CHALMERS
UNIVERSITY OF TECHNOLOGY

Back-off Regulator for Improved Throughput, Congestion Avoidance and Fairness

Master's thesis in Computer Systems and Networks

EMIL KRISTIANSSON
JOHAN PERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

Back-off Regulator for Improved Throughput, Congestion Avoidance and Fairness

EMIL KRISTIANSSON
JOHAN PERSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Networks and Distributed Systems
Chalmers University of Technology
Gothenburg, Sweden 2016

Back-off Regulator for Improved Throughput, Congestion Avoidance and Fairness
EMIL KRISTIANSSON
JOHAN PERSSON

Supervisor: Elad Schiller, Department of Computer Science and Engineering
Supervisor: Iosif Salem, Department of Computer Science and Engineering
Examiner: Olaf Landsiedel, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Networks and Distributed Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Back-off Regulator for Improved Throughput, Congestion Avoidance and Fairness
EMIL KRISTIANSSON
JOHAN PERSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In this thesis, we present an architectural component, the task request regulator, for flow control of incoming server-task requests that are made by a large-scale number of clients. The challenge is to keep the server at high utilization levels while avoiding overloads. Our solution is based on ad-hoc (re)scheduling of incoming client server-task requests. Namely, the regulator can order the client to back-off and return at a server-convenient time.
Our solution includes a regulator that monitors the server load and tries to keep the number of client-requests at service at a preferable level. We have designed and demonstrated, both analytically and experimentally, three algorithms for implementing the regulator. The first algorithm is elegant, has modest implementation requirements but provides no fairness guarantees. The second algorithm has a shorter convergence period than the first one, at the expense of a modest increase in the storage and communication costs (but provides no fairness). Our third proposal is an extension of the first two algorithms which provides fairness with respect to the number of rescheduling events that a task may get, at a small added computational cost for the regulator.

Keywords: distributed systems, throughput, congestion avoidance, flow control, scheduling, fairness

Acknowledgements

Many people have been involved during the life-time of this thesis. We would like to thank, in no particular order:

Elad Schiller, for his valuable ideas and suggestions during the planning and development of this work.

Iosif Salem, for his assistance with the formal analysis of our algorithms.

Qmatic AB, for letting us work in their offices and use their equipment and software. The technical expertise of their staff has been invaluable.

Johan Andersson, for helping us set up our environment and providing insight into Qmatic's systems.

Pierre Leone, for his help with the peer-review of the report.

Sofia Larsson, for her help with proof-reading and quality control of the report.

Emil Kristiansson, Gothenburg, September 2016
Johan Persson, Gothenburg, September 2016

Contents

1
Introduction

Managing large scale distributed systems presents many challenges. We consider the problem of load management and task scheduling. Specifically, we consider the throughput of a central application server that performs tasks requested by a large set of remote clients. Each client opens a connection with the server and spawns one task for the server to process. The server processes tasks at a rate that is dependent on the number of concurrent connections, i.e., the server load. One of the challenges is to keep the server utilization high. For example, if the number of concurrent connections is too low, the server may not be able to fully utilize its resources. If the number of concurrent connections is too high, the work load could lead server overload, i.e., resource exhaustion and growing overhead from context switching et cetera. We propose a new architectural component, the regulator, that balances the server load and the server utilization. To that end, it instructs clients to defer their requests according to a targeted server load. We develop two algorithms, consider fairness and validate our proposal via experiments.

We assume that the there is a known number of concurrent connections to the application server which results in targeted server performance. We use this level to impose an application level restriction on the number of concurrent connections. Then, we use the TCP backlog queue as a buffer between the regulator and the application server. The regulator then uses the level of the backlog to decide when a client can access the server. If a client is denied access, the regulator calculates a time for the client to try to access the server again. The concept is that the regulator emulates a virtual queue. It holds any clients that cannot fit on the application server (which includes the backlog).

We present two algorithms for managing the virtual queue. The Dynamic Insert with Bounded Append algorithm (DIBA), is an elegant and computationally cheap algorithm to deal with our problem. It loosely controls the virtual queue, and performs at its best when the task completion rate of the server is constant. In addition to DIBA, the Vector algorithm allows more fine-grained control of the queue by splitting it into several parallel queues with distinct return rates. As a result, it offers convergence after a change of the task completion rate. We also introduce an algorithm to manage fairness, by prioritizing clients that have been denied access to the server multiple times. We have conducted an initial analysis and experimental tests on all algorithms.

The goal of our solution is to provide good throughput, with some level of fairness, while avoiding congestion. On the contrary to our solution, one trivial way for trying to ensure that the application server always has tasks to process is to let clients immediately retry when rejected, i.e., brute-force. Compared to our solution,

this would impact the throughput and responsiveness as it would greatly increase the congestion. Additionally, if no measure is taken to limit the number of connections to the application server, a brute-force approach could lead to excessive server load.

To validate our algorithms, we have constructed an experimental testbed where the regulator algorithms are implemented against a real world application server[1] in the Microsoft Azure[2] cloud environment. Client requests are mocked using the stress-testing tool Gatling[3]. The results show good performance for the studied test scenarios. We have also developed a method for estimating the task completion rate, by measuring task completion times.

1.1 Related work

The C10k [7] problem, and more recently C10M [9], deals with the challenge of handling a large number of concurrent server connections in the transport layer. Optimising network sockets while avoiding blocking techniques is an important aspect in solving these kinds of issues. However, even after the transport layer has been tuned to handle the large number of connections, the system needs to be able to handle the workload on the application layer as well.

Limiting the number of concurrent connections to a server is well-known approach to avoid overload. However, such solutions do not generally consider what happens with any denied connections. Clients might try to reconnect immediately, at a later time, or not at all. To control how clients reconnect, some kind of backoff protocol is often employed. One well-known example is exponential backoff. However, exponential backoff is noted to have sub-constant throughput with the number of connections [3]. Bender et al. [4] present a modification of exponential backoff that provides constant throughout and polylog access attempts. Unlike us, they assume the presence of a global broadcast channel.

Traffic policing and traffic shaping can be employed to control network throughput. However, the concept of flow control mostly deals with managing traffic per client, and not managing the flow of clients themselves [11].

Cederman et al. [5] studies how to choose programming language, as well as synchronization methods to achieve the best throughput. Atalar et al. [2] propose a method to estimate throughput for lock-free algorithms and, as a consequence, a way to design a back-off method to maximize throughput. These methods try to increase the throughput by focusing on the software.

Another perspective that can be taken into consideration is the power consumption of the server. Li et al. [8] proposes the application server Chameleon, which they describe as a novel adaptive green throughput server. It uses multiple power management policies combined with learning algorithms to provide best throughput per power consumption.

Our regulator-server interaction can be viewed as a variation on the leaky bucket

[1] Qmatic Orchestra
[2] https://azure.microsoft.com
[3] https://gatling.io

as a meter [10], where non-conforming packets (requests) are not allowed to proceed to the server, with the addition of a scheduling component for any dropped requests. Thus, the purpose of the regulator is to act as a work-conserving scheduler [6], i.e., a scheduler that keeps the server busy as long as there are waiting tasks.

The analytical part of this thesis leverages queueing theory [1] to show scenarios where our algorithms work well.

1.2 Our contribution

We introduce a new architectural component, the regulator, capable of managing the flow of incoming server connections. As a part of the regulator, we present two different algorithms for managing a virtual queue. Our DIBA algorithm is suitable for a server with a fairly constant task completion rate, whereas the Vector algorithm allows for more control over the virtual queue. As a consequence, the Vector algorithm is able to tolerate more dynamic server behavior. To provide fairness in our system, we also propose an algorithm to use with our two virtual queue algorithms. We have done initial analysis and experimental testing for all algorithms. Via the experimental tests, we see that the server load is kept at the targeted level, and we have some level of fairness.

1.3 Limitations

In our experimental setup, we only consider tasks of fixed size, i.e., the task completion rate of the application server is roughly constant. As a consequence, our task completion rate estimation algorithm uses a non-aging average of all job times to compute the task completion rate.

Additionally, the task completion rate algorithm needs to be initialized with a value that is greater than or equal to the actual task completion rate of the server, to ensure that the system functions correctly while the estimation algorithm converges on the correct rate.

Our system works under the assumption that a task does not need to be performed immediately. An example where this assumption does not hold could be a web server, where users are actively waiting for a response from the web server. An example where this assumption does hold could be a system where clients perform nightly tasks on a server, where it does not matter when during the night a task is executed. This assumption allows us to measure fairness solely by the number of access attempts.

We only consider the case where each client has exactly one task that it wants to run on the application server, i.e., the number of tasks to be performed by the server is finite. Although, this can result in many requests due to rescheduling. If this assumption is removed, and no additional assumption about the arrival distribution is made, the analysis of the fairness algorithm also has to consider this.

2
System Architecture

The system consists of an application server, regulator, firewall and a large number of clients. The clients request access to the application server via the regulator (Figure 2.1).

The regulator is between the firewall and the application server. It has access to the current number of queued and active connections. The regulator advises the remote clients about their back-off period whenever the number of queued connections at the application server exceeds the limit. Thus, the regulator places clients either in a queue on the application server or in a virtual queue to the regulator, when the application server queue is full.

