoring the response time and fairness provided to clients. Both Capriccio and TAME paper have not included such experiments.

Another thing that we feel should be kept in mind is the ability to write simple code, easy maintenance and understanding. Events based system are always criticised in this respect. Though TAME provide new semantics to overcome such problems, we feel that it is difficult to force people at a larger scale to accept a new paradigm. This might be one reason that TAME semantics have not been implemented baring few practical systems such as in okcupid.com.

The papers also reply compiler support for improved performance. We feel that assumptions are unrealistic as today compilers tend to be general and it is difficult to tweak them to enhance particular software. The capriccio authors accept that they hope techniques such as resource-aware scheduling and linked stacks will be integrated with compiler technology in the future. They also hope that compiler might expose more opportunities for tuning the server for both static and dynamic performance. Also even if such modifications are done, the overhead to provide such properties have to be studied. With this regard TAME proves to be better as it consists of C++ libraries and does not require any platform-specific support or compiler modifications.

By going through the papers we observe that no author has criticised the performance of event based systems. The only criticism of event based systems is that they are not readable and it takes a lot of time and high modularisation to code event based system. Though TAME tries to overcome this by providing new semantics we feel the middle approach taken by SEDA is most appropriate. The paper uses both threads and events to achieve concurrency and flexibility. The best feature of SEDA is the stages that are considered to be the building blocks. We feel that this not only increases the concurrency but also provides better debugging. This approach seems to be novel and we can also prevent the programmers from the complexity of performance tuning. The only disadvantage that we see of this approach is that SEDA's dynamic control is unaware of OS scheduling policy, which should be enough for internet services. We can also look at different servers implementing different stages rather than relying on a single server. This is possible as Moore's law is still valid and can provide better concurrency.

Though internet seems to be open to new technologies, it has been observed that it takes time for the technology to widely adopt throughout the globe. So, whatever the new advancements are done they should be backward compatible and should require less work from programmers and architects. Technologies such as TAME are losing because they call for rewrite of the application which is not possible in majority of the cases.

The webprophet approach for prediction of performance of web services is novel. However, today's web browsers do a lot of optimizations on their end. This includes caching DNS name and web objects. Also many websites today have a dynamic nature that is many objects are requested on the fly using JavaScript rather than specifying in html object. Hence, their approach proves to very efficient for the first request, though, the subsequent requests performance might not be very accurate on modern browsers. Nevertheless, we feel this approach can identify the effect of network layer conditions such as DNS delay or RTT on the web services performance, which is difficult to observe on, built in web developer tools in browsers.

9. Moving forward

By studying the systems we feel that one can relook at the operating systems design to support these approaches. Servers would be able to provide better performance and robustness if operating system exposes some of its properties. The papers Capriccio also calls for the development of better profiling tools to assist the programmer and compiler to tune the system with respect to the application. The event driven systems such as TAME require better debuggers as the errors in such system are very hard to debug. Such systems can also be made widely acceptable if they can provide a better way to translate the source code from traditional system to new system. This suggestion might not be trivial to implement.

Another area that one can look into is to develop systems that are easily tuneable to the application needs. As discussed earlier, today's services are deployed on servers that are generic in nature. Hence, we feel server should provide some tuning abilities so it can be made specific to the application needs. A server should also be capable of tuning its services dynamically based on the load. Though a lot of systems are able to do this efficiently the overhead to keep track of the load is high.

Capriccio paper calls for extending the system to work with multi-CPU machines. This would remove the dependency on co-operative thread scheduling to provide atomicity. One can also look at compilers if they can include performance tuning to support generic systems.

The web hosting companies such as Amazon AWS can look to improve the webprophet techniques, reducing the overhead and making it more real-time. This would enable them to dynamically move objects from one region to another providing better performance. It would also be a learning to look at a recently released event driven HTTP server by Facebook [17]. The company claims that despite being an event driven server, it is very easy to program and provides code reusability, easy integration with current stack and scalability. For scalability it has already been tested on Facebook scale, which is more than enough for most of the web services.

10. Conclusion

We discussed the age old debate of thread versus events in this document and have concluded that a middle path is the way to go. Moreover, if a person still wants to choose one of the two approaches it should be seen which is better for him in the context of its application. Performance should not be considered as the only criteria for judging a system, flexibility, ease of programming and maintainability are equally important factors. We also discussed a method to predict the performance of web pages and improve web services on the results. We feel that our opinions would be useful for server architecture. We hope this literature survey would act as a starting point for someone looking to start research in the area of server architecture.

11. References

[1] http://www.statisticbrain.com/facebook-statistics/ (Accessed on 02-11-2014)

[2] https://www.youtube.com/yt/press/statistics.html (Accessed on 02-11-2014)

[3] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well Conditioned, Scalable Internet Services. Retrieved from http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf

[4] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. Retrieved from http://capriccio.cs.berkeley.edu/pubs/capriccio-sosp-2003.pdf

[5] Maxwell Krohn , Eddie Kohler and M. Frans Kaashoek. Events Can Make Sense. Retrieved from http://pdos.csail.mit.edu/papers/tame-usenix07.pdf

[6] Zhichun Lix, Ming Zhangy, Zhaosheng Zhuz ,Yan Chenx, Albert Greenbergy, Yi-Min Wangy. WebProphet: Automating Performance Prediction for Web Services. Retrieved from http://www.usenix.org/events/nsdi10/tech/full_papers/li.pdf

[7] Rob von Behren, Jeremy Condit and Eric Brewer. Why Events are a bad Idea. Retrieved from http://www.usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren_html/

[8] E. Toernig. Coroutine library source. http://www.goron.de/˜froese/coro/ (Accessed on 02-11-2014)

[9] http://berb.github.io/diploma-thesis/original/055_events.html (Accessed on 04-11-2014)

[10] http://nodejs.org/about/ (Accessed on 05-11-2014)

[11] J. C. Hu, I. Pyarali, and D. C. Schmidt. High performance Web servers on Windows NT: Design and performance. In Proc. USENIX Windows NT Workshop 1997, August 1997.

[12] V. S. Pai, P. Druschel, andW. Zwaenepoel. Flash: An efficient and portable Web server. In Proc. 1999 Annual Usenix Technical Conference, June 1999

[13] Shivakant Mishra and Rongguang Yang. Thread-based vs Event-based Implementation of a Group Communication Service.
Retrievedfrom http://pdf.aminer.org/000/410/044/thread_based_vs_event_based_implementation_of_a_group_communication.pdf

[14] https://thesynchronousblog.wordpress.com/tag/stack-ripping/ (Accessed on 2014-11-01)

[15] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In Proc. 1998 Annual Usenix Technical Conference, New Orleans, LA, June 1998.

[16] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, Robert Morris. Event-driven Programming for Robust Software. Retrieved from http://www.scs.stanford.edu/~dm/home/papers/dabek:event.pdf

[17] https://code.facebook.com/posts/1503205539947302/introducing-proxygen-facebook-s-c-http-framework/ (Accessed on 2014-11-08)