ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Graph Based Processing

Ευάγγελος Φ Καραγεώργος

Επιβλέπουσα **Μέμα Ρουσσοπούλου,** Αναπληρώτρια Καθηγήτρια

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2017

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Graph Based Processing

Ευάγγελος Φ. Καραγεώργος
Α.Μ.: Μ1364

ΕΠΙΒΛΕΠΟΥΣΑ **Μέμα Ρουσσοπούλου,** Αναπληρώτρια Καθηγήτρια

Δεκέμβριος 2017

ΠΕΡΙΛΗΨΗ

Η παρούσα εργασία εξερευνεί την αρχιτεκτονική εξυπηρετητών. Μία ανάλυση διαφορετικών αρχιτεκτονικών σχεδιασμών αποκαλύπτει τους λόγους για τα διαφορετικά χαρακτηριστικά εκτέλεσης που αναδεικνύουν. Προτείνεται μία νεα αρχιτεκτονική, ο γράφος υπολογισμού (process graph), που στοχεύει να αποτελέσει ένα πλαίσιο ανάπτυξης υπηρεσιών και γενικευμένων υπολογισμών. Ο γράφος υπολογισμού, μαζί με πιθανές υλοποιήσεις του, στοχεύει στην αντιμετώπιση προβλημάτων επιδόσεων των υπαρχόντων αρχιτεκτονικών, καθώς και στη διευκόλυνση ανάπτυξης διαχειρίσιμων υπηρεσιών. Μέσω ανάλυσης και επαλήθευσης, υποστηρίζω ότι τα πιθανά πλεονεκτήματα που παρουσιάζονται ισχύουν και ότι ο γράφος υπολογισμού είναι ικανός να είναι ανταγωνιστικός με σύγχρονες αρχιτεκτονικές εξυπηρετητών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Πληροφορική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: εξυπηρετητής, υπηρεσία, αρχιτεκτονική, γράφος, επιδόσεις

ABSTRACT

This thesis explores the software architecture of servers. An analysis of different architectural designs reveals the reasons for the different execution characteristics that they exhibit. A new computation abstraction is proposed, the process graph, that aims to be a framework to develop services and generic computations. The process graph, along with its potential implementations, aims to address performance problems with other architectures, as well as facilitate the easy development of maintainable services. Through analysis and evaluation, I argue that the potential benefits that are presented are valid and the process graph has the potential to be competitive with existing state of the art server architectures.

SUBJECT AREA: Computer science

KEYWORDS: server, service, architecture, graph, performance

ΠΕΡΙΕΧΟΜΕΝΑ / CONTENTS

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ / LIST OF FIGURES

1. INTRODUCTION

The internet age has redefined communication in an unprecedented scale. Globally deployed and massively popular applications and social media have redefined the simple client-server model. Performance requirements for servers that handle all those services pose great challenges to software and hardware engineers. This thesis focuses on server architectures and how detrimental they are to an entire system's performance and behavior.

Servers have been evolving for decades, both at the hardware and software level. Simple HTTP page servers gave way to web application platforms and distributed content delivery networks. Exotic deployment strategies like cloud computing and multi-tiered distributed applications, however, still have a single software component at their core, the server. A server is the software component that receives requests from clients, does some processing and returns appropriate responses. A complex application will typically have numerous servers, like DBMS, network routers, HTTP servers for pages and web services, cloud endpoints or RPCs. The behavior of these applications depends directly on the efficiency and performance of the servers they utilize.

In this thesis, I propose a new computational model and server architecture, the process graph, that attempts to facilitate the easy development of robust and maintainable services, while providing superior performance, tailored to the application. Through analysis, testing and evaluation, I demonstrate the benefits that can be gained and the potential to utilize the process graph approach to develop robust, efficient and flexible server architectures.

2. REQUEST MODEL AND SERVER ARCHITECTURES

2.1 Server characteristics

Servers have certain characteristics that define their behavior under all circumstances.

From the perspective of a client, an efficient server must basically provide low latency and availability. A client only cares that a request is processed as quickly as possible, and that the server has consistent and regular responsiveness.

From the perspective of the server, the system must have high throughput, low latency and robustness under any load. In order to achieve that, the implementation must provide:

- Low latency: Every request must spend minimal time on the server. This translates to minimal time waiting on some queue and minimal actual processing time.
- High resource utilization: The server's available resources, CPU time, memory, I/O etc. must be fully utilized as much as possible while serving requests, in order to achieve maximum throughput.
- Scalability: The server should be able to handle as many simultaneous requests as possible, without causing problems, losing performance or underutilizing the machine's resources.

2.2 Request model

We can analyze different server architectures, based on a simple request model. Every request can be served by the server in specific processing steps that alternate between computation sections and waiting sections. Computation sections are steps that are purely computational and can fully utilize a processor when they run, like a mathematical calculation, or processing data in memory. Waiting sections are steps where the processor blocks waiting for some event to occur, for example waiting for network input or waiting for data from the disk while reading a file.

2.3 Thread-based and Event-based architectures

2.3.1 Thread-based

On thread-based architectures, every request waits until a thread is available and then executes to completion on the specific thread. All processing steps will be executed sequentially, so the thread will alternate between computing and blocking as it executes computation and waiting sections respectfully. At the instances that the thread blocks, other threads can utilize the processor if they are on computation sections themselves. In order to increase processor utilization, we must increase the number of concurrently running threads. However, as the number of threads increases, more processor time will be wasted on context-switching among them, so there is a threshold of the number of threads that, if exceeded, resource utilization and throughput will actually decrease. In order to overcome this, you need to enforce a maximum number of threads, that would depend on the number of available CPUs/cores. This imposes a limit on throughput and scalability, since we can increase concurrency and resource utilization only by increasing the number of threads.

2.3.2 Event-based

On event-based architectures, the requests are executed in a set of event handlers. Every handler is called on a particular event, like data availability on an I/O operation. The server has an event loop and a limited number of threads, typically the number of available CPUs/cores, that execute event handlers for every dispatched event. Instead of modelling the service as sequential code execution, one must develop the service as a set of event handlers for asynchronous I/O operations. The event handlers should never block in order to take advantage of the architecture's performance. Every operation that could potentially block must be implemented with separate event handlers. For instance, a procedure that reads a file must be implemented as triggering a file read operation with asynchronous I/O and registering a separate event handler for the rest of the procedure. Request computation sections are implemented in event handlers, while waiting sections are implemented at event handler boundaries and ideally do not impede the server threads nor waste any CPU time or resources at any scale. These architectures can be highly efficient and scalable, but implementation and debugging of services is much more complex and unintuitive than on thread-based servers.

2.4 Hybrid and alternative solutions

Although, thread-based application design can be equivalent to event-based or message-based design in theory, as proposed by C. Lauer [1], in practice, there are distinct properties of both models that can make a big difference on implementation and performance. Different design principles result to different architectures that can have profound impact on execution characteristics.

According to this analysis, event-based architectures can offer superior performance, but only if implemented correctly. It requires proper asynchronous I/O primitives and careful implementation of the event handlers. Blocking operations must be avoided, which can be difficult, as page faults and cache misses can also produce blocking. Thread-based architectures embrace blocking operations and are far easier to implement both the server, as well as the services themselves.

To try to mitigate both architectures' drawbacks and utilize their benefits, developers have often turned to hybrid solutions as alternatives.

One such solution was the staged event-driven architecture (SEDA) [2] by Matt Welsh et al. SEDA was focused on high scalability and resource utilization for internet-based services. Request processing is broken down to small stages, connected by event queues. Every stage has a queue of incoming events, a user-defined event handler and a thread pool that continually processes those events. Applications are defined as a set of stages that can send events to other stages. This is essentially a pipeline of stages, where every request will pass through, as it is being executed. Every stage has its own thread pool and optional controllers that can regulate the thread pool size, load shedding and other functionalities. SEDA and derivative work has shown great potential in the past and performed better than traditional thread-per-request architectures.

Another similar solution was proposed in 2001 in the paper "Using Cohort Scheduling to Enhance Server Performance" by James Larus et al [3]. The concept is called cohort scheduling, and tries to execute staged requests, utilizing as many threads as available cores. Kernel threads "visit" the stages in succession, executing batches of events on every stage queue. This architecture most closely resembles the process graph, and specifically the AsynchronousProcessGraph implementation, yielding similar benefits and peculiarities.

3. THE PROCESS GRAPH

Given our analysis of server architectures, we want an architecture that has the following characteristics.

- Utilizes multiple CPUs/cores
- Can serve many requests concurrently
- Keeps minimum threads/processes running in parallel
- Minimizes state per request
- Facilitates easy and intuitive development of services
- Is easy to debug and test
- Minimizes processor idle time
- Maximizes resource utilization
- Benefits from, but does not require, asynchronous I/O

In this thesis, I propose a model of computation that attempts to achieve all the stated goals, by introducing the concept of a process graph. The nodes of the graph represent simple processing stages of complex computations. A service or any arbitrary functionality can be represented by a number of these nodes and the specific connectivity among them.

3.1 Definitions

Processing Item

A processing item is an arbitrary piece of data that can be sent to a node as a parameter to trigger execution of that node. The data type of the item is a pointer to an object of any type, and usually encapsulates some kind of execution context.

Processing node

A processing node is an object that can execute a single processing stage. It has a main user-defined procedure that takes a processing item as a parameter. The node consumes a processing item, performs a specific task, and produces zero or more new processing items to be sent to other nodes.

Graph interface

The graph interface is an object that can be used to send a processing item to a specific processing node for execution.

Process graph

The process graph is the set of available process nodes and a graph interface. The connectivity of the graph is not explicitly defined, but it depends on the procedures of the nodes themselves and the execution context. During execution, the host application will produce processing items and use the graph interface to send them to specific processing nodes. These nodes, in turn, will execute their procedure and potentially produce new processing items to be sent to other nodes through the graph interface.

As a new processing item enters the graph, it is executed by a specific node. That node sends the item to another node or produces new processing items and sends them to other nodes and so on. Although, this can theoretically continue cycling through the graph forever, typically, it will stop on some node that doesn't send any items anywhere. We can call this an execution flow and it is usually a single path on the graph. Many services and tasks can easily be modeled and implemented as processing nodes. You can potentially have many different functionalities contained in a single application and a single process graph.

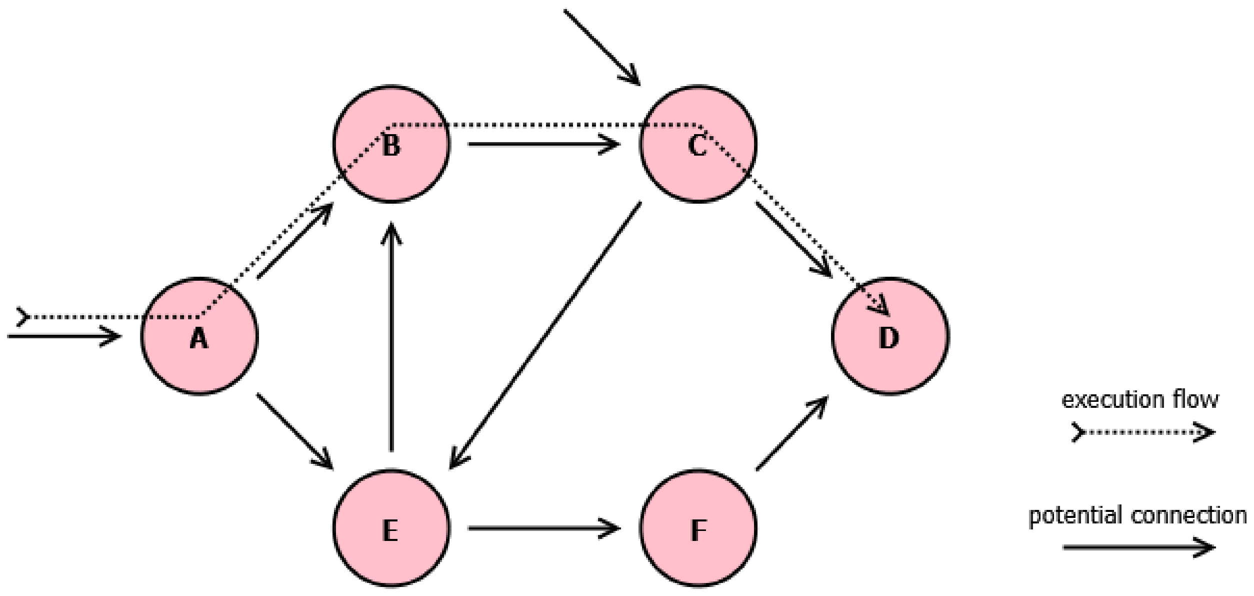


Figure 1: Process graph

3.2 HTTP Server example

As a simple example of a process graph, you can implement a simple HTTP server as the following set of processing nodes. Different nodes can perform specific tasks to process a request and generate a response:

- "read-header": Reads the request header and sends the request to "parse-header"
- "parse-header": Parses the request header and, if it is erroneous, sends the request to "return-error", else, it sends the request to "dispatch-file"
- "dispatch-file": Searches for the requested file in the cache or the file system. If it exists in the cache, it sends the request to "return-file", or else, if the file exists on the disk, it sends the request to "read-file", else, it sends the request to "return-error"
- "return-error": Returns an erroneous response
- "return-file": Returns a response with the specific file from the cache
- "read-file": Reads the specific file, stores it in the cache, and returns a response containing the file

The host application will listen for incoming connections and invoke the process graph to serve them. For every connection, it will create a request processing item and send it through the graph interface to the "read-header" graph node. The resulting execution flow will serve the request and return either an error or the requested file to the client.