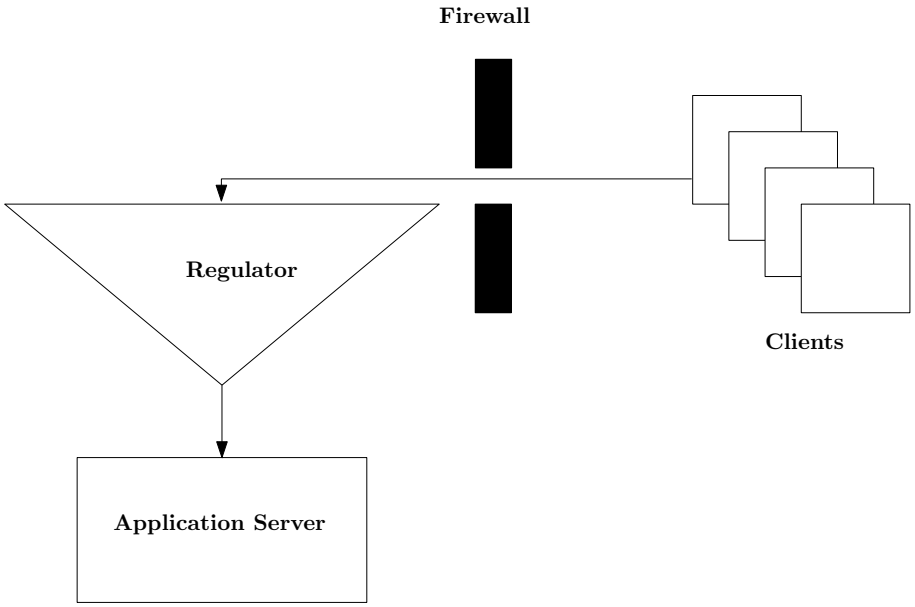


Figure 2.1: System overview

2.1 The Application Server

To achieve good troughput on the application server we want to limit the number of concurrent connections, c, that the server will accept to the application concurrently. The intention is to keep the level constant at some chosen level, c_{chosen}, by maintaining a queue to the application.

The queue management on the application server should be computationally cheap, therefore we use the TCP backlog as a queue. We do this by holding off the

accept() system call, leaving excess clients in backlog the queue. One example of how this can be implemented is described further in Section 5.2.1.

The maximum size of the TCP backlog is controlled by the application server kernel. Additionally, there may be restrictions on the maximum time a connection can spend in the backlog, before the application server and/or the client assumes there has been a problem with the connection. Additionally, the firewall may place limits on how long connections can stay idle. These parameters may need to be altered depending on desired queue size and task completion rate.

For the queue, we declare low and high water mark levels (LWM and HWM, respectively). LWM represents the minimum queue size needed to guarantee high server utilization, while HWM is an upper boundary on how many (or how long) clients we allow to be waiting, to avoid timeouts et cetera. Thus, our algorithm should aim to keep the queue size at an aimed mark (AM) somewhere between LWM and HWM.

2.2 The Regulator

We want the number of concurrent connections to the application server, c, to be steady at some chosen level, c_{chosen}. We can achieve this by ensuring that there are always pending requests in the queue on the application server, by trying to keep the level of pending requests at an aimed mark, AM (explained in section 2.1).

The regulator can be viewed as two separate functionalities, one that schedules incoming connections and one that monitors the server, however, we will refer to the regulator as one component.

2.2.1 Scheduling Functionality and the Virtual Queue

When the server recieves excess clients, e.g., when the backlog is full, they are told to return at a later time. This means that there can be multiple clients waiting to return to the server at the same time, this concept or group of clients we call the virtual queue. The virtual queue is an abstracion for waiting client and is controlled by setting the return time, i.e., waiting time, of waiting clients. The return rate of the queue is meant to match the task completion rate of the server and returning clients from the virtual queue should also be given priority over incoming requests.

Algorithms for the scheduling module appear in Section 3.1.

2.2.2 Monitoring Functionality and the Need for Sharing Information between Application Server and Regulator

The regulator requires access to information about the application server in order to decide the feed rate to the application server. Such information includes the number of concurrent connections to the application server, backlog length and task completion rate. Therefore, the application server needs to continuously provide the regulator with information.

2.3 Access Tokens

The proposed architecture uses tokens for granting the remote clients access to the application server. This way we can let remote clients connect directly to the application server once given an Access Token by the regulator (Table 2.1).

Table 2.1: An Access Service message

Field	Value	Comments
type	Access Service	Identifies what type this message is
accessToken	Number	A token for accessing the application server

2.4 Client States

A sequence is initialized by a client by sending a request to the regulator. The structure of the Request message is described in Table 2.2. If the connection is not dropped by the firewall, it will open a connection with the regulator and wait for a reply. If the reply is an Access Service message, as described in Table 2.1, it will close the connection with the regulator and open a new connection to the application server and access its service. Otherwise, if the reply is a Schedule message, as described in Table 2.3, the client will increment the numbTries-field and will retry after waiting the time given in the Schedule message returnTime-field. However, if the connection to the regulator cannot be established, then we can assume that the message was dropped at the firewall due to too many concurrent connections. The occurrences of this phenomena is something that our approach aims to greatly reduce. An illustration of the client states is presented in Figure 2.2, where an Access Service message is denoted by "Go", and a Schedule message is denoted by "Wait".

Table 2.2: A Request message

Field	Value	Comments
type	Request	Identifies what type this message is
numbTries	Number	The number of times a client has previously contacted the regulator

Table 2.3: A Schedule message

Field	Value	Comments
type	Schedule	Identifies what type this message is
returnTime	Number	The time, for when the client should return

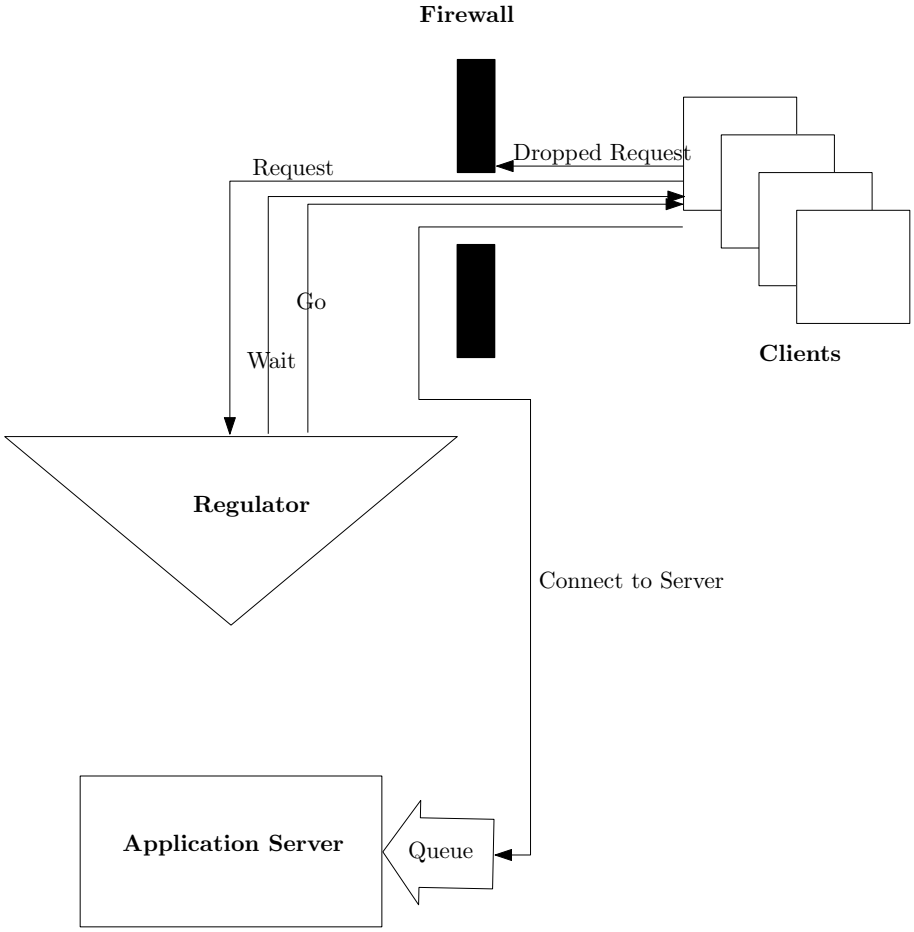


Figure 2.2: Client states with a queue located on the application server

3
Algorithms

In this chapter, we present the different algorithms for the system.

Any request to the regulator is initialized by a client (Algorithm 1). Depending on the reply, the client will either proceed in connecting to the application server, or wait the time specified in the reply and then contact the regulator again.

Algorithm 1: Code for remote Clients

```
Data: numbOfTries - The number of times the client has sent a request to
      the regulator
1 init begin
2     numbOfTries = 0 send (Request, numbOfTries) to Regulator
3 upon arrival ⟨Wait, retryTime⟩ from Regulator do begin
4     increment numbOfTries
5     send (Request, numbOfTries) to Regulator atTime retryTime
6 upon arrival ⟨Access Service⟩ from Client do begin
7     access application server /* implementation specific          */
```

If there are empty slots on the application server, it continuously processes tasks from the backlog (Algorithm 2). Whenever a task is completed, the application server provides the regulator with information about its state, i.e., the number of concurrent connections and the time each task takes to complete. The regulator then keeps track on the virtual queue and decides if incoming client requests should gain access to the server or be deferred to the virtual queue.

Algorithm 3 contains the basic behavior of the regulator. It is responsible for deciding if a client should be given access to the application server, or if it should be rescheduled in the virtual queue. This decision is based on the current backlog level of the application server, i.e., Algorithm 3 checks if the backlog level is below some given threshold. If not, Algorithm 3 uses the *getReturnTime* interface to call one of the virtual queue algorithms from Section 3.1 to decide when the client should return. Algorithm 3 also handles incoming information from the application server.