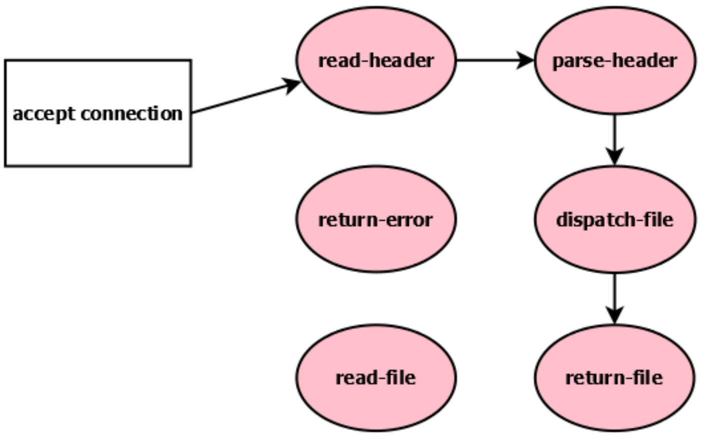


Figure 2: HTTP request - read from cache

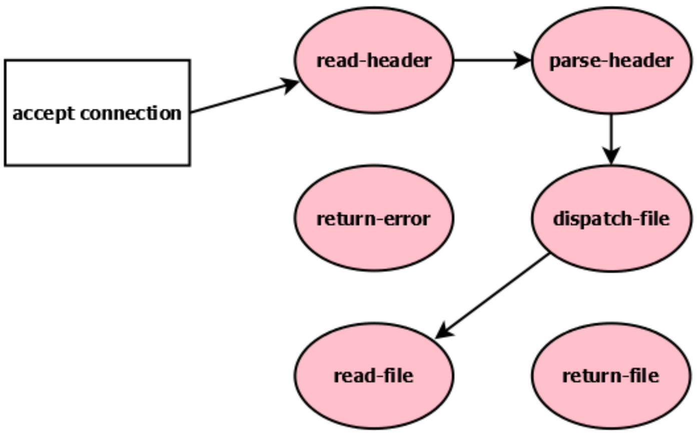


Figure 3: HTTP request - read from file

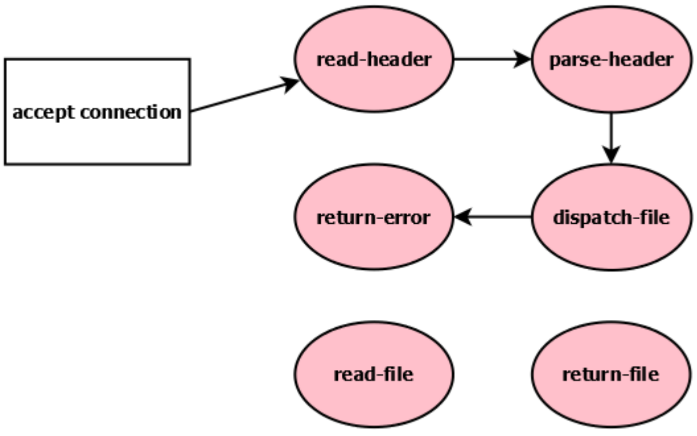


Figure 4: HTTP request - error

3.3 Process graph implementations

Different implementations of the process graph can yield different execution characteristics of latency, scalability, throughput and resource utilization, without changing the code of the graph nodes. For the context of the thesis, I have developed two different implementations, the SimpleProcessGraph and the AsynchronousProcessGraph.

3.3.1 SimpleProcessGraph

The SimpleProcessGraph, as the name suggests, is a basic compliant implementation. It is single-threaded and the graph interface executes each processing node as a simple function call. Consequently, an entire execution flow is executed as nested function calls. This results in low-latency processing and a small memory footprint, but it has very poor throughput and scalability. Also, infinitely long execution flows cannot be allowed, since they will run until the thread stack overflows, resulting in error.

3.3.2 AsynchronousProcessGraph

The AsynchronousProcessGraph is a more complex implementation. Every processing node has a queue of pending processing items. The graph interface sends a processing item to a specific node by inserting it to its queue. Moreover, the graph has a number of "executor" threads that continually iterate through all processing nodes and execute items from their queues. The number of executor threads is configurable, but it should usually be small and not exceed the number of available CPUs/cores. The number of processing items that a thread executes on a single pass through a node depends on the size and growth rate of the specific queue. This results in a tendency to equalize the lengths of the nodes’ queues. The way the executor threads iterate through the processing nodes, as well as the number of items they process for every node, results in specific overall execution behavior, scalability, latency and throughput of the graph itself.

4. POTENTIAL BENEFITS AND IMPROVEMENT

4.1 Elimination of blocking

Using the request model we previously defined, we can implement a service as nodes of a process graph in such a way as to benefit from its architectural characteristics. We can encode the computation sections as processing nodes, and the waiting sections at the boundaries of two consecutive processing nodes, using asynchronous I/O primitives to our advantage. For example, one processing node can perform some computations, trigger an I/O operation and send its processing item to a second node. The second node, in turn, can wait on the I/O operation, perform more computations and trigger another node, and so on. This way, the process graph implementation can execute the first node on an item, execute other irrelevant nodes, and execute the second node for that item later on. In that case, during a portion of the waiting section between the first and second nodes, the system will work on parts of other requests instead on blocking and waiting for the I/O operation to finish. This is similar to how an event-based architecture would behave, eliminating blocking on waiting sections of the request, when it is appropriate, and utilizing the system's resources better. It would be, however, simpler to implement, since you don't have to use very specialized operations, events and handlers. All you have to do is break a serial procedure into smaller parts and utilize simple async-wait operations on the node boundaries.

4.2 Flexibility of behavior

Another potential advantage of the process graph approach is in terms of flexibility of responsiveness. Most service systems operate on a typical first-come first-served basis. The server can be processing a maximum number of requests simultaneously, and every single request will take a certain time complete, once it starts being processed. If the server is saturated, new requests will be queued, waiting for their turn to be processed. This behavior can be modeled as a simple queue system. Requests are incoming at a certain rate, they are being processed, and the response is forwarded back to the client. Under queuing theory, you can assume an incoming request rate of Poisson distribution with parameter λ and a service time of exponential distribution with parameter μ. Such systems have been thoroughly studied for communication systems, especially packet-switching networks.

We can define:

- λ: request arrivals per time unit
- μ: request service rate
- Tr: execution time for a request
- T: total time a request is on the server
- ρ: Utilization coefficient, $\rho = \lambda/\mu$
- θ: request rate stability threshold

Every server architecture can be modeled as a queue system, where given certain computational resources and an average time μ it takes to process each request, there is a critical threshold θ of requests per time unit that determines the time it takes to service each request.

- If the request rate λ is less than θ, every request will be serviced in Tr, and there will be no congestion on the server.
- If the request rate λ is bigger than θ, the system is out of balance, and every consecutive request will take longer time to be serviced, until the server is essentially unresponsive to new requests.
- If the request rate λ is exactly θ, requests will be serviced by a constant rate that is determined by the constant congestion on the server.

Usually, the server's request queue will be capped at a certain maximum, so that the response time does not exceed a maximum value, and, if the queue is full, additional requests will be rejected. The server will be saturated if it is under a constant request rate λ > θ, and there is little you can do to mitigate that behavior. However, in most situations, the request rate will not be constant. Most of the time it will be significantly smaller than θ, and occasionally there will be bursts of requests that might exceed θ for a limited time frame. The way the server handles these bursts can make a significant difference to the overall responsiveness.

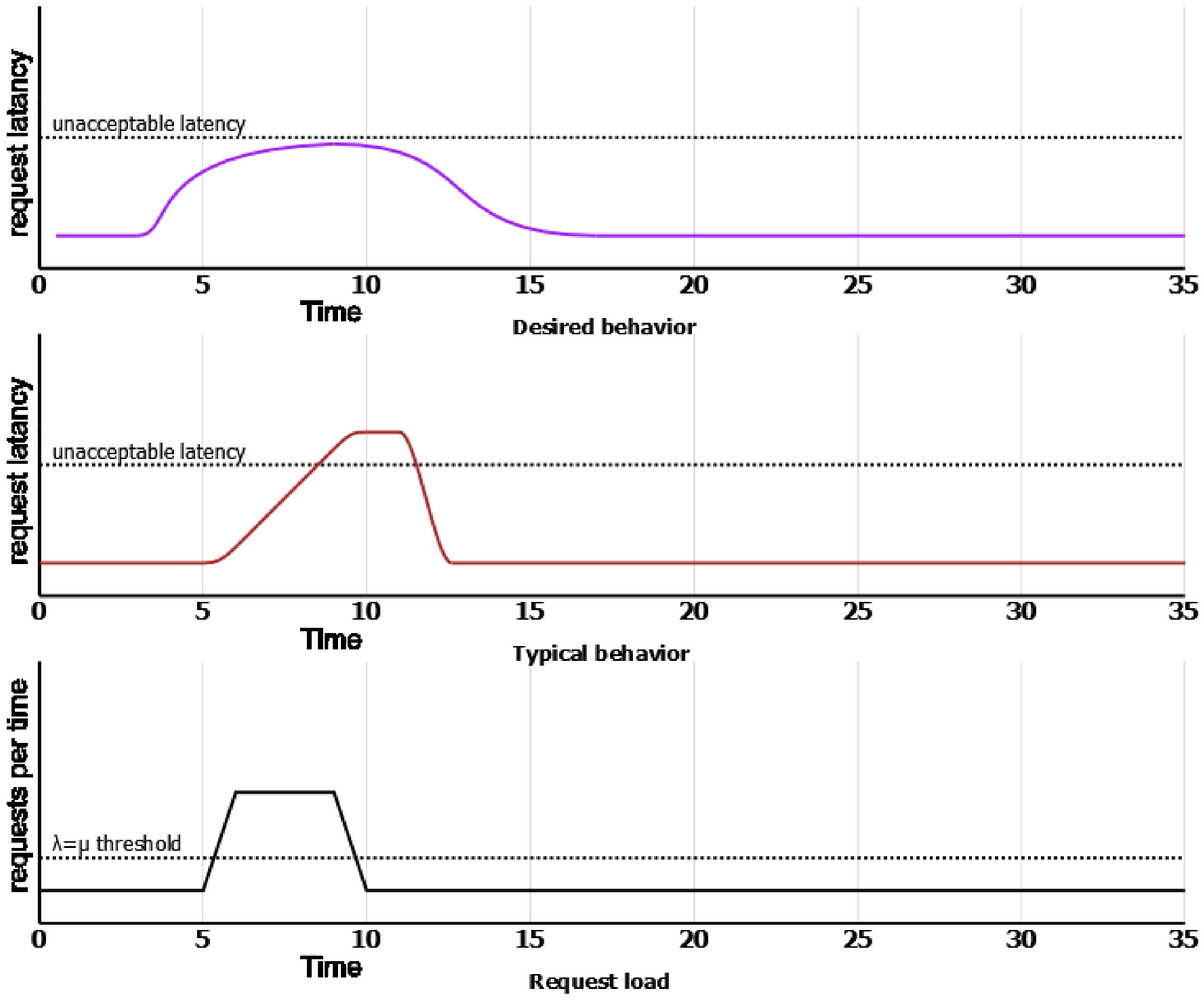


Figure 5: Request latency behavior under burst

In a conventional server architecture, when a request burst is encountered, the request queue as well as the response time for each request will grow constantly until the burst is over. Requests before and at the start of the burst will be serviced instantly, with a response time close to Tr, while requests during the burst will have a linearly growing response time. When the burst ends, subsequent requests will still have large response times, until the queue starts clearing. This behavior can be illustrated in Figure 5 as the typical behavior.