In Section 3.1 we present two different approaches for controlling the virtual queue. Then, in Section 3.2, we present a way to estimate the task completion rate of the application server. Finally, in Section 3.3 we present an improved version of Algorithm 3, to introduce fairness to the system.

Algorithm 2: Code for Application Server

Data: queue - pending tasks
Data: c - The number of tasks currently running on the server
Data: c_{chosen} - The chosen number of tasks concurrently running on the server

```
1  do forever begin
2      dequeue a task from the queue if c < c_chosen
3  upon arrival ⟨Access Service, task⟩ from Client do begin
4      add task to queue
5  upon completion of task do begin
6      send ⟨taskCompletionTime, queue.length()⟩ to Regulator
```

Algorithm 3: Basic regulator behavior. Decides if a client can access the server.

Data: *desiredReturnRate* - desired return rate from virtual queue
Data: *backlogLevel* - number of clients in the application server backlog
Data: *virtualQueueLength* - number of clients in the virtual queue
`/* virtualQueueLength - Not used in the Vector algorithm */`
Data: γ - constant controlling what clients gets priority (defaults to 0)
Data: β - constant controlling for how long clients will get priority (defaults to $\frac{(HWM+AM)}{2}$)

```
1  Interface getReturnTime(desiredReturnRate) - calls one of Algorithms 4, 5, 6 or 7
2  upon arrival ⟨Request, numbOfTries⟩ from Client do begin
3      if numbOfTries > 0 then
4          decrement virtualQueueLength
5      if backlogLevel < AM OR ( numbOfTries > γ AND backlogLevel < β ) then
6          send (Access Service, token) to Client
7      else
8          increment virtualQueueLength
9          returnTime ← getReturnTime(desiredReturnRate)
10         send (Wait, returnTime) to Client
   /* Receives updates from the application server */
11 upon arrival ⟨arrBacklogLevel, arrJobTime⟩ from Application Server do begin
12     backlogLevel ← arrBacklogLevel
13     desiredReturnRate ← calculateDesiredReturnRate(arrJobTime)
```

3.1 Virtual Queue

As was described in Section 2.2.1, the virtual queue i an abstraction of the waiting, returning, clients and it is by controlling their return rate we can ensure a steady flow of incoming jobs to the server.

Our two algorithms for controlling the virtual queue are DIBA (Algorithm 6) and Vector (Algorithm 7). The Vector algorithm is designed to perform better when the task completion rate fluctuates.

Algorithms 4, 5, 6 and 7 are all different implementations of the $getReturnTime()$ interface from Algorithm 3. They all receive a desired return rate as input, and return at what time the next client should return, but differ in how they calculate that time.

3.1.1 DIBA - Dynamic Insert, Bounded Append

In this section, we present DIBA, our first implementation of a virtual queue. The idea is to control the placement of requests in the virtual queue by keeping track of the end time of the queue, as well as the number of clients in the virtual queue. We also present two intermediate algorithms (Algorithms 4 and 5), to explain how we arrived at DIBA, algorithm 6.

Our initial approach was to append requests at the end of the queue, by saving the end time of the queue (Algorithm 4) (end time of the queue is when we expect the last client of the queue to return to the regulator). This approach struggles when the desired return rate increases, either due to a real increase or an initially too low estimate of the task completion rate. The effect is that the return rate is too low until all clients that were queued before a rate increase have been processed.

Algorithm 4: Calculates client return time by appending the client to the end of the virtual queue

```
   Data: endOfQueueTime
   Data: currentTime
1  function getReturnTime(desiredReturnRate)
2  begin
3      queueInterval ← 1 / desiredReturnRate
4      if endOfQueueTime − currentTime ≤ 0 then
5          endOfQueueTime ← currentTime + queueInterval
6      else
7          endOfQueueTime ← endOfQueueTime + queueInterval
8      clientWaitDuration ← endOfQueueTime − currentTime
9      return clientWaitDuration
```

Algorithm 5 attempts to fix this behavior by scheduling requests earlier than the absolute end of the queue, making it more suitable for scenarios where the task completion rate (TCR) increases. However, it instead performs poorly when the

TCR decreases, as requests will then be queued long after the previous end time of the queue, creating a gap in the queue.

Algorithm 5: Calculates client return time by dynamically inserting the client in time based on the virtual queue level

```
   Data: virtualQueueLength
1  function getReturnTime(desiredReturnRate)
2  begin
3      queueInterval ← 1 / desiredReturnRate
4      clientWaitDuration ← queueInterval * virtualQueueLength
5      return clientWaitDuration
```

Algorithm 6 is an attempt to combine the advantages of Algorithm 4 and 5, while avoiding the drawbacks. It uses the behavior of Algorithm 5 when the TCR increases (dynamic insert), but switches to the behavior of Algorithm 4 when the TCR decreases (bounded append). This way, a higher desired return rate will take effect faster, and there will not be any empty gaps in the queue due to the bounded append. The naming of the algorithm comes from the two techniques of placing request in the virtual queue, either by dynamic insert or bounded append.

Algorithm 6: Hybrid approach for calculating return time, DIBA

```
   Data: endOfQueueTime
   Data: currentTime
   Data: virtualQueueLength
1  function getReturnTime(desiredReturnRate)
2  begin
3      queueInterval ← 1 / desiredReturnRate
4      clientWaitDuration ← queueInterval * virtualQueueLength
5      if
       currentTime+clientWaitDuration−endOfQueueTime < queueInterval
       then
           /* This is the insert behavior from Algorithm 5          */
6          returnTime ← currentTime + clientWaitDuration
7          if returnTime > endOfQueueTime then
8              endOfQueueTime ← returnTime
9      else
           /* This is the append behavior from Algorithm 4          */
10         returnTime ← endOfQueueTime + queueInterval
11         endOfQueueTime ← returnTime
12     clientWaitDuration ← returnTime − currentTime
13     return clientWaitDuration
```

3.1.2 Vector Algorithm

In this section, we introduce an alternative to DIBA (Section 3.1.1), that uses multiple queues in parallel. Each queue has a different return rate, and together they make up the desired return rate. The queues scale exponentially with base 2, starting at one, meaning that each discrete task completion rate can be constructed from exactly one combination of queues. In DIBA, the return rate of the virtual queue needed time to adjust after a change of the task completion rate. With a queue composed of multiple subqueues, we can adjust the return rate in real time. However, in a case where the task completion rate decreases during an execution, the message cost will increase, as we cannot control clients in already existing queues.

Algorithm 7 keeps a set of active queues, calculated by Algorithm 8, that together make up exactly the *desiredReturnRate*. When a request is to be rescheduled, it checks which of the active queues that has the shortest end time with Algorithm 9, and places the request in that queue.

Algorithm 7: Calculate return time for client

```
   Data: queueVector - each element holds the end time for each queue
   Data: currentTime
1  function getReturnTime(desiredReturnRate) begin
2      activeQueueIndices ← getActiveQueues(desiredReturnRate)
           /* Algorithm 8                                          */
3      shortestQueueIndex ← getShortestQueueIndex(activeQueueIndices)
           /* Algorithm 9                                          */
4      if queueVector[shortestQueueIndex] ≤ currentTime then
5          returnTime ← currentTime + 1/2^shortestQueueIndex
6      else
7          returnTime ←
           queueVector[shortestQueueIndex] + 1/2^shortestQueueIndex
8      queueVector[shortestQueueIndex] ← returnTime
9      return returnTime − currentTime
```

Algorithm 8 calculates what queues to use for any given desired return rate. For example, if the desired return rate is 7 tasks per second, Algorithm 8 returns queues 0, 1 and 2 (rates $2^0 = 1$, $2^1 = 2$ and $2^2 = 4$). Since this algorithm will be called frequently with the same input, it might be a good idea to cache the return values for faster lookups, i.e., memoization.

Algorithm 9 returns the queue with the earliest end time. Figure 3.1 illustrates a virtual queue state where the second slowest queue (queue 1, rate $2^1 = 2$) would be returned.

Algorithm 8: Calculates the set of indices for the active queues for a specific return rate

Data: $queueVector$ - each element holds the end time for each queue, i.e., for the respective return rates

```
1  function getActiveQueues(desiredReturnRate)
2  begin
3      remainder ← ⌈desiredReturnRate⌉
4      index ← queueVector.length − 1
5      while index ≥ 0 do
6          quotient ← remainder / 2^index
7          if quotient ≥ 1 then
8              activeQueueIndices.add(index)
9              remainder ← remainder − 2^index
10         decrement index
11     return activeQueueIndices
```

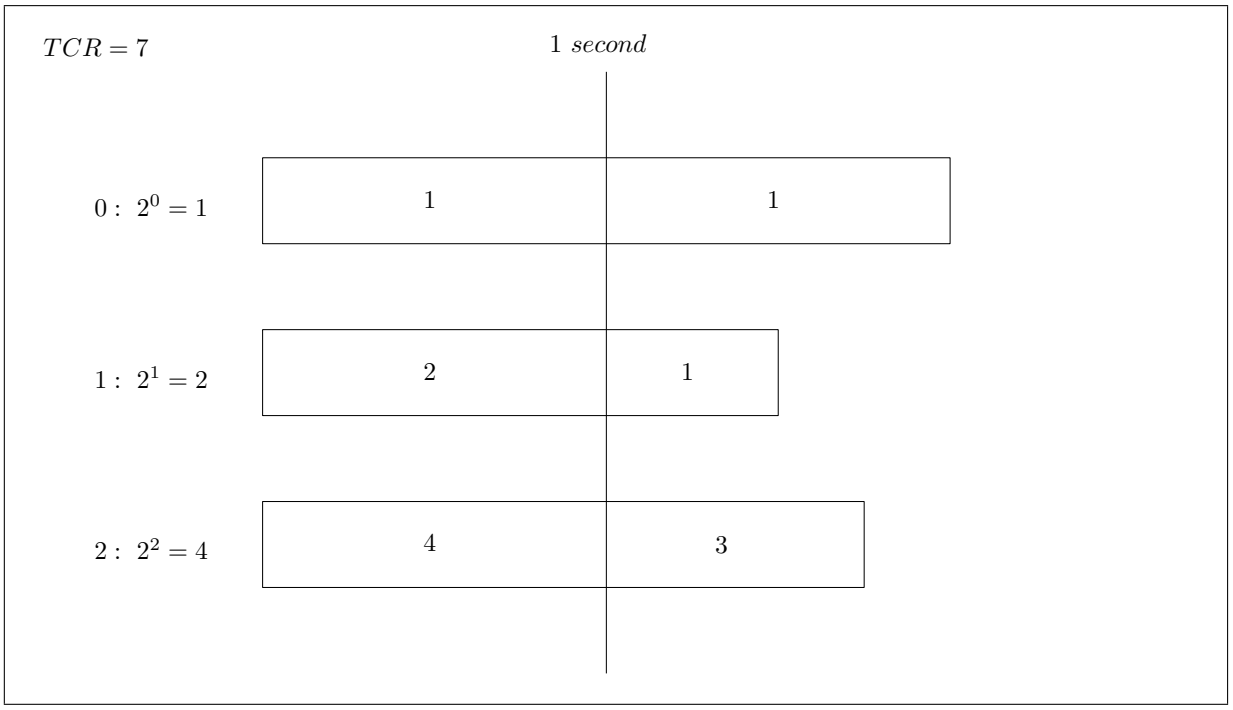


Figure 3.1: An illustration of a virtual vector queue, with 12 requests in queue. In this case, queue 1 (return rate = 2) would be returned by Algorithm 9. The number of requests in each queue is added for clarity, in practice only the end time of each queue is saved.

Algorithm 9: Finds the index of the shortest queue

```
Data: queueVector - each element holds the end time for each queue, i.e., for
      the respective return rates
1  function getShortestQueueIndex(queues)
2  begin
3      minLength ← −1
4      foreach queueIndex in queues do
5          if minLength = −1 then
6              minLength ← queueVector[queueIndex]
7              shortestQueueIndex ← queueIndex
8          else if queueVector[queueIndex] < minLength then
9              minLength ← queueVector[queueIndex]
10             shortestQueueIndex ← queueIndex
11     return shortestQueueIndex
```

3.2 Calculating the Desired Return Rate

To decide the desired return rate of the virtual queue, the regulator has to know what the task completion rate of the application server is, or at least have a way to estimate it. Algorithm 10 shows our implementation of a calculation for a desired return rate. An estimate of the task completion rate is calculated from individual job times. This estimated task completion rate (eTCR) is then scaled up with the ratio between the standard deviation of the average job time and the average job time, to account for fluctuations in task completion times, as well as being able to tolerate crashing clients. The result of the scaling is what we denote as the desired return rate. It is important that the desired return rate is larger than the real task completion rate, as the server may become underutilized otherwise. Excess requests resulting from the larger return rate are rescheduled again.

We call the rate at which requests returns from the virtual queue the return rate. The goal is to keep the return rate equal to the desired return rate. However, as the system cannot control request return times once they have been placed in the virtual queue, the return rate at any given time depends on what the desired return rate was when the requests in that part of the queue were queued. We depict the relationship between task completion rate and return rate, see Figure 3.2.

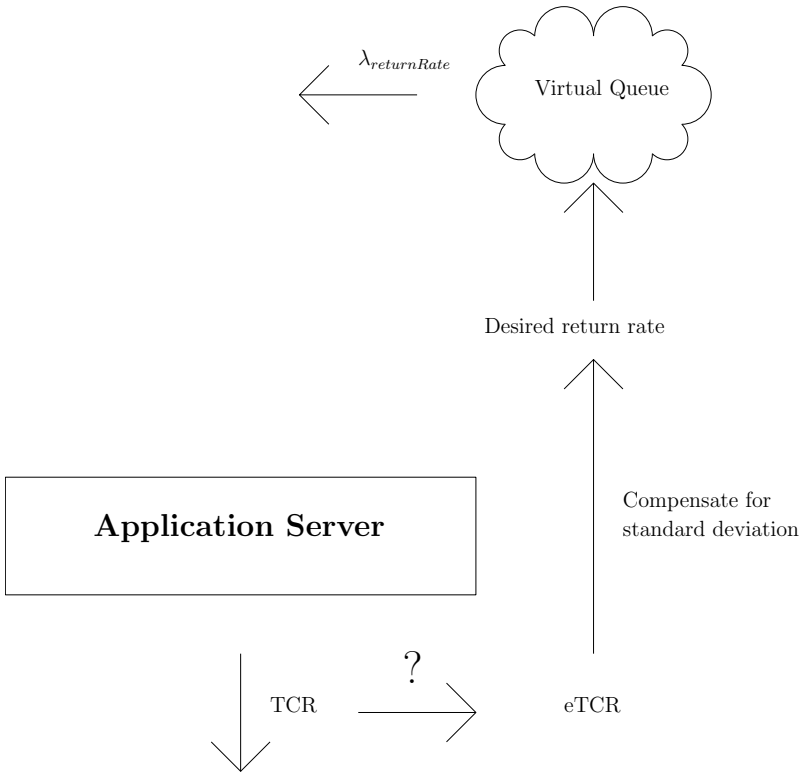


Figure 3.2: The relationship between task completion rate and virtual queue return rate

Algorithm 10: Regulator calculates desired return rate for virtual queue

Data: C_C - number of concurrent jobs at the server
Data: $numberOfServerUpdates$ - number of server updates
Data: $sumOfJobTimes$
Data: $sumOfSquaredJobTimes$

1 **function** $calculateDesiredReturnRate(arrJobTime)$ **begin**
2 $\quad sumOfJobTimes \leftarrow sumOfJobTimes + arrJobTime$
3 $\quad sumOfSquaredJobTimes \leftarrow sumOfSquaredJobTimes + arrJobTime^2$
4 $\quad$ **if** $numberOfServerUpdates > 1$ **then**
$\quad\quad$ `/* Compute standard deviation */`
5 $\quad\quad \mu \leftarrow \frac{sumOfJobTimes}{numberOfServerUpdates}$
6 $\quad\quad \sigma^2 \leftarrow \frac{sumOfSquaredJobTimes}{numberOfServerUpdates} - \mu^2$
7 $\quad\quad \sigma \leftarrow sqrt(\sigma^2)$
8 $\quad\quad overrate \leftarrow 1 + \frac{\sigma}{\mu}$
9 $\quad\quad clientFinishInterval \leftarrow \frac{\mu}{C_C}$
10 $\quad\quad desiredReturnRate \leftarrow \frac{1}{clientFinishInterval} * overrate$
11 $\quad$ **return** $desiredReturnRate$

3.3 Fairness

We define fairness in terms of request return levels, i.e., how many times a client has to contact the regulator before being allowed access to the application server. Our primary goal is to maximize throughput while avoiding congestion, therefore we consider fairness with respect to the return level rather than with the waiting time.

None of the algorithms that has yet been introduced considers fairness. We introduce fairness by extending Algorithm 3, by proposing more dynamic access guards. The general idea is that we divide the area of valid values in the backlog, that is the area between LWM and HWM, into quarters, which we control with four thresholds (Figure 3.3). The backlog quarter levels are named, from low to high:

- Free Go - All clients may enter the application server.
- Prio 3 - Only returning clients may enter the application server.
- Prio 2 - Only returning clients, who have a return level higher than the average return level in the virtual queue may enter the application server.
- Prio 1 - Only returning clients that are top prioritized may enter.

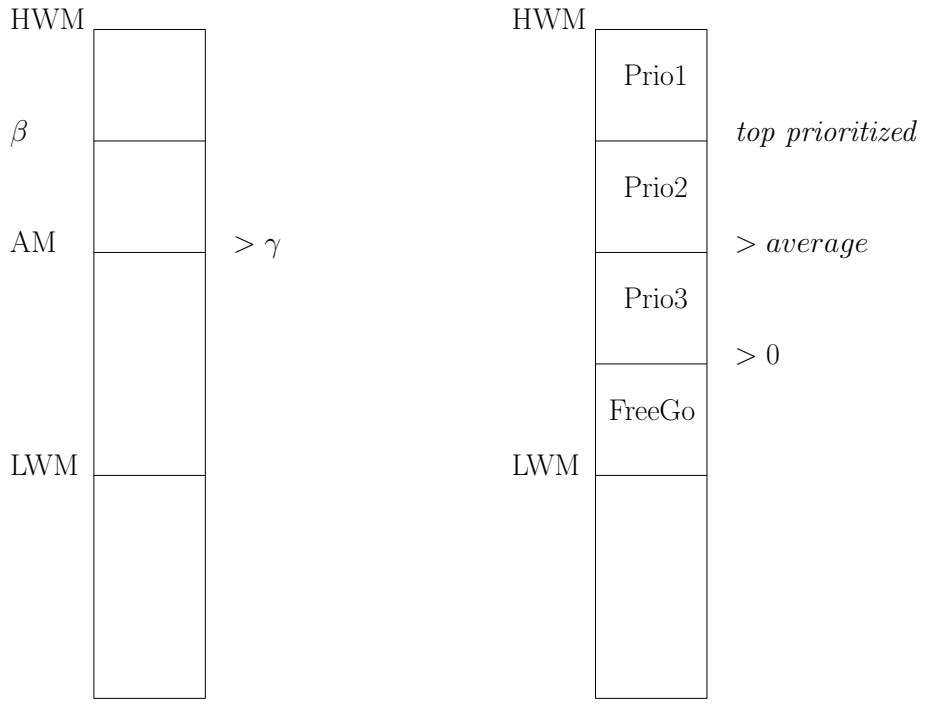


Figure 3.3: Backlog illustration for Algorithm 3 (left) and Algorithm 11 (right).