We can observe that the response time grows linearly, until the burst ends. This means that some requests will take an unreasonable amount of time to complete. The clients that issued those requests may timeout, and would ultimately be dissatisfied with the server's behavior. Even though it is expected that under heavy load any server would decrease their responsiveness, it seems unfair that requests near the start of the burst are not affected at all, while requests near the end are heavily degraded. A more desirable behavior would be such, so that all requests near the burst be affected equally, as shown in Figure 5 as the desired behavior.

This way, we would contain the spike of unreasonable response times, by distributing the latency around a larger time-frame. One observation anyone can make is that in order to achieve that, you must make the requests before the burst take longer to complete and divert computational resources towards newer requests from the burst. In a system where a request waits on a queue, and then executes uninterrupted to completion, this is not possible. You can't predict that a burst is about to occur to delay execution, and you can't interrupt the execution of a request, once is has already started. In order to solve this problem, you generally have two options. One option, in a thread-based system, is to forcibly slow down the execution of the thread that is processing those requests, by customizing its priority policy. This is, however not a viable option, since you rarely have such control over the threads. The threads are in most cases operating system constructs and use preemption with priority policies set by the kernel. Also, even if you have access to such flexible threading API, fine-grade control over the thread to manipulate specific requests and parts of requests can be very difficult. Another option is to break-down the request processing in multiple steps and control when those steps will run on a thread. Modeling a service as process graph nodes gives you that exact option. Since the server handles processing stages of requests separately, it can manage the execution strategy to implement the desired server behavior.

The AsynchronousProcessGraph implementation tries to exhibit this behavior. In this implementation, every node has a queue of pending processing items, and a number of threads that iterate through the nodes and execute a specific number of items every time. By controlling the number of processing items that are being executed on every node, the implementation tries to manipulate their queue lengths in a specific manner as to artificially delay earlier requests and divert computational resources more uniformly across a wide time-frame of a burst. The end effect of this strategy is as following:

- Requests that arrive before a burst take longer to complete
- Requests that arrive after the burst also take longer to complete
- Requests that arrive during the burst take even longer to complete, but their latency is kept relatively constrained

Although the burst would be an abrupt and heavy spike in load, the server responsiveness has a long, gradual and benign bump in latency. From the side of a client, during the spike, the server increases in latency for a while, but doesn't get to unreasonable response times.

An additional effect is that even if a request takes a lot longer to complete under heavy load or a burst, its execution is different from a simple queued server. Typically, under load, a request will wait on a queue for a long time, before abruptly executing to completion. The client has no information about the execution status until the request is serviced. Using our approach, the request will start executing its stages sooner. The entire request will take some time to finish, but its stages will execute periodically, with small delays between them. This is a sign of progression, and the client can observe it and conclude that the request is actually being serviced, as well as estimate how long it might take to complete. This can be demonstrated for an HTTP request. There are several points of feedback during an HTTP request after the initial step of the client initiating the TCP connection.

- The server will accept the TCP connection
- The server will read the request header
- The server will read the request data
- The server will send the response header
- The server will send the response body
- The server will finish sending and terminate the connection

Every step of the way, the client will know about the status of the HTTP request, because all these steps have feedback. On a simple thread-based HTTP server under heavy load, the client will wait indefinitely for the TCP connection initiation, while the request is still in the server's connection queue. Then, the request will execute abruptly, but if the waiting time is excessive, the client will just assume that there was a network connection problem and abort. If the server is implemented as a process graph, however, the client will receive signs of progress early on and at every execution stage. This is really desirable, since the client will know that their request is being properly processed, so it wouldn't abort, and can predict when the request will be complete.

5. COMPARISON WITH RELATED WORK

5.1 SEDA

A similar approach to the process graph was the staged event-driven architecture (SEDA) [2]. SEDA is an architecture that breaks a procedure down to a set of stages. Every stage is executed independently, by different threads, and a request is processed stage-by-stage until completion, or until a stage decides to drop the request, based on load shedding policies. SEDA was popular because it was an architecture that combined some of the advantages of event-based concurrency with a framework for adjusting and controlling resource management and adaptive load shedding. The result was high scalability, robustness, superior performance, better resource utilization and ease of development of services compared to event-based architectures. Although the process graph approach is similar to SEDA in many aspects, there are points of differentiation that I believe can give an edge to the process graph.

5.1.1 Ease of development

Although SEDA was designed to be simple for the developer to construct services, development on it is far from trivial. SEDA is a framework with a somewhat complex API. The developer is responsible not only for the service stages and their API-compliant implementation. In order to take advantage of SEDA's potential performance the developer must consider advanced features of SEDA, like load shedding, precise configuration, thread and queue bounds and asynchronous I/O controllers. The process graph, on the other hand, has a very simple API, and offers decent performance without very specialized involvement with the graph's internal workings.

5.1.2 Performance and Flexibility

SEDA has a precisely defined architecture. On SEDA, every stage is executed by dedicated threads in the form of a deep pipeline. This is advantageous in terms of throughput and scalability, if implemented carefully, but has a severe impact on latency. A request passes through thread barriers on every stage that is being executed. This can result in exacerbated latency, due to the sequential involvement of multiple threads for the same request. An appropriate process graph implementation, like the aforementioned SimpleProcessGraph, can make sure that a request executes without interruption on the same thread, minimizing latency. Even on AsynchronousProcessGraph, there is a high chance that a request will execute on a single thread, and there is a very small chance every stage will be executed on different threads, even under high load.

Another problem that can arise on SEDA due to involving multiple threads for a request is poor performance due to diminished cache locality. Although SEDA tries to mitigate that by having a pool of shared buffers for specific operations, like file or network data transfers, any other request-specific data will be accessed by multiple threads during execution. This can be even worse if the developer doesn't use the specific I/O stages that SEDA provides that use these shared buffers, using instead external libraries or naive implementations.

The pipeline approach, also, has the side effect that there is a minimum required number of threads, at least one per stage, not to mention additional threads for controllers and asynchronous I/O emulation. If the underlying hardware has too few CPUs/cores, and the service in question has many stages, it would result in a suboptimally large number of concurrently running threads. The AsynchronousProcessGraph implementation, in contrast, can use the optimal number threads for the hardware it is running on, independent of the number of nodes.

The architectural decisions that were made by SEDA's authors made sense because they targeted a very specific application domain, OS properties and hardware. Internet services typically involve superior hardware serving a huge number of clients, with significantly dynamic load. High throughput, scalability and good resource utilization are far more important than low latency and ease of development in this context. The difference with the process graph is that it is not restricted to a specific application domain. In fact, since the process graph is a more abstract architecture, it is far more generic. There can be many different implementations that can focus on low latency, throughput, scalability, better hardware utilization or any other factors. Depending on the application and deployment in question, the developer can choose an appropriate implementation, or even develop one themselves, to better fit the requirements. Additionally, many implementations can coexist, and be switched among themselves, adapting to current runtime conditions.

5.2 Cohort scheduling

Cohort scheduling [3] is a staged architecture whose execution model is very similar to the AsynchronousProcessGraph implementation. Services are implemented as sequential stages. A number of threads, typically one per CPU/core, will traverse the stages in a specific pattern, executing events from the stages' queues along the way. The number of events that a thread will execute at one stage is dynamic, and depends on the load on the server, the queue length and the configuration. Data locality, low synchronization overhead and low number of expensive context switches are benefits inherent to both cohort scheduling and the AsynchronousProcessGraph implementation. The points of differentiation between cohort scheduling, as described, and the process graph are as follows.

5.2.1 Ease of development

Again, the process graph API is extremely simple. Its purpose was to decouple server architecture intrinsics from service development, and the resulting API is small, simple and intuitive. Cohort scheduling implementations still require event semantics and a complex API to define the services.

5.2.2 Flexibility

We can compare the cohort scheduling architecture with the specific AsynchronousProcessGraph implementation in terms of performance. Although both their intrinsic properties can yield similar performance, the process graph is much more generic. Different implementations can focus on throughput, resource utilization, low latency or other properties. You can customize the implementation and deployment parameters to fit an application's specific needs, without developing different service process nodes. The cohort scheduling architecture defines a list of sequential processing stages. The process graph, on the other hand, is a processing abstraction, and can support complex graph configurations, like loops, recursion and an overall dynamic execution path.

Overall, Cohort scheduling defines both a server architecture and a scheduling policy. This approach is tailored to specific application domains, assumptions about the underlying hardware and specific families of request metrics and distributions. The process graph, being a generic abstraction tethered to a simple API, can be customized significantly for a wide set of applications, constraints and deployment environments.

5.3 Enhanced threads

There has been a lot of work on improving threads to be able to use a pure thread-based model without significant scaling problems. Modern system-provided threads are far superior to threading decades ago. This has led to a trend of returning to traditional server architectures. SEDA author Matt Welsh addressed this as early as 2010 (http://matt-welsh.blogspot.gr/2010/07/retrospective-on-seda.html), on which he acknowledged shortcomings of SEDA and how the server landscape had changed. Moreover, specialized threading packages like Capriccio [4] address certain shortcomings of kernel-residing threads by providing lightweight, user-space, fast and resource aware threads. These solutions are designed to make the traditional thread-per-request model feasible for high concurrency by significantly empowering threads. This could shorten the gap between event-based and thread-based concurrency, making elaborate approaches like the process graph less necessary.

However, there is still significant value to these approaches. Any architecture that uses a limited number of threads still benefits from improved threads anyway. Faster thread context-switching, low memory footprint, less kernel crossings, faster synchronization and lower cache thrashing are bound to improve event-based or staged architectures as well. Better threading improves both thread-based architectures and process graph implementations. Of course, thread-based architectures will get the lion's share of performance improvement, but the additional benefits of the process graph still stand. The benefits of increased flexibility, precise request control and resource utilization that the process graph can offer would still give it an edge. In order to implement policies as described in section 4.2, it is necessary to break request processing down into sections that can be managed individually. Pure threading solutions cannot do that, since requests can only be managed as a whole. The only entities that can be managed are requests, queues of requests and the threads themselves. On a process graph, you can manage the threads, processing items (which are effectively request stages), and processing nodes and their queues if applicable.

6. JSERVICE PROTOCOL

In order to develop and evaluate the server architectures that are being presented, I have designed a service framework and API that can be utilized for arbitrary point-to-point communication. The original idea was to implement the HTTP protocol, as it is a common communication protocol. I decided to design a new protocol in order to highlight the characteristics of the process graph. The protocol is designed around a form of structured, serializable JSON-based messages.

Message

A JService Message has three main sections.

- Length: a hex-encoded 32-bit unsigned integer that encodes the header length in bytes
- Header: a JSON object that contains the structured message parameters
- Data: an optional section that contains unstructured binary data

The header has certain protocol-defined fields along with any user-defined ones.

- "parameters": an object with user-defined structured parameters
- "content": an optional header field that specifies the form of the data section, like the encoding method, data length etc.

A message can convey any kind of information, from a simple signaling message to large binary blobs of data. In order to facilitate the server-client model I have developed two specializations of the general message.

Request

A JService Request is a message with additionally defined header fields that represents a client request to the server.

- "type": a mandatory string that defines a request type
- "source": an optional string that defines the request source as a network hostname
- "destination": an optional string that defines the request destination as a network hostname
- "persist": an optional boolean that signals the server to keep the underlying connection open after the response has been sent to the client
- "cookies": an optional array of objects that play a similar role as HTTP cookies

Response

A JService Response is a message with additionally defined header fields that represents a response from the server to the client.