The entry conditions get more relaxed with lower backlog levels. The three lowest thresholds aim at mitigating mild return level discrepancies, by denying access to requests with low return levels in favor of requests with higher return levels. Now we

have some basic fairness that considers all clients that are below average. To further improve fairness, we now consider the clients that are above average, where we prevent that higher return levels are postponed by introducing the *Prio1* threshold.

The highest return levels are the top prioritized group. It is important that this group does not include too many return levels and remains relatively small so that we can provide space for them at the application server. There are multiple trade-offs that should be considered, where one is the size of a quarter, denoted as s.

The top prioritized group is defined as the highest return levels which together does not surpass s. For example, if s is 50 and the return levels distribution in the virtual queue is {[1, 100], [2, 10], [3, 10]}, where [$returnlevel$, $numbersofrequests$], then the return levels 2 and 3 would be top prioritized. For a distribution {[1, 100], [2, 50], [3, 10]}, only return level 3 would be top prioritized. The function call $isTopPrioritized(numberOfTries)$ returns *true* if a request is top prioritized, and $false$ otherwise.

The necessary extensions that needs to be applied on Algorithm 3 lines 4 to 9, are described in Algorithm 11 and here in this paragraph. Counters like the $virtualQueueLength$ (line 3 and 7 in Algorithm 3) and updates for maps/arrays that are needed to support the logic are omitted in Algorithm 11. Furthermore, information needs to be kept about how many requests of each return level there are in the queue, to calculate the average return level. We also assume the presence of a helper function $isOverAverage(numberOfTries)$, which return *true* if the return level of the current request is above the average return level of the virtual queue, and $false$ otherwise.

Algorithm 11: Regulator with fairness, dynamic gate logic. Extension of Algorithm 1.

Data: as in Algorithm 1
Data: $freeGo \leftarrow LWM + \frac{(HWM-LWM)}{4}$
Data: $prio3 \leftarrow LWM + \frac{(HWM-LWM)}{2}$
Data: $prio2 \leftarrow LWM + \frac{3 \cdot (HWM-LWM)}{4}$
Data: $prio1 \leftarrow HWM$
1 **upon arrival** $\langle Request, numbOfTries \rangle$ **from** Client **do begin**
2 $accessService \leftarrow false$
3 **if** $backlogLevel < freeGo$ **then**
4 $accessService \leftarrow true$
5 **else if** $backlogLevel < prio3$ ***AND*** $numberOfTries > 0$ **then**
6 $accessService \leftarrow true$
7 **else if** $backlogLevel < prio2$ ***AND*** $isOverAverage(numberOfTries)$ **then**
8 $accessService \leftarrow true$
9 **else if** $backlogLevel < prio1$ ***AND*** $isTopPrioritized(numberOfTries)$ **then**
10 $accessService \leftarrow true$
11 **if** $accessService$ **then**
12 **send** (Access Service, token) **to** Client
13 **else**
14 $returnTime \leftarrow getReturnTime(desiredReturnRate)$
15 **send** (Wait, $returnTime$) **to** Client

4

Analysis

In this chapter, we start by analyzing the behavior of the algorithms controlling the return rate of the virtual queue, i.e., DIBA and Vector. After stating the limitations of the return rate algorithms, we continue by showing that both approaches lack fairness. Lastly, we provide an initial analysis showing that our fairness approach improves fairness.

4.1 General

We want to remind the reader of some settings that have been previously stated in Chapters 2 and 3. Tasks on the application server are initiated by requests from remote clients. The application server has a limit on how many tasks can be processed simultaneously. At any given time, the utilization refers to the number of tasks currently being processed. We define full utilization as the state in which the server has reached the (predefined) limit of concurrent tasks.

We can assume that that the system call to accept requests from the backlog is significantly faster than the task completion time, so that as soon as a task finishes, the server accepts a new task from the (non-empty) backlog.

To simplify argumentation, we also assume that the server comes with an unbounded and unrestricted backlog where excess requests can be stored until they can be processed by the server, i.e., no requests are discarded. This way we do not have to consider the regulator nor the virtual queue for Lemma 1 and Corollary 1.

In order to argue that any algorithm can keep the server at full utilization, we first state the conditions for this to be possible. We have illustrated the different kinds of rates that exist in the system to help differentiate between these (Figure 4.1).

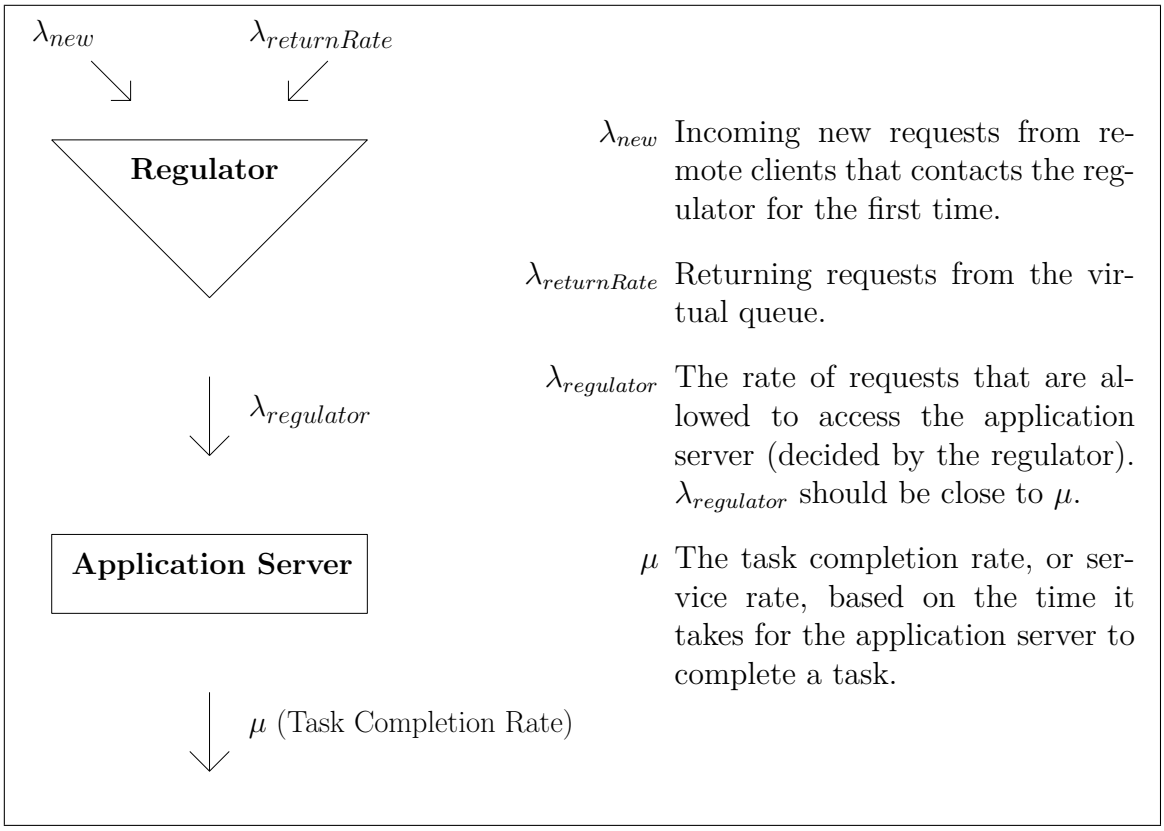


Figure 4.1: Illustration of the different rates.

To keep full utilization, or more specificly a certain backlog queue level, we need:

$$\lambda_{regulator} \approx \mu$$

This means that λ_{new} and $\lambda_{returnRate}$ have to be at least equal $\lambda_{returnRate}$:

$$\lambda_{new} + \lambda_{returnRate} \geq \lambda_{regulator}$$

We start by not considering the regulator, and not making any assumption about the return rate, $\lambda_{returnRate}$, from the virtual queue prior this state. Here, the number of incoming requests has to be greater than the number of completed tasks, i.e., λ_{new} has to be on average greater than or equal to μ. An example is if 100 tasks have been completed, at least 100 new requests have had to come in during the same period. That reasoning gives us Lemma 1.

Lemma 1. *Given a time period from t_0 to t_{end}, where the total number of new arrivals, λ_{new}, is at least as large as the total number of completed tasks for all times t in the entire period counted from t_0, there are enough tasks to maintain full utilization.*

Proof. In order to be able to give full service, the average input has to be greater than the average output for all points in time. This means that the average arrival rate, counted from starting point t_0, always has to be greater than the average task completion rate counted from the same time. □

As long as the system is in a state with full utilization, the task completion rate is equal to the service rate. From here on, we will only talk about task completion rate.

From the system properties, we know that the accept call from the backlog is much smaller than the task completion time, making it insignificant. This allow us to draw Corollary 1.

Corollary 1. *As long as the backlog is non-empty, we can guarantee full utilization.*

We now start to consider the regulator and want to prove one of the key concepts, that if the return rate from the virtual queue is equal or greater than the task completion rate, full utilization can be maintained. Furthermore, by considering the regulator we can also limit the backlog to hold a maximum number of requests, by placing any excess requests being placed in the virtual queue. For the sake of presentation simplicity, we assume that the regulator knows the real task completion rate, and requests are scheduled in the virtual queue according to that rate. If the return rate from the virtual queue is higher than the task completion rate, any excess clients are simply rescheduled. This givers us Corollary 2.

Corollary 2. *Given a server state with full utilization and a return rate of the virtual queue that is higher than or equal to the task completion rate, full utilization is guaranteed.*

4.2 DIBA - Dynamic Insert, Bounded Append

Suppose that the system is in a state with full server utilization. Moreover, suppose that all clients in the virtual queue are queued with return rate equal to the task completion rate. Then, we say that the system is in *Normal state*.

4.2.1 Constant Task Completion Rate

If the system has a known constant task completion rate, the regulator will queue tasks in the virtual queue on at least that rate. That is, the only time the return rate is lower than the task completion rate is when the virtual queue is empty (Corollary 3), and this only happens if there are no requests to put in the queue.

Corollary 3. *Given a known constant task completion rate, the only time the return rate from the virtual queue is smaller than the task completion rate is when the virtual queue is empty.*

Theorem 1 prove that, as long as there are enough client requests, full utilization can be maintained.

Theorem 1. *Given Normal state at time t_0, a non-empty backlog, and a constant task completion rate, the regulator will keep the server at full utilization given that there are enough incoming requests to maintain full utilization (Lemma 1).*

Proof. Assume that, at time t, the server is not fully utilized. Corollary 2 states that for this to occur, the backlog has to be empty, which means that the arrival rate and the return rate from the virtual queue together would have been lower than the task completion rate. For this to happen, the virtual queue has to be empty (Corollary 3) and the arrival rate has to be smaller than the task completion rate, which contradicts the conditions given by Lemma 1. Contradiction! □

4.2.2 Task Completion Rate Changes

Suppose that the task completion rate change with a factor $\epsilon > 0$ and a backlog queue level is AM, we can maintain full utilization (Theorem 2). To understand the theorem we first have to declare some terms in a general system setup:

Let μ be the system task completion rate $[\frac{tasks}{time\ unit}]$, V be the number of tasks in the virtual queue, and $\epsilon \cdot \mu$ be the task completion rate after the change.

In the case where the task completion rate decreases, the algorithm will append new jobs at the end of the queue. For the case where the task completion rate increases we conclude that:

1. The total queue length in time at the time of the task completion rate change is $V \cdot \mu^{-1} = \frac{V}{\mu}$.

2. The task completion rate change will have effect on the queue at time $V \cdot (\epsilon \cdot \mu)^{-1} = \frac{V}{\epsilon \cdot \mu}$.

3. Until that time, $(\epsilon \cdot \mu) \cdot \frac{V}{\epsilon \cdot \mu} = V$ tasks will be finished. We call to this the *output*.

4. The *input* to the backlog from the virtual queue before the change has taken effect is $(\mu) \cdot \frac{V}{\epsilon \cdot \mu} = \frac{V}{\epsilon}$ tasks.

When there is an increase in task completion rate, the returning requests will initially come back too slow, and to prevent the backlog from emptying, we have AM clients in the backlog plus *input* from the virtual queue, i.e. returning clients. This gives us Corollary 4.

Corollary 4. *From Corollary 1 we can conclude that the system will remain fully utilized as long as: output* $< AM +$ *input.*

The properties for how DIBA works adds some restrictions on how big rate increases the regulator can cope with. Theorem 2 shows that DIBA can maintain full utilization given that limitation.

Theorem 2. *Given Normal state, a backlog with AM requests, and an increase in task completion rate, i.e.,* $\epsilon > 1$. *As long as* $\epsilon < \frac{V}{V - AM}$, *or if* $AM > V$, *the regulator will keep the server at full utilization as long as there are enough requests to maintain full utilization (Lemma 1).*

Proof. From Theorem 1, we conclude that full utilization will be maintained from time $\frac{V}{\epsilon \cdot \mu}$, as long as the backlog does not empty before that time. This is due to

the fact that after time $\frac{V}{\epsilon \cdot \mu}$, the return rate from the virtual queue will be greater than or equal to the task completion rate (Corollary 2). Therefore, we need to show under which conditions the backlog does not empty before time $\frac{V}{\epsilon \cdot \mu}$. From Corollary 4 we derive that when $AM > output$, i.e. $AM > V$, the property will be fulfilled no matter what the input is, that is, any $\epsilon > 1$ fulfills the criteria. This also applies in the case where $AM = output$, as the number of available requests is at least as large as the number of completed tasks. In the case where $AM < output$, we have a greater restriction on ϵ:

$$output < AM + input$$

$$0 < AM + input - output$$

$$0 < AM + \frac{V}{\epsilon} - V$$

$$(V - AM) < \frac{V}{\epsilon}$$

$$\epsilon \cdot (V - AM) < V$$

$$\epsilon < \frac{V}{V - AM}$$

□

This means that the more requests there are in the virtual queue, the smaller any change to the task completion rate can be before the system becomes underutilized.

Given a previously constant task completion rate, from which the task completion rate decreases, the algorithm will append new jobs at the end of the virtual queue at the new desired return rate, i.e., the new task completion rate. For the time period before this change takes place, the return rate is higher than the new rate, i.e., any excess requests will be rescheduled at the new desired return rate.

That is, for any decrease $0 < \epsilon < 1$ full utilization will be maintained (Theorem 3). The case where $\epsilon = 0$ would mean that the application server has stopped processing tasks, and this case is not interesting for us.

Theorem 3. *Given Normal state with full utilization and a decrease in task completion rate, i.e.,* $0 < \epsilon < 1$*, full utilization will be maintained for any rate* $\epsilon \cdot \mu$*.*

Proof. Before a task completion rate decrease has effect on the return rate, the tasks will come in too fast and the server will maintain full utilization. □

Convergence time

We define convergence time as the time until we are able to do another change with factor ϵ, or how long time it takes to get back to *Normal state*.

After the task completion rate increases with a factor ϵ, the system will not return to *Normal state* until all previously queued clients in the virtual queue have returned, i.e., $\frac{V}{\mu}$ (item 1 in the list above).

4.3 Vector Algorithm

In order to achieve a specific return rate from the virtual queue, the vector algorithm activates multiple queues to construct the correct return rate. For any task completion rate, μ, the number of queues necessary to construct the corresponding return rate we denote as y.

Corollary 5. *To construct any distinct return rate, there has to be at least one request queued in each active queue (a minimum of y), i.e.,* $\min_{i \in Q} V_i > 0$*, where Q is the set of active queues and V_i the length of queue i.*

It is not possible to create a specific return rate if there are less than y requests to queue (Corollary 5). This renders a corner case which we describe in Lemma 2.

Lemma 2. *Given a state with full utilization, constant task completion rate, and that it is possible to maintain full utilization (Lemma 1), if the virtual queue never reaches a a state where* $\min_{i \in Q} V_i > 0$*, full utilization is maintained, given a backlog level* $> y - 1$*.*

Proof sketch. Assume that the backlog can fluctuate more than $y - 1$. All active queues never gets filled, therefore the return rate will be to slow. Assume that the desired return rate is μ_y, but as we do not reach level y, our rate will be slower. The return rate μ_y would render μ_y returning clients during a period t, for the same period all requests ($< y$) from the lower rate would also have had returned. Note that we make no assumption on the distribution of the returning clients, only that they return before the time period t ends. For the initial assumption to be false, the returning clients would have to return after the period t, or this proof contradicts Lemma 1. Contradiction! □

4.3.1 Constant Task Completion Rate

For Vector, the virtual queue has to fill up all active queues before having the intended rate on the returning clients. As this value is relatively small, we choose to start our proof from that state (Theorem 4).

Theorem 4. *Given a starting state where* $\min_{i \in Q} V_i > 0$*, a constant task completion rate, and the conditions given by Lemma 1, full utilization will be maintained.*

Proof sketch. At least one active client in each active queue means that our return rate is the same as the task completion rate. Under these conditions, full utilization is maintained (Corollary 2). Assume that there is a way to become underutilized. The only way to get fewer clients in the virtual queue would be if the average task completion rate is higher than the average request arrival rate, but that would violate Lemma 1. Contradiction! □

4.3.2 Task Completion Rate Changes

When the desired return rate is changed from μ, to $\epsilon \cdot \mu$, the algorithm might open or close queues. Closed (inactive) queues will add to the return rate until they empty.

Queues that are reused will be harmonized with new queues by only adding new requests to the shortest queue.

The task completion rate change can be either an increase or a decrease. For both cases, we need the virtual queue to stabilize in a state where there is at least 1 client in each active queue to provide at least the desired return rate $\epsilon \cdot \mu$. Any old queues (inactive) that have not emptied yet, will add to the return rate, but this will not prevent us from maintaining full utilization (Corollary 2).