- "success": a mandatory boolean that signifies a successful request
- "error": an optional object with information about the error, if the request was unsuccessful
- "cookies": an optional array of objects, similar to Request cookies

The design of the JService message is such that facilitates sequential processing. A process can read the header length, parse the header and process the data. Even though one can read the whole message, write it in a buffer and process it later on, the message can be processed as it is being received, without any intermediary buffer or memory copy overhead, especially for trivial processing. The utilization of JSON as the parsable section can be fast and efficient, as the standard is ubiquitous and easy to implement efficiently.

7. APPLICATION

The implementation part of the thesis includes two separate applications, implemented in C++, that utilize opLib, an extensive library that I have developed as an application framework. The JService protocols, the process graph, the server framework and the windowing application framework are implemented as part of the library, and were utilized to build the testing applications.

7.1 JServiceServer

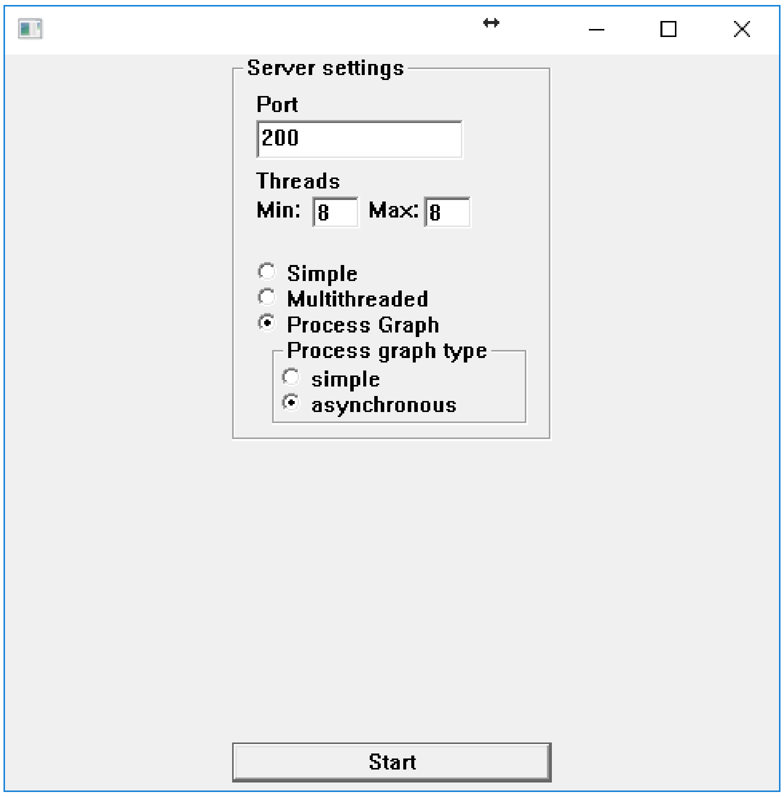
JServiceServer is the application that implements the different server architectures to compare their execution characteristics. During execution, the user can choose among different server implementations and set appropriate parameters to alter the server's behavior and performance. You can choose among three different implementations

- Simple: A simple, single-threaded implementation. The server is on a loop waiting for incoming connections. When a new connection is established, the request is being parsed, processed and the appropriate response is returned to the client.
- Multithreaded: A thread-based server implementation. A thread waits for incoming connections and puts them in a queue. A set of executor threads continually check the queue for available connections. For every connection, an executor thread removes the connection from the queue, processes the request and generates the appropriate response. This is not a traditional thread-per-request server, since the literature suggests that such architectures are inefficient. This architecture is essentially a bounded thread pool that executes requests as an M/M/m queue system.
- Process Graph: A server implementation family that utilize a process graph. The Process Graph choice prompts the user to choose a specific process graph implementation. The choice "simple" designates a SimpleProcessGraph implementation, as described earlier. The choice "asynchronous", in turn, designates the complex AsynchronousProcessGraph implementation.

You also choose the service network port, and the minimum and maximum number of threads. When the server is running, the application displays a rolling window graph of requests per second that are being processed.

The application supports two services that were implemented to demonstrate the server’s behavior on different request characteristics.

- “null”: a trivial echo service that does essentially nothing and returns a simple response to the client
- “blur”: A computationally intensive service. The service interprets the request data as an image, performs multiple steps of Gaussian blur on that image, and returns it back to the client.



Server settings

Port

200

Threads

Min: 8 Max: 8

Simple
Multithreaded
[✓ Process Graph]

Process graph type
simple
[✓ asynchronous]

Figure 6: JServiceServer idle

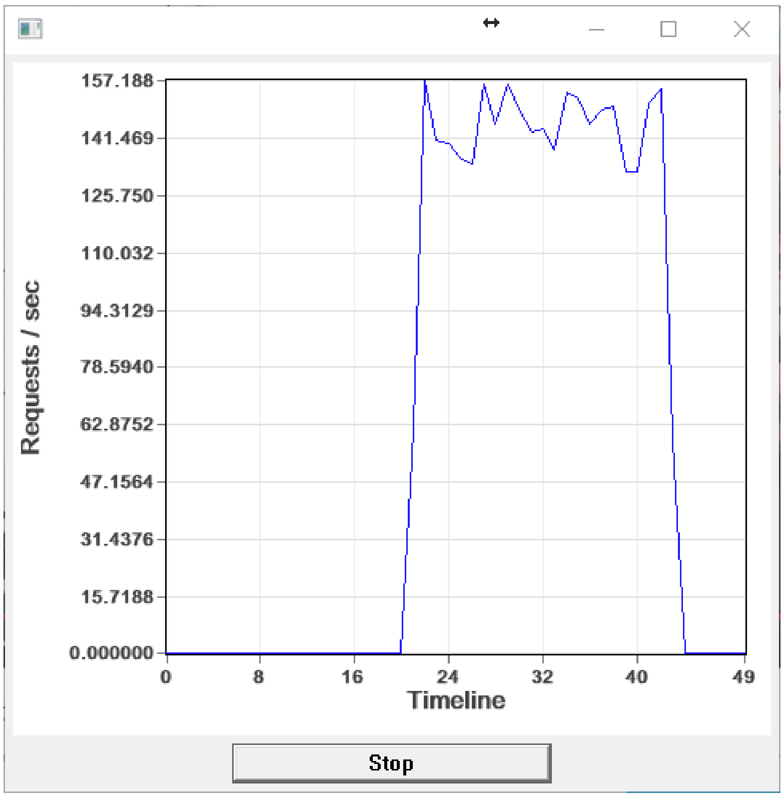


Figure 7: JServiceServer running

7.2 JServiceClient

JServiceClient is the application that tests the server for correct behavior and performance. The application enables the user to set the network connection parameters and perform certain tests on the server. Along with a small section to define the network parameters, the main window has three tabs that correspond to different tests.

7.2.1 Convolution

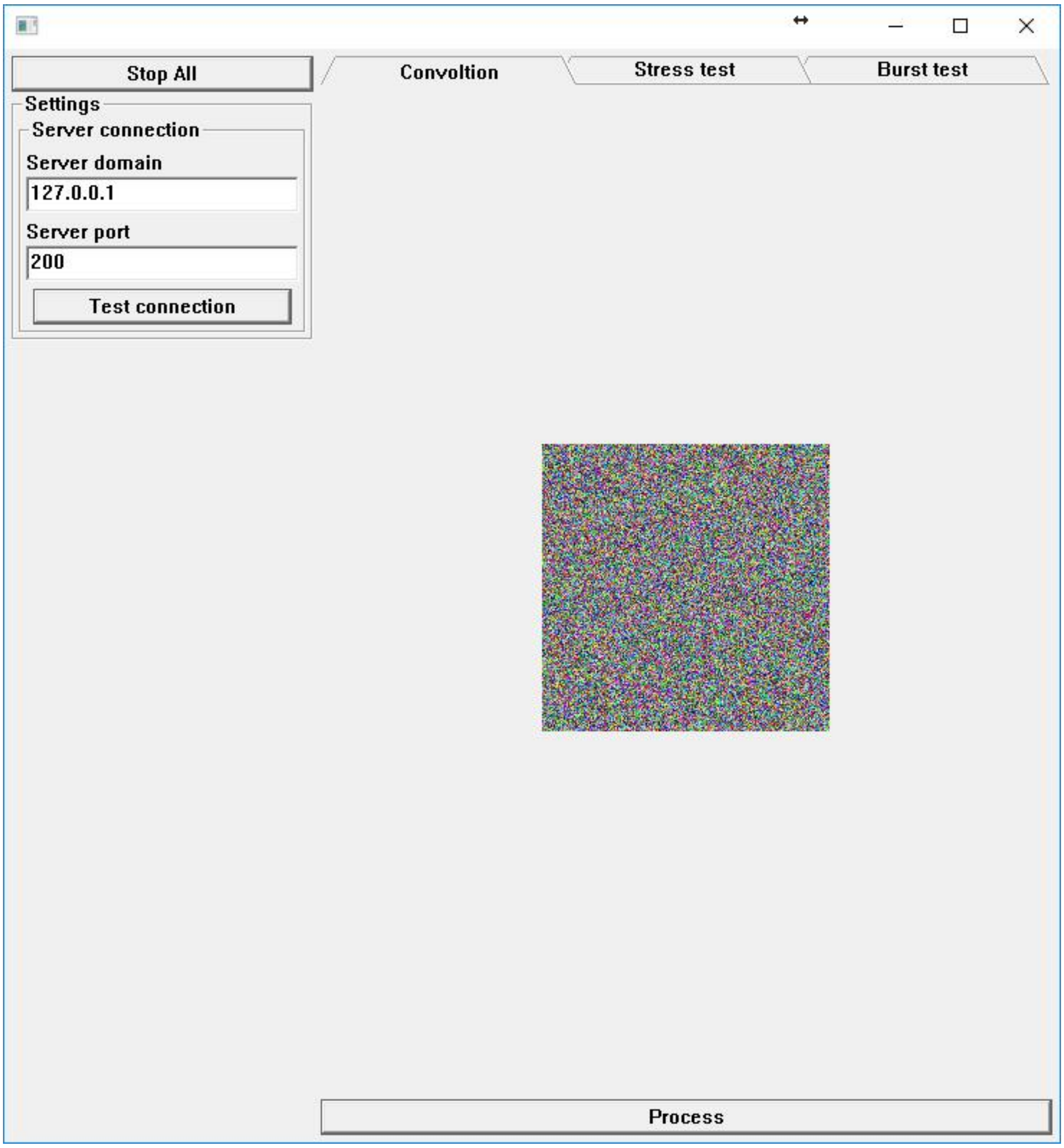
This tab has a white noise image at the center and a button with the caption “process” on the bottom. When the user clicks the button, a request is generated for the “blur” service, with the image as a parameter. The request is being sent to the server, and when the response arrives, the image at the center is replaced by the result. The test is simple and has the sole purpose of validating the correct behavior of the server.

7.2.2 Stress test

This tab is for testing the server’s behavior under a high, constant load. You choose the services being tested, the test iterations for every thread and the number of threads. Every thread runs a loop where it generates requests for the selected services, queries the server for its responses and waits for the specified interval. Having many threads constantly issuing requests will put the server under a constant load. The testing will start once you click the “start” button and will finish when every thread executes the number of iterations or when you click “stop”. After the run, a graph will be generated showing the response time for every request.

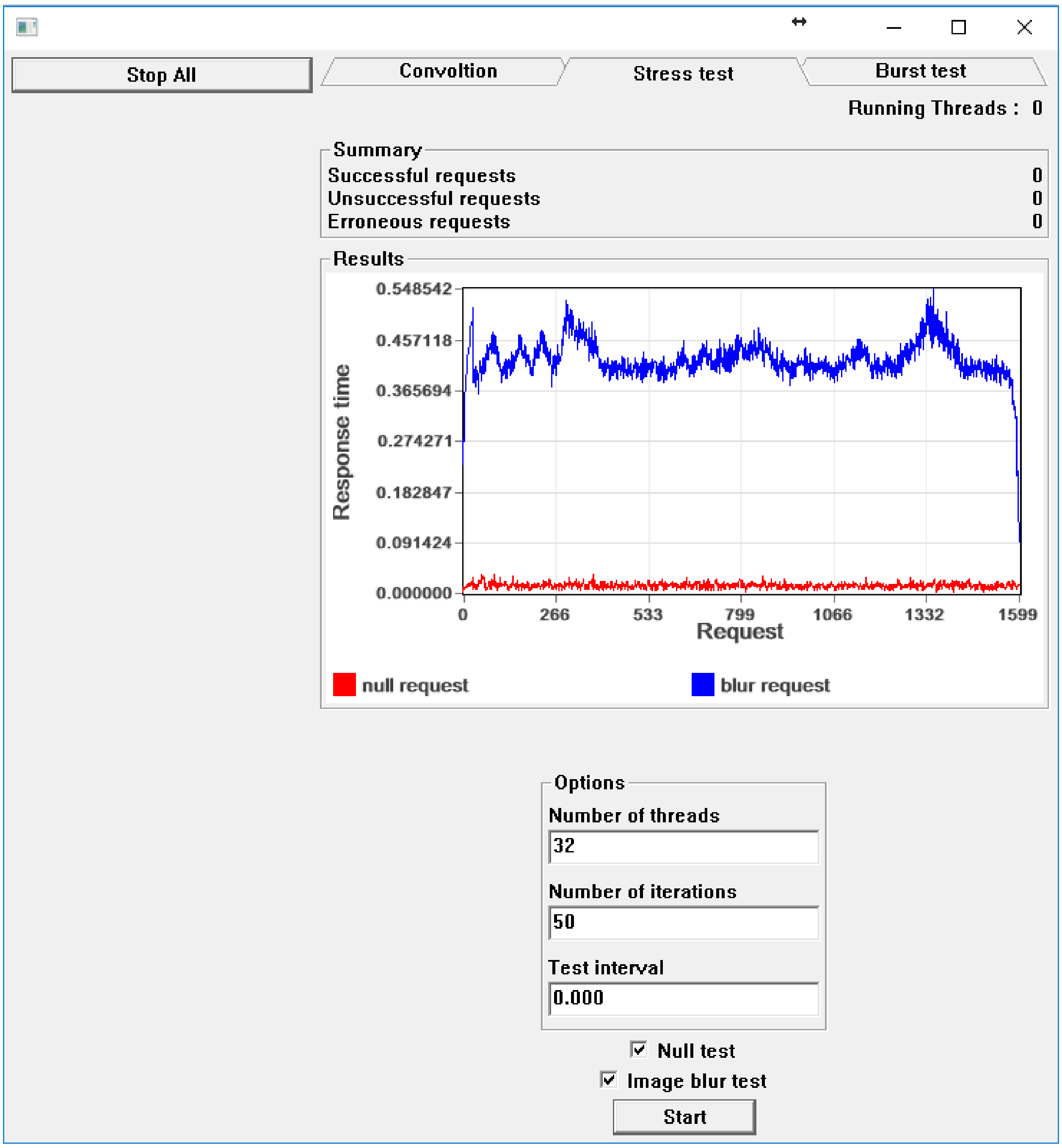
7.2.3 Burst test

This tab tests the server’s response to a sudden burst of requests. You choose the baseline requests per second, the requests per second during the spike and the duration of the spike in seconds. You also select the services being tested, like the stress test tab. When the test starts, a constant flow of requests will begin, putting the server at a low load. Then, the user will click the “spike” button, starting a brief period of a high request rate, and then receding to the background rate. The test will continue until the user clicks “stop”. Once again, when the test stops, a graph will be generated, showing the response time over the duration of the test.



Server domain
127.0.0.1
Server port
200

Figure 8: JServiceClient



Running Threads : 0

Summary

Successful requests	0
Unsuccessful requests	0
Erroneous requests	0

Results

Options

Number of threads: 32

Number of iterations: 50

Test interval: 0.000

☑ Null test

☑ Image blur test

Figure 9: JServiceClient after running a stress test

8. EVALUATION

In order to test the claims of the thesis, the applications were tuned to conduct consistent tests for all architectures. The tests were conducted on a machine with the following characteristics.

- Laptop
- Intel Core i7 4 cores, 2.2 GHz, HyperThreading
- 8 hardware threads
- 8 GB main memory
- Windows 10 64 bit

The server was parameterized to operate with 8 threads (8 minimum, 8 maximum). The client was running on the same machine, to exclude networking corner cases and connectivity issues.

I performed the stress test and burst test using the thread-based architecture and the AsynchronousProcessGraph implementation of the process graph. Moreover, the tests evaluate the “null” and “blur” services to show how the server behaves under a quick, trivial service, a computationally intense service, and both services at the same time. The tests are conducted to demonstrate the different behavior that the architectures exhibit under different circumstances.

8.1 Stress test, “blur” service

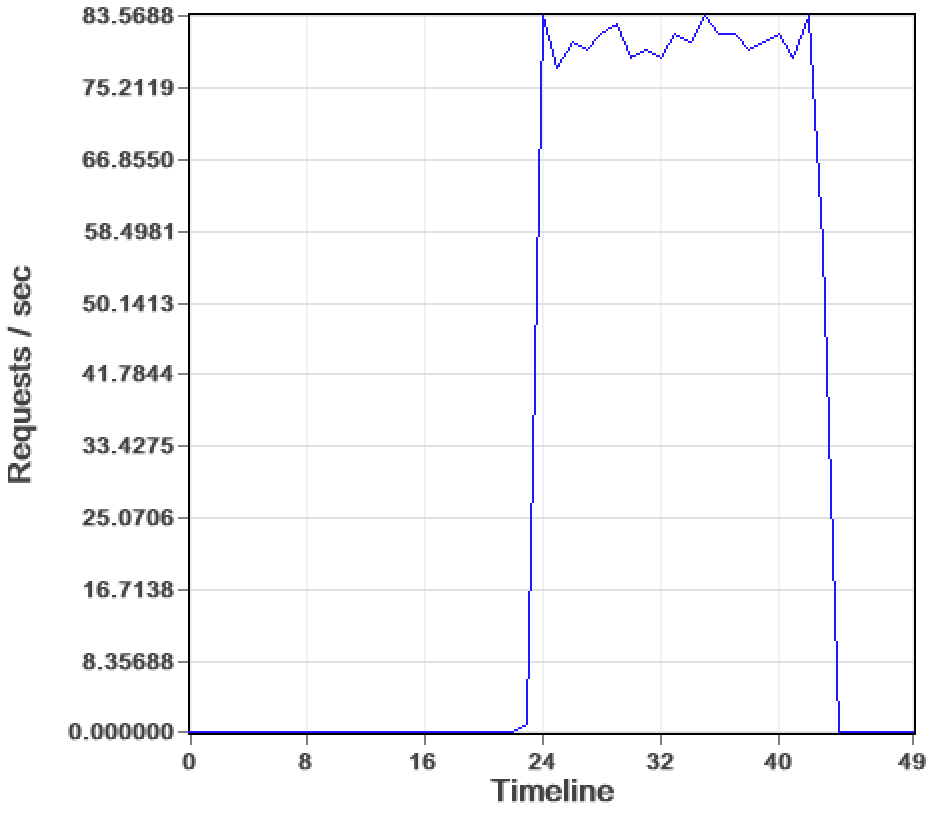


Figure 10: Stress test, "blur" service, multithreaded, server

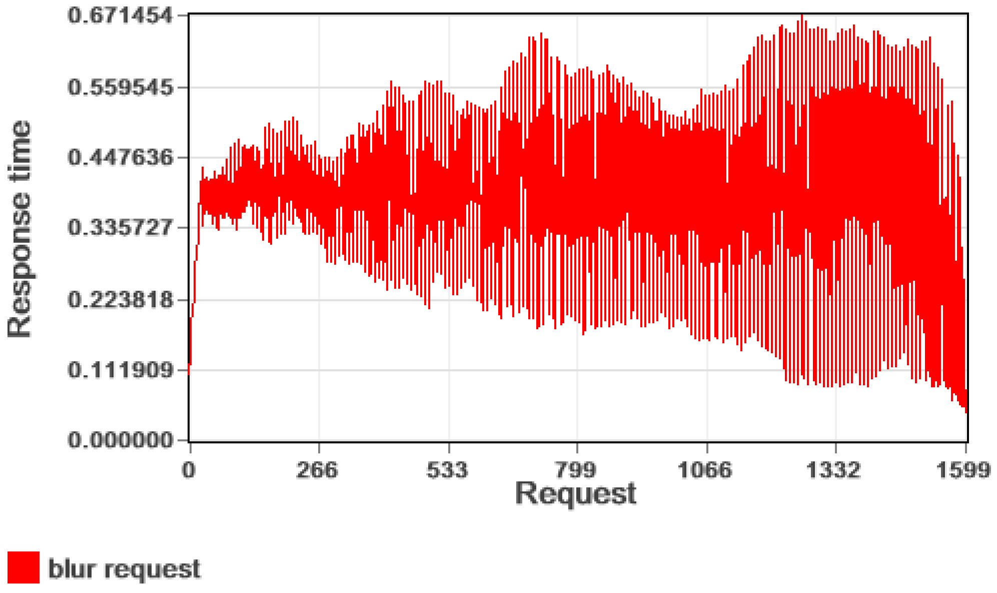


Figure 11: Stress test, "blur" service, multithreaded, client

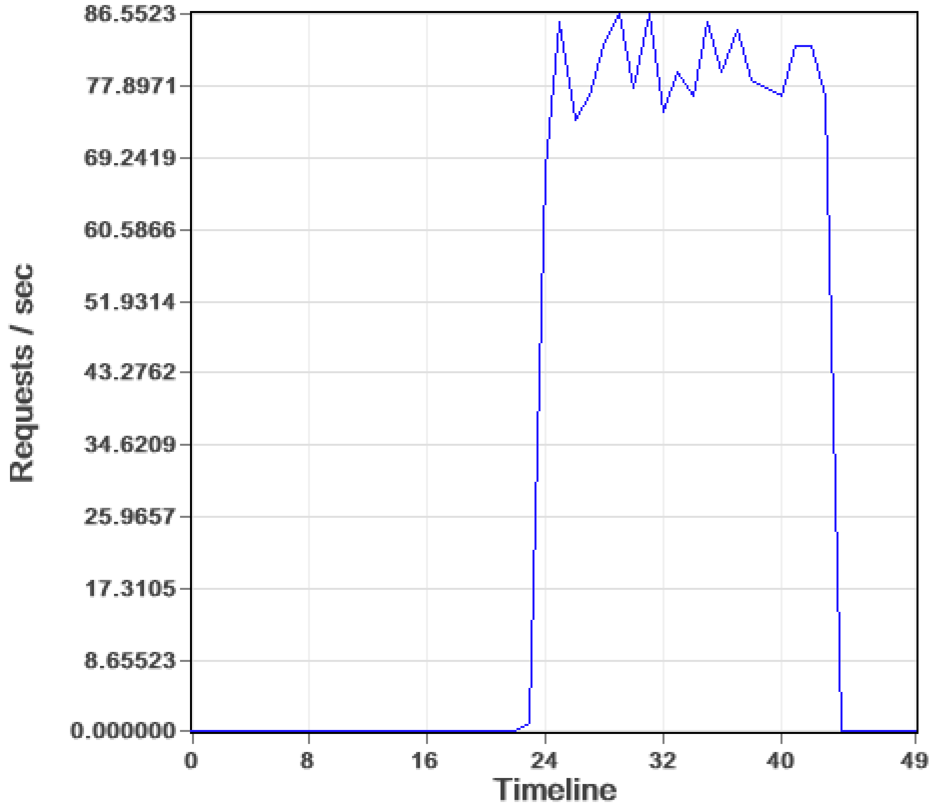


Figure 12: Stress test, "blur" service, process graph, server

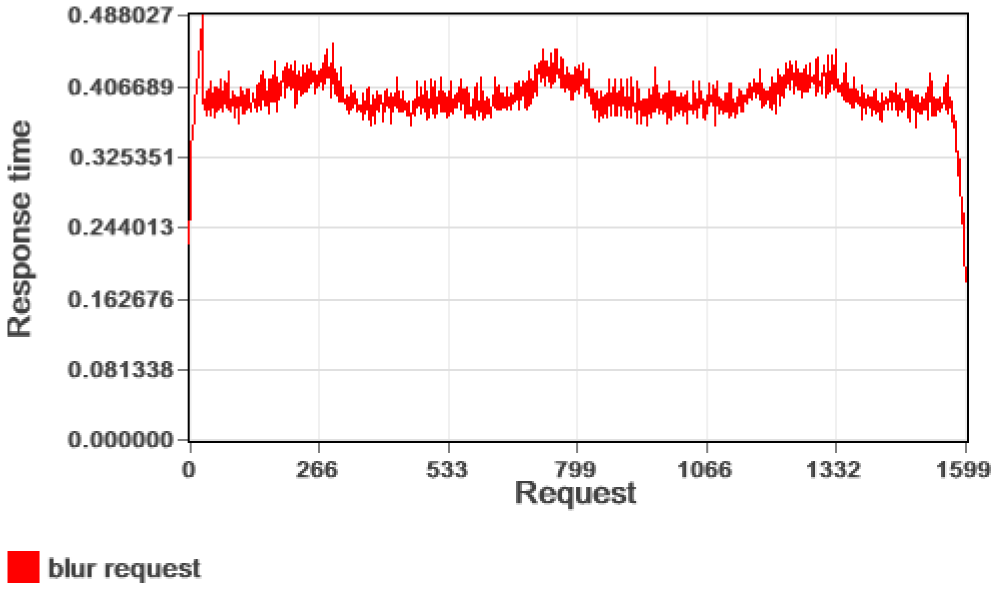


Figure 13: Stress test, "blur" service, process graph, client

We observe a consistent rate of service from the server on both architectures. For the client, however, there is a clear differentiation. For the threaded server, there is a big fluctuation of latency among the requests. The process graph seems to work in a more consistent manner in terms of latency.

8.2 Stress test, both services

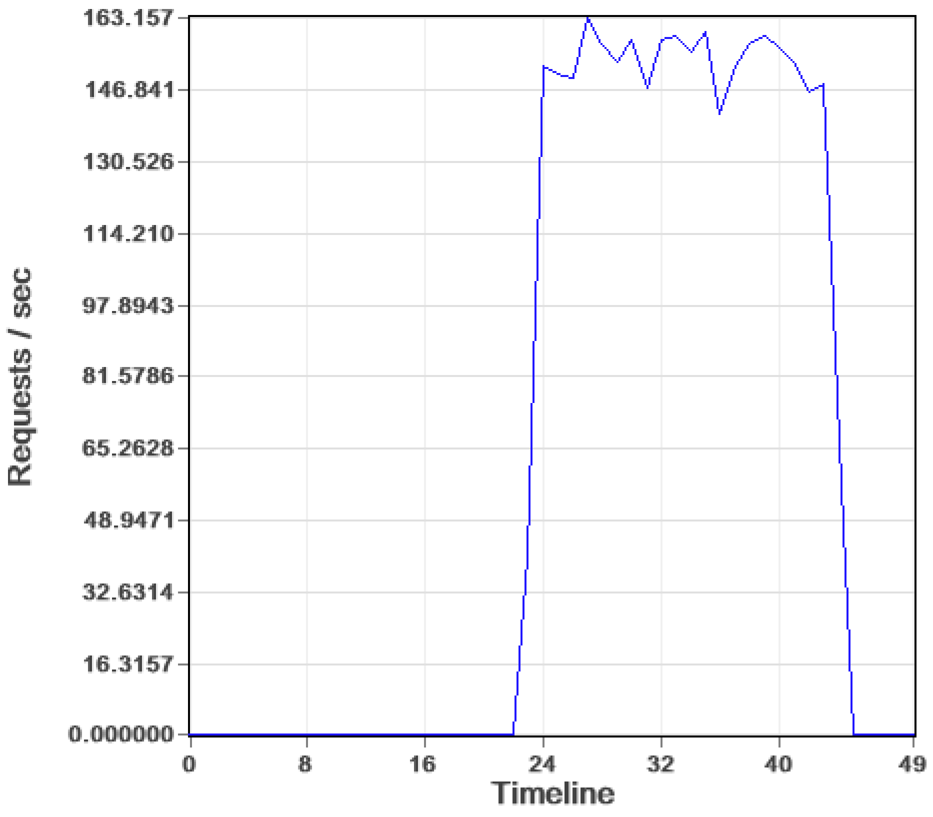


Figure 14: Stress test, both services, multithreaded, server

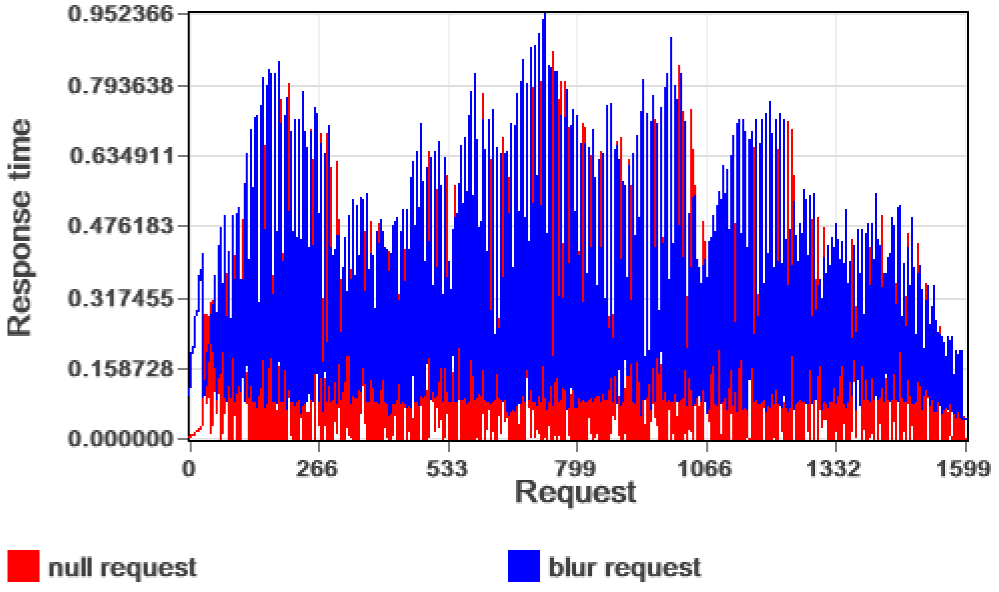


Figure 15: Stress test, both services, multithreaded, client

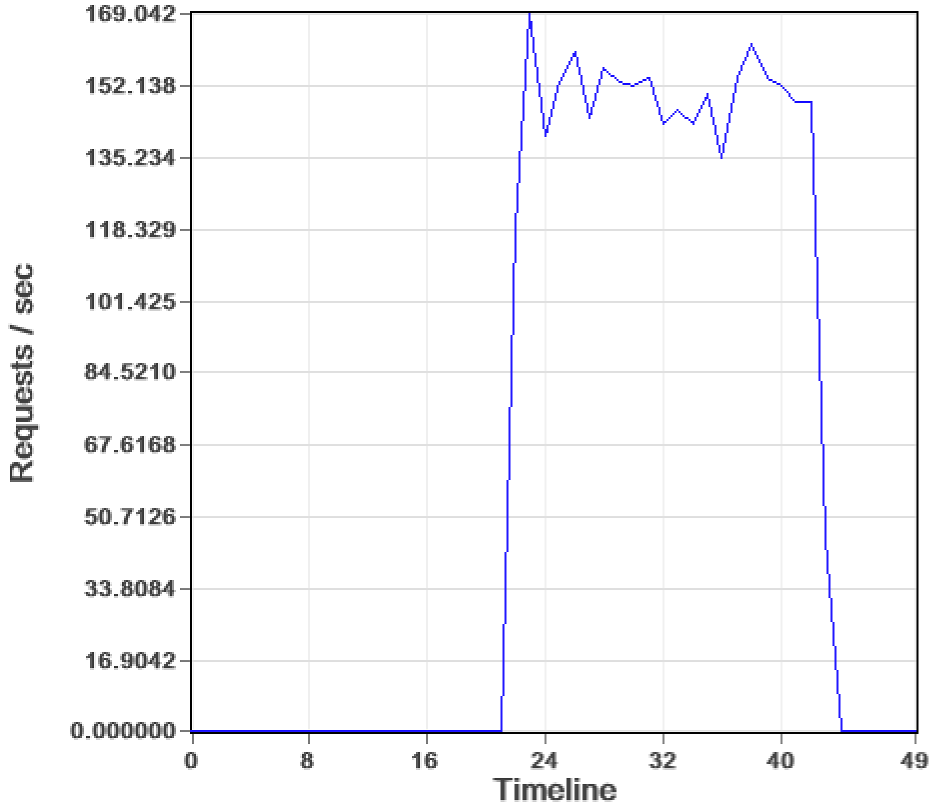


Figure 16: Stress test, both services, process graph, server

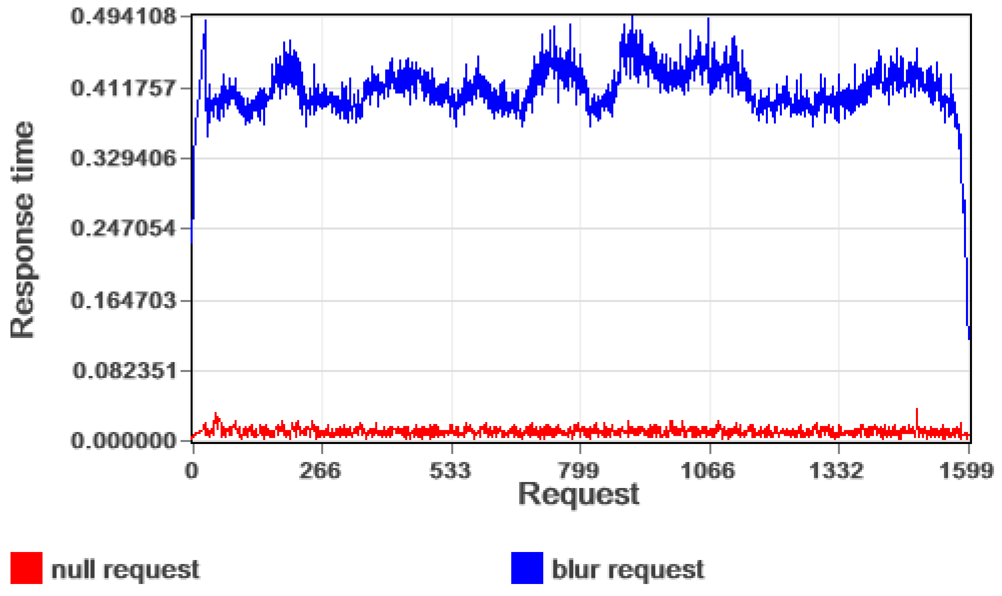


Figure 17: Stress test, both services, process graph, client

When we test both services at the same time, the difference becomes more clear. Although the server still exhibits a constant rate of service, the behavior that the client observes is widely different. Not only is there an inconsistent latency among requests, the “null” request and the “blur” request both can have big latency on the multithreaded server. The process graph implementation demonstrates superior behavior on that case, since the “null” requests are trivial and should execute almost instantly, it is expected that there would a distinct differentiation on the response times of the services.

Overall, on the process graph server, the latency seems to be under reasonable bounds with consistent behavior.

8.3 Burst test, “blur” service

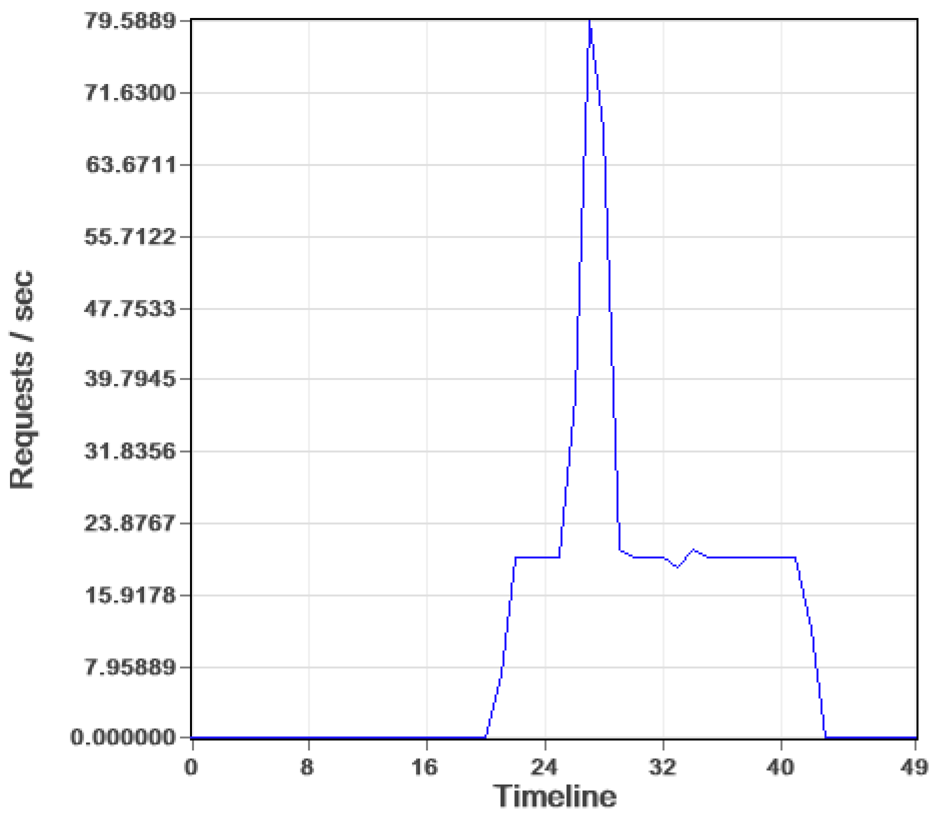


Figure 18: Burst test, "blur" service, multithreaded, server

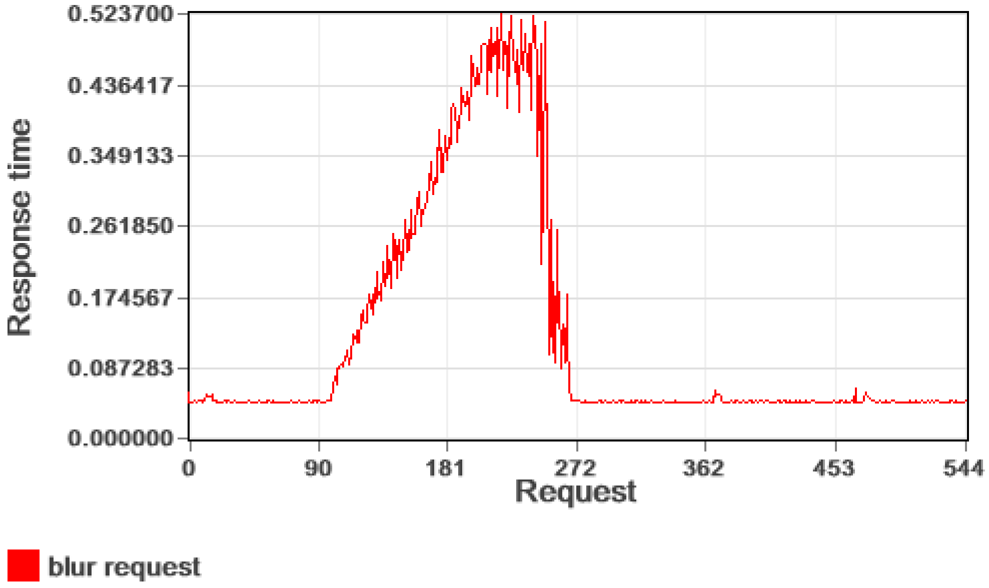


Figure 19: Burst test, "blur" service, multithreaded, client

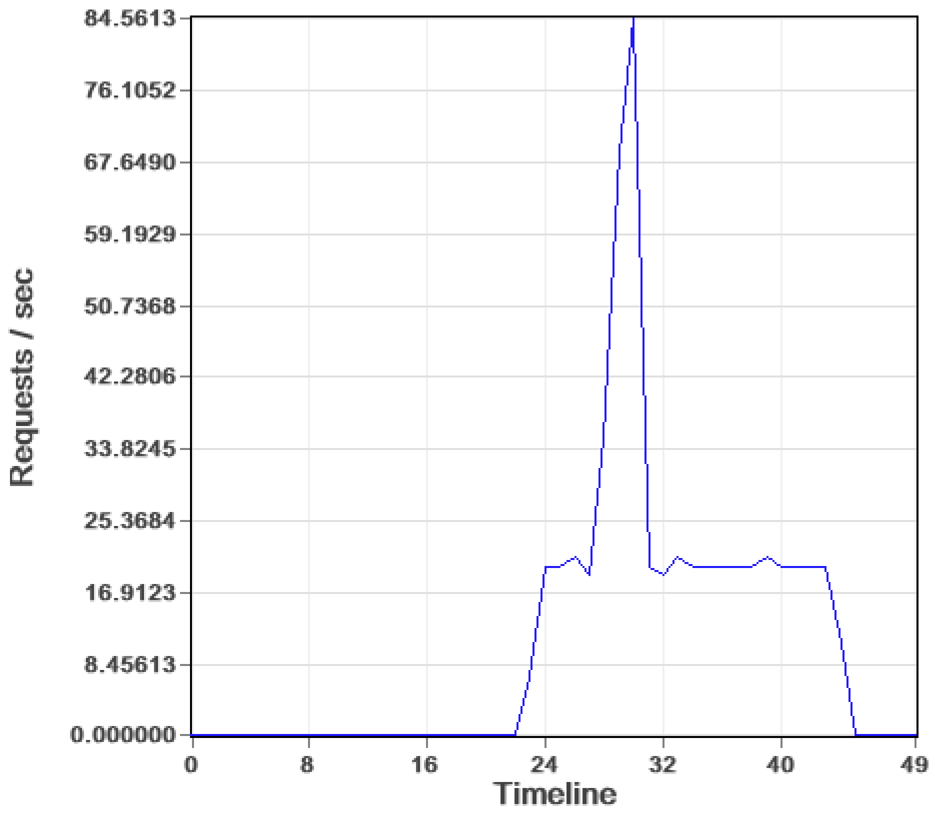


Figure 20: Burst test, "blur" service, process graph, server

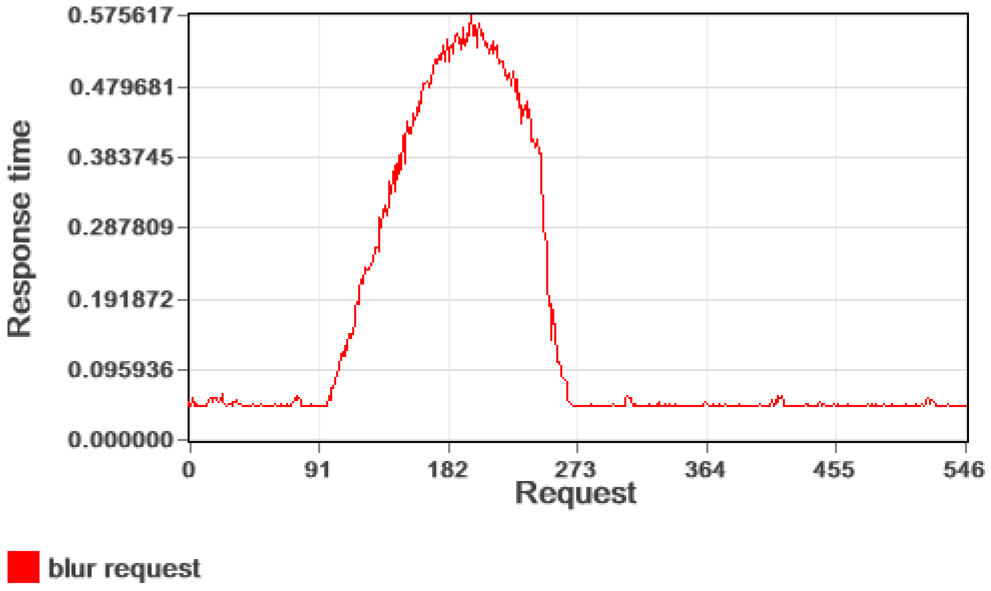


Figure 21: Burst test, "blur" service, process graph, client

We observe similar request rate behavior from both server architectures. On the clients, however, there are specific differences. First, the small latency fluctuation for the multithreaded server is present, although not so significant as under constant stress, as was demonstrated above. The other differentiation is on the shape of the curves. The threaded architecture has the expected saw-tooth appearance. The process graph, on the other hand, has a latency curve resembling a semi-circle, which, as we stated on Section 4.2 should be more fair, improve throughput, and even keep the latency under control for small bursts.

8.4 Burst test, both services

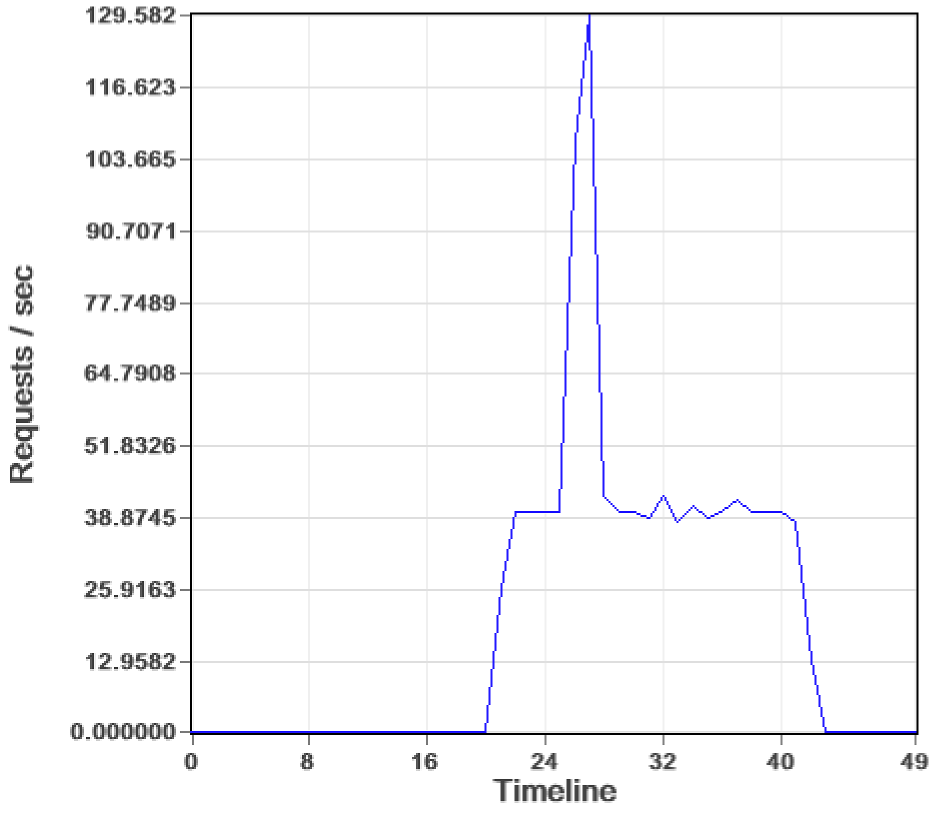


Figure 22: Burst test, both services, multithreaded, server

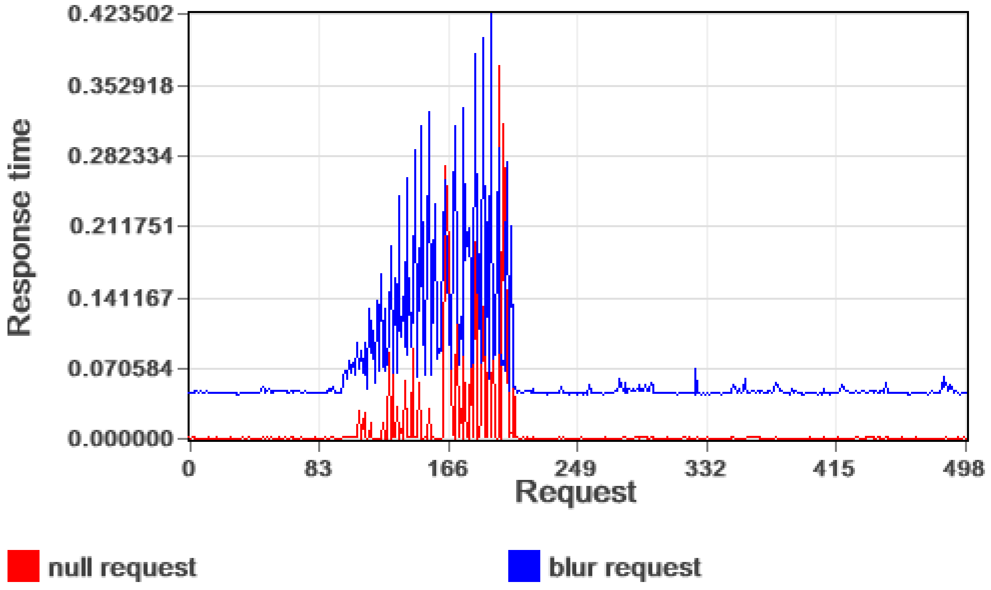


Figure 23: Burst test, both services, multithreaded, client

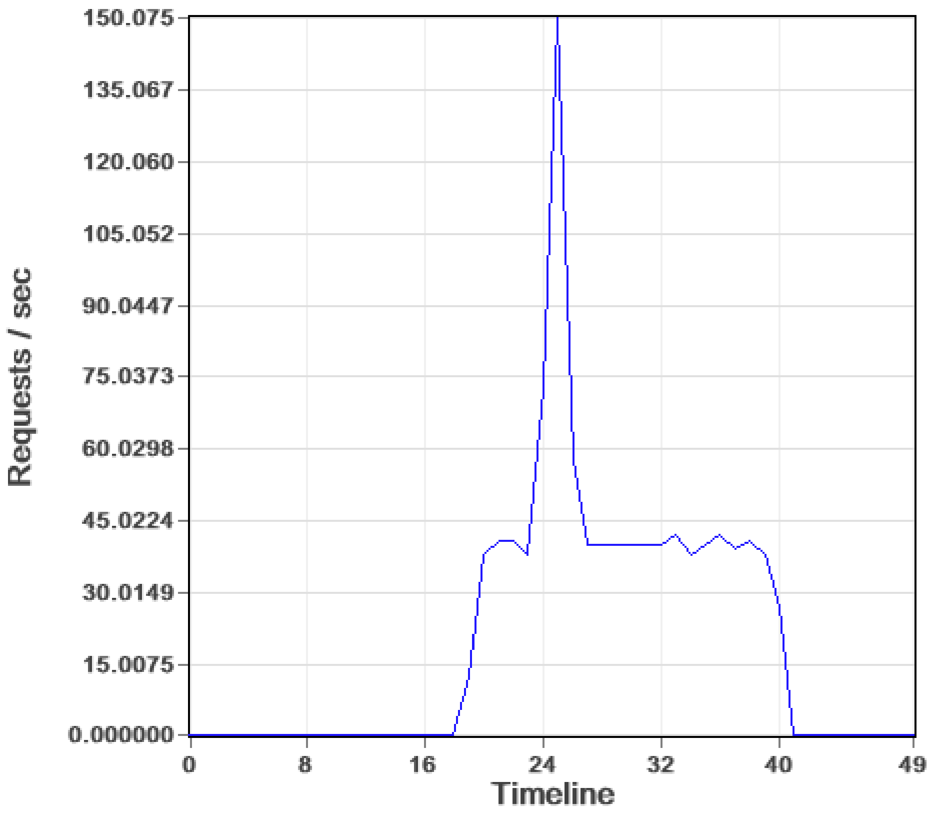


Figure 24: Burst test, both services, process graph, server

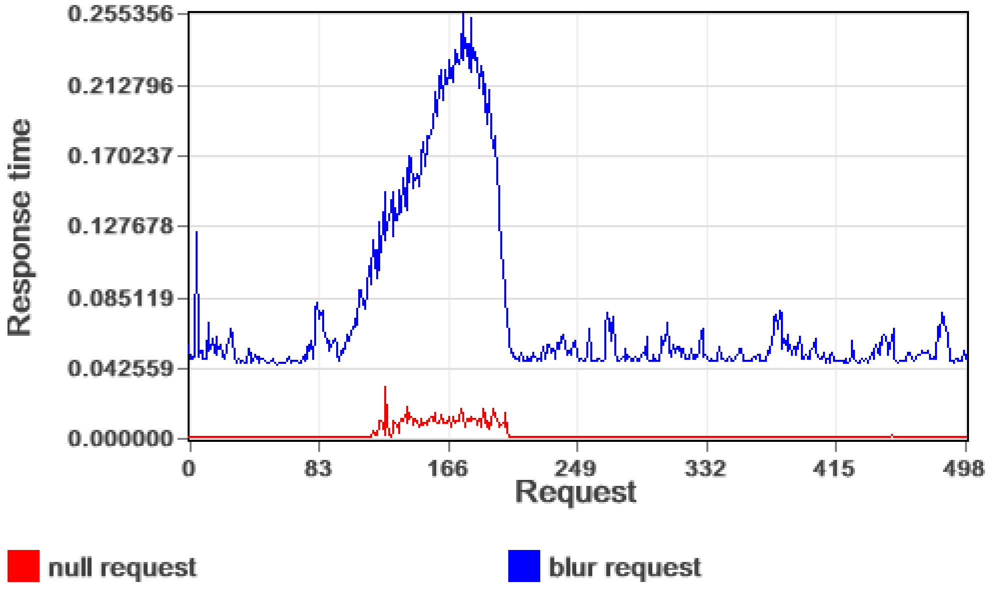


Figure 25: Burst test, both services, process graph, client

Under both services, the superiority of the process graph implementation is clear. The response time is more consistent, there is clear distinction on the response time of the “null” and “blur” services, and the latency is kept under control. On the multithreaded test you can observe outlier requests reaching over 0.4 seconds, while on the process graph all requests are kept under 0.25 seconds. A notable observation is that on the process graph, during low stress, there seems to be a small variation on response time compared to the threaded architecture.

8.5 Observations

Under the conditions of the tests, the results are consistent with the prediction of the analysis of the architectures.

- The multithreaded server is slightly more stable under a very small load.
- The process graph implementation is significantly more consistent and stable under heavy load and on bursts.
- The process graph implementation minimizes outliers; requests with unreasonably high response time are fewer and more bounded.
- Slow requests don’t affect the latency of fast requests on the process graph.
- Sudden changes on load are handled more gracefully and fairly on the process graph implementation.
- Even though on both architectures the server processed a similar rate of requests per second, the architecture and policies of the server significantly alters the behavior on individual requests and the overall responsiveness towards the client.

We must note that the architectures being tested and the testing conditions are specific and selected to demonstrate certain characteristics of the process graph. A properly deployed test should operate under a real network, multiple client machines, dedicated server machine and a battery of tests under all common and corner cases, but in the context of this thesis there was no time or necessary resources to accomplish that.

Also, the multithreaded implementation was not the traditional thread-per-request model. Some characteristics, like small requests along with expensive requests having similar response times would not be exhibited. I chose this multithreading model, limited kernel threads executing requests from a shared queue, to decouple thread performance and architecture performance. If the threading package is expensive in terms of memory footprint and context-switching, a limited-threads architecture will perform better independently of the intrinsic properties of the architecture. On the other hand, if a user-space, low overhead, scalable threading package is used, the multithreaded solution might outperform other architectures, making the comparison more difficult.

9. FUTURE WORK

The current process graph system is a simple model with two basic implementations. I consider it to be only the beginning, though. There is much experimentation that can be done in terms of alternative architectures. There can be implementations with configurable policies regarding prioritization of requests. Since the core concept is so abstract, further research into this concept is almost limitless.

The core process graph system and library can be further improved in the following areas:

- Develop more novel process graph implementations
- Optimize the core library and process graph implementations
- Multiple architectures being switched among during runtime
- Dynamic graph management during runtime
- Stand-alone process graph server application with a dynamic graph that can be controlled externally via IPC
- Collection of optimized general purpose nodes
- Distributed process graph running on multiple connected machines
- Optional message-based processing items instead of pointer-based
- Monitoring tools and testing framework
- Detailed documentation and language support

Moreover, it is essential that the architecture should be evaluated even more thoroughly. In the context of the thesis, I performed a basic proof-of-concept evaluation on a trivial server configuration, comparing two basic architectures. Some of the proposed benefits could not be shown from such a test. More work should be done on this area.

- Implement real-world services and real-world testing
- Develop services that utilize asynchronous I/O on node barriers to demonstrate the premise that the performance should approach event-based servers
- Test against event-based architectures
- Test against other hybrid architectures like SEDA
- Test against thread-per-request multithreaded architectures
- Try to match and exceed commercial server application software
- Utilize professional, trusted benchmarking and testing software

10. CONCLUSIONS

Server performance and behavior is closely tied to the underlying hardware, OS and server architecture and deployment, and request characteristics, load and distribution. Selecting the appropriate architecture to maximize performance is bound to depend on all the other factors. Since the hardware, OS, applications and server research are constantly changing and evolving, it would be unreasonable to expect that server architectures would remain the same. There is no single architecture that is always optimal.

The process graph approach has the advantage of not being tied to a specific architecture or application. Even though I performed a rudimentary analysis of potential benefits performance wise, and conducted a basic evaluation, further development on the subject should yield even more impressive results.

Another point that I should make is that the ease of development can be even more important than minor performance gains. Usually, one of the most expensive aspects of an application is the human work required to develop and maintain a particular system. I believe that a good abstract model of computation can drive these costs down. The testability, maintainability, clarity and ease of development that a modular approach like the process graph offers could go a long way towards that goal.

Many modular approaches to computation, like pipelines, staged architectures, event graphs and queue systems can be generalized to an abstract graph of computations. This is the reason that a process graph can be intuitive as a natural all-encompassing architectural approach. Usually, such systems are analyzed as graphs during the initial stages of development. Using an architecture like the process graph maintains this nature, and the resulting code can be easier to reason about.

Since Internet services and applications are ever more pervasive and ubiquitous, and internet penetration is always increasing, there has been a need for powerful service deployment. Server architecture engineering is constantly evolving and expanding. Research on this area has and will be essential to be able to support the massive demand of efficiency and performance of applications and services.

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ / ABBREVIATIONS

HTTP	Hyperlink Text Transfer Protocol
DBMS	DataBase Management System
RPC	Remote Procedure Call
CPU	Central Processing Unit
I/O	Input / Output
SEDA	Staged Event-Driven Architecture
API	Application Programming Interface
TCP	Transmission Control Protocol
OS	Operating System
JSON	JavaScript Object Notation
IPC	Inter-Process Communication

ΑΝΑΦΟΡΕΣ / REFERENCES

[1] Lauer, H. C., & Needham, R. M. (1979). On the duality of operating system structures. ACM SIGOPS Operating Systems Review, 13(2), 3–19.
[2] Welsh, M., Culler, D., Brewer, E., Welsh, M., Culler, D., & Brewer, E. (2001). SEDA: an architecture for well-conditioned, scalable internet services. In Proceedings of the eighteenth ACM symposium on Operating systems principles - SOSP ’01 (Vol. 35, pp. 230–243). Banff, Alberta, Canada: ACM Press.
[3] Larus, J. R., Parkes, M., & Larus, J. (2001, March 1). Using Cohort Scheduling to Enhance Server Performance.
[4] von Behren, R., Condit, J., Zhou, F., Necula, G. C., & Brewer, E. (2003). Capriccio. ACM SIGOPS Operating Systems Review, 37(5), 268.
[5] Cooper, R. B., & B., R. (1981). Queueing theory. In Proceedings of the ACM ’81 conference on - ACM 81 (pp. 119–122). New York, New York, USA: ACM Press.
[6] Amin, A., Mehta, P., Sahay, A., Kumar, P., & Kumar, A. (2014). Optimal solution of real time problems using Queueing Theory. International Journal of Engineering and Innovative Technology (IJEIT), 3(10).
[7] Alfa, A. S., & Isotupa, K. P. S. (2004). An M/PH/k retrial queue with finite number of sources. Computers & Operations Research, 31(9), 1455–1464.
[8] Kumar, R., & Kaur, J. (2008). Towards a Queue Sensitive Transport Protocol. In 2008 IEEE International Performance, Computing and Communications Conference (pp. 319–326). IEEE.
[9] Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., Cheriton, D. R., … Cheriton, D. R. (2007). Comparing the performance of web server architectures. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 - EuroSys ’07 (Vol. 41, p. 231). New York, New York, USA: ACM Press.