Given that it is possible to maintain full utilization (Lemma 1) from the point where a rate change is started, there are two cases. Either requests come in too fast, or they come in at the same rate as they finish (on average).

In the case where the requests come in at the same rate as they finish, no queue will be built, but full utilization will still be maintained (Lemma 2).

Corollary 6. *Given full utilization, and that the virtual queue return rate equals the application server task completion rate, full utilization is maintained.*

In the case where requests come in too fast, excess requests will be placed in the active queues for the new rate. Given the worst case where the new active queues are all empty, it would need to accumulate at least y requests to ensure the desired return rate (Corollary 7).

Corollary 7. *Given a virtual queue level $\geq y$ we can guarantee full utilization (Corollary 2).*

For the case where the virtual queue level is less than y the return rate from the virtual queue level will be too low to achieve the desired return rate. Therefore, it can result in fluctuations of the backlog level similar to those we saw in Lemma 2. Hence, we need to be able to tolerate fluctuations in the order of $y - 1$. That is, as long as the backlog can handle the fluctuations, we can guarantee full utilization.

Theorem 5. *Given a state with full utilization, a change in task completion rate, and that it is possible to maintain full utilization (Lemma 1), full utilization is maintained given a backlog level $> y - 1$.*

Proof sketch. Given by Lemma 2 and Corollaries 6 and 7. □

Convergence time

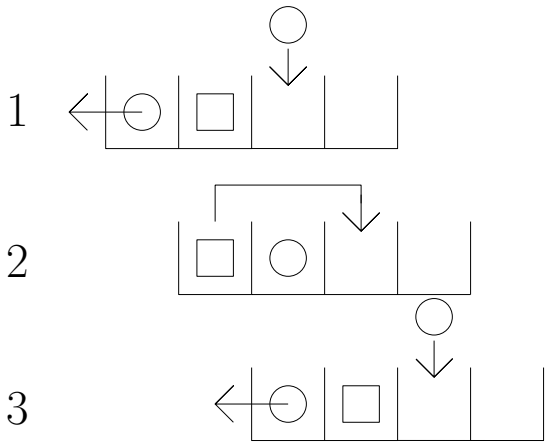
For the Vector algorithm there is immediate convergence, as a change of the desired return rate would result in new queues being opened immediately. That is, the new desired return rate will be achieved instantaneously (excluded special case numbers of clients in the virtual queue sizes $< y$).

4.4 Upper bound for Return Level

As neither DIBA nor the Vector algorithm prioritizes queued requests based on their return level, the question arises whether a request can be rescheduled *ad infinitum*, only limited by the total number of requests. We begin by showing a scenario where

this happens with an overestimation of task completion rate, and then show that the same scenario can happen due to fluctuating task completion rate.

In Figure 4.2, we show an example where the request denoted by a square is never allowed access to the application server, as long as there are new requests entering the system. In the example, the return rate from the virtual queue is twice the task completion rate of the application server, i.e., for every two requests that come back from the virtual queue, one is accepted to the application server. The reasons for having a return rate larger than the task completion rate have previously been discussed in Section 3.2.



1. A request is accepted by the server. Simultaneously, a new request arrives and is placed at the end of the virtual queue.
2. The request denoted by a square is denied access to the server, and placed at the end of the virtual queue.
3. A request is accepted by the server. Simultaneously, a new request arrives and is placed at the end of the virtual queue.

Figure 4.2: Example where repeating step 1 and 2 forever would result in the request denoted by a squared being rescheduled forever.

Lemma 3. *A request can be rescheduled every time it tries to access the application server, given a known constant task completion rate.*

Proof. Consider a server state with full utilization, a backlog at level β, and two requests in the virtual queue. Let the return rate from the virtual queue be twice that of the server task completion rate. Consider the following scenario:

1. A new request arrives. It is put at the end of the virtual queue.
2. A task on the server completes and the first request in the virtual queue accesses the server.
3. The second request in the virtual queue tries to access the server and is rescheduled.

Item 1 and 2 are interchangeable. After this scenario, the system state is identical to the initial state. Thus, by repeating the scenario above, we show that a request can be rescheduled forever.

□

We have shown that the return level of a client is only limited by the total number of requests. The same can happen with a fluctuating task completion rate instead of a overestimation of the task completion rate. Appendix A contains examples of two such scenarios.

4.5 Fairness

To prove that Algorithm 11 indeed increases fairness, we show a more efficient bound on the number of return levels. The primary mechanism for this bound is the *Prio1* threshold, or more precisely, the reserved space between this threshold and the high water mark, which we denote as s (illustrated in Figure 4.3). This reserved space means that a client with the highest priority can no longer be denied access to the server because of client with lower priority.

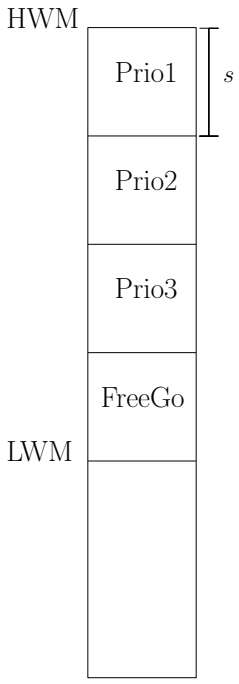


Figure 4.3: Illustration of the size of *Prio1*.

4.5.1 Constant Task Completion Rate

We start by assuming a known constant task completion rate μ, and a return rate which is the μ overestimated with a factor $\theta_0 \geq 1$. If $\theta_0 = 1$, requests arrive at the same rate as tasks are completed, which means that no requests will be rescheduled and the system would be fair. For any distinct $\theta_0 > 1$, a certain proportion of incoming requests will be rescheduled. This proportion is equal to $\theta_0 - 1$, and we need to investigate what will happen to them.

With a constant return rate, the virtual queue acts as a FIFO-queue, with any excess requests being rescheduled at the end of the queue. With this in mind, we define queue *iterations*. An *iteration* is the time period from the rescheduling of

a request until that particular request returns to the regulator the next time. An important property of an *iteration* is that the group of highest priority requests, T, at the start of an *iteration* is a subset of the group of requests, T_{old}, that had the highest priority in the previous *iteration*. Alternatively, if all members of T_{old} were given access during the previous *iteration*, T is a subset of the highest return level group that was not given access in its entirety during the previous *iteration*.

Theorem 6. *Given a constant task completion rate and a return rate which is an overestimation of the task completion rate by a factor θ_0, the highest possible return level is bounded from above by $1+\log_{1+x}(n)$, where n is the total number of requests, and $0 < x \leq \frac{1}{\theta_0 - 1}$.*

Proof sketch. We know that for every request that is given access to the server, $\theta_0 - 1$ requests are rescheduled. Because of the *iterations*, we know that in order to advance a request from T_{old} to T, $\frac{1}{\theta_0 - 1}$ requests of T_{old} needs to be given access to the server. Then, the number of new requests needed to move one request from T_{old} to T is exponential, with base $1 + x$, where x is expected to be $\frac{1}{\theta_0 - 1}$ and is guaranteed to be greater than 0, and the exponent being the return level of T_{old}. Thus, the highest possible return level is bounded by $1 + \log_{1+x}(n)$. □

Remark. *In our bound, x is a measure of the number of requests of some return level needed to postpone one other request of that return level. While we have not been able to prove a better (hard) bound than $x > 0$, this property of x stems from the fact that below the highest return level, requests can be postponed by clients of both higher and lower return level. However, as the difference in return level is controlled by both the $Prio2$ and the $Prio3$ thresholds, x should be significantly larger than 0.*

In addition to the above properties, at least s clients of T will be given access during each iteration, where s is the size of the reserved space in the backlog, as long as the the size of the virtual queue is large enough to let the backlog level sink to the $Prio1$ threshold. After the backlog level reaches HWM, there is a period until the level sinks back to $Prio1$, under which no requests can be given access, as T_{new} is the new top prioritized group. This behavior is dependent on the size of s, and affects our value x, $x > 0$.

The conditions we stated in this section refer to the worst case, in a more realistic scenario top prioritized requests would also enter the backlog at other levels than *Prio 1*, and the server would finish some tasks while new requests are coming in, both would decrease the number of iterations further.

4.5.2 Task Completion Rate Changes

In the presence of changes in the task completion rate, and thus in the return rate, the scenario may differ somewhat from the constant case. In the case of an isolated task completion rate change, the analysis for the constant case still applies, as the virtual queue can once again be viewed as a FIFO queue after the change has taken place.

If the task completion rate continues to fluctuate, the return rate may be too fast. In this section, we will only consider the Vector algorithm, as it can open new

queues from any time. The bound for DIBA will be at least as good, as this behavior is less prominent in DIBA.

In the presence of continuous fluctuations, the return rate may be faster than the task completion rate by a factor $\theta = \theta_0 * \frac{RR_{total}}{RR_{active}}$, where RR_{total} is the total return rate from the virtual queue, and RR_{active} is the return rate from the active subqueues of the virtual queue (any passive queues will add to the return rate until they run empty). The overestimation factor θ will then be somewhere between θ_{low} and θ_{high}, where θ_{low} is the lowest overestimation factor during the entire execution, and θ_{high} is the highest. Theorem 7 generalizes Theorem 6.

Theorem 7. *Given a task completion rate and a return rate that is faster than the task completion rate by a factor θ, $\theta_{low} \leq \theta \leq \theta_{high}$, the highest possible return rate is bounded from above by $1 + log_{1+x}(n)$, where n is the total number of requests, and $0 < x \leq \frac{1}{\theta - 1}$.*

Proof sketch. This follows from Theorem 6, but the constant overestimation is replaced with the ratio between the return rate of the virtual queue and the task completion rate of the server. □

Figure 4.4 shows an example of how requests of the highest return level can be postponed. Figure 4.5 shows an example of the cost tree for postponing a single request from the highest return level.

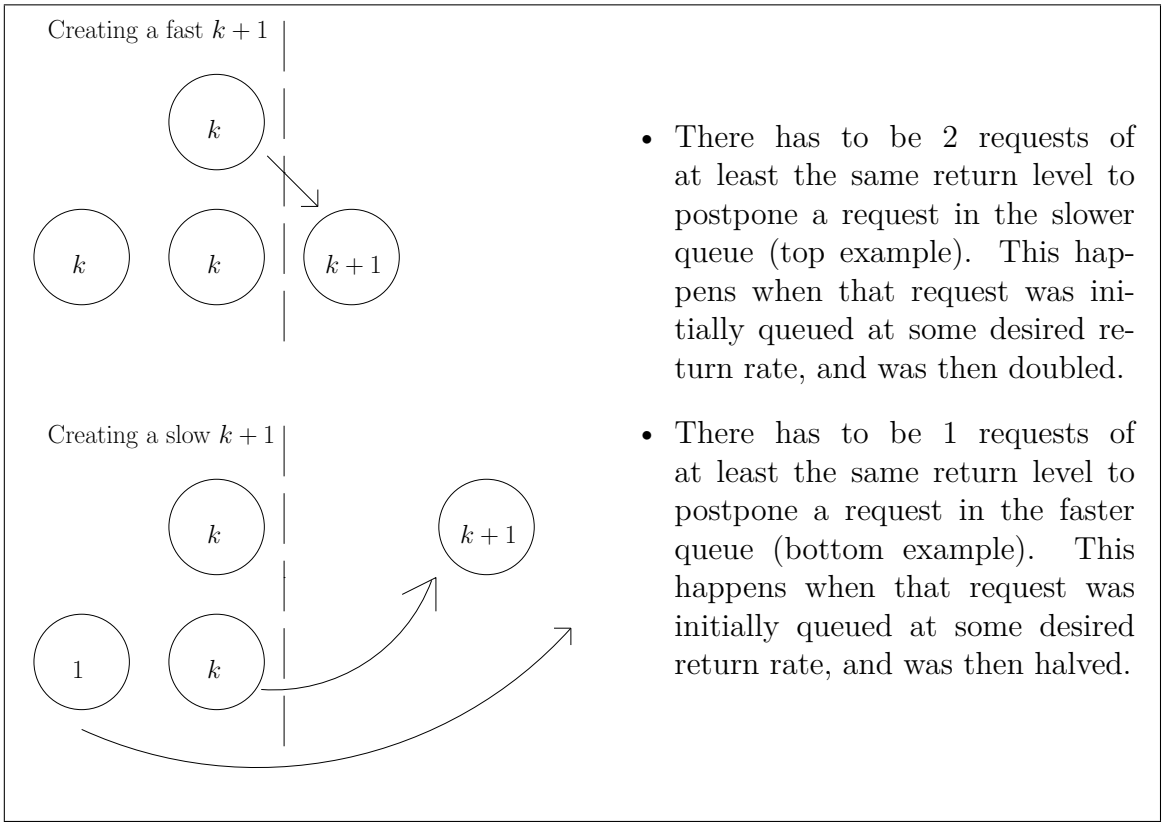
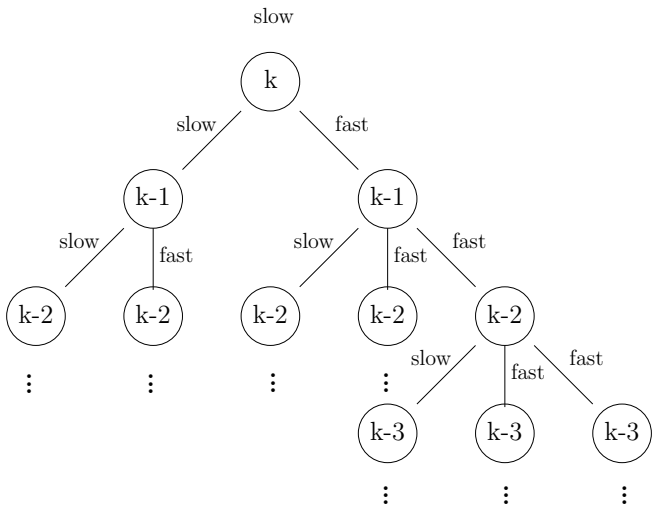


Figure 4.4: Illustration of a specific request with a return level of k can be postponed.



- Each slow request renders two request, one slow and one fast, that are necessary to postpone it.
- Each fast request renders three request, one slow and two fast, that are necessary to postpone it.

Figure 4.5: Illustration of the toal cost of postponing a specific request with a return level of k.

5
Evaluation

In this chapter, we explain our experimental evaluation process. The evaluation consists of an implementation of the regulator algorithms against a real world distributed system. Section 5.1 describes the scenarios we test our algorithms against. Furthermore, Section 5.2 describes the practical details of our setup. Finally, Section 5.3 shows the outcome of our experiments, along with some reflections about the results.

5.1 Evaluation scenarios

Some general settings will be the same for all tests (Table 5.1), for conveniency and because it will be easier to compare results from the different tests.

Table 5.1: General test settings

LWM	100
AM	200
HWM	300
c_c	100
γ	0
β	250

Constant flow

Test 1 consists of a constant flow of incoming clients, 20 requests per second for 430 seconds, totaling 8600 requests, and is supposed to work as a gentle test with low level of requests coming simultaneously.

Initial burst

Test 2 consists of an initial burst of 600 requests, followed by 100 requests each second for 80 seconds. The reasoning behind the test is that the initial burst quickly loads the system and pushes it to full utilization, while the continuously arriving requests allow us to test the properties of our queue algorithms and our task completion estimation algorithm when there are large volumes of requests coming in continuously.

Constant flow, that is followed by a large burst

Test 3 consists of a constant flow of request for a short period, 20 requests per second for 100 seconds, followed by a large burst of 6600 requests. The idea is to let the regulator stabilize under the constant flow of requests (estimated task completion rate et cetera), then see how it reacts to a large burst of requests.

5.2 Experiment setup

We have implemented and deployed the algorithms in Chapter 3 on virtual machines in the Microsoft Azure[1] cloud environment to evaluate them experimentally. Our setup consists of three separate machines, one for the application server, one for the regulator and one for the clients.

5.2.1 Application Server

The application server is deployed on a virtual machine, using Microsoft Azure's DS3 standard[2], with CentOS 6.5[3] as its operating system. To be able to use the backlog in the way we intended, we had to increase SOMAXCONN, and for the server to handle multiple connections we also had to increase the maximum number of open file descriptors.

As application server software we use Orchestra[4], which is developed by Qmatic. The application server software is mainly written in Java and uses JBoss[5] or WildFly[6] as its underlying web server. We use Wildfly, as the JBoss version requires licensing. The WebSocket communication is handled by Netty[7].

Aside from the original functionality of the application server software we added communication with the regulator and limited the number of concurrent connections in Netty. The application server sends the number of active tasks and the task completion time whenever a task finishes, as long as the server was fully utilized. It also sends information periodically every 5 second about the system state, to provide additional data points (mostly interesting in the initial or last phase of a test). The limit on the number of concurrent connections is implemented by only calling *accept*() on a chosen number of connections, c_{chosen}, in the source files for Netty. This prevents client requests from being passed into the application layer, i.e., they will remain in the backlog, until there is room for them in the application.

5.2.2 Clients

The remote clients are deployed on a Microsoft Azure's DS2 standard, again running CentOS 6.5. To run a large number on concurrent connections, i.e., allowing us to

[1] https://azure.microsoft.com
[2] https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-size-specs
[3] https://www.centos.org
[4] http://www.qmatic.com/products/software/orchestra
[5] http://www.jboss.org
[6] http://wildfly.org
[7] http://netty.io

use a single machine to simulate a large number of clients, we had to increase the maximum number of open file descriptors.

Clients are simulated using Gatling[8], a scalable and asynchronous stress-testing tool. To be able to do checks on binary WebSocket frames, we used a patched version of Gatling[9].

We have created a custom Gatling scenario, written in Scala, implementing the client behavior described in Appendix **??**. Gatling's DSL[10] is then used to configure the simulation setup (number of clients, arrival distribution et cetera).

5.2.3 Regulator

The regulator is deployed on a Microsoft Azure DS2 standard, running CentOS 6.5. To run a large number on concurrent connections, i.e., allowing us to use a single machine to simulate a large number of clients, we had to increase the maximum number of open file descriptors.

The regulator is a RESTful service based on the RESTexpress[11] framework. The regulator is implemented in Java, with a configuration file enabling easy configuration of the regulator (queue algorithm, fairness, backlog thresholds et cetera).

5.3 Test results

Here we present the experimental results for the different test setups specified in Section 5.1. Task completion time average and standard deviation are excluded in this section, as well as the desired return rates that are built from these those values, but are provided in Appendix B. The task completion rate of the system for each test is also provided in the appendix.

By comparing the desired return rate for each test with the task completion rate it is possible to say some things about what results we can expect, for example the average return level. Some tests had a relatively big standard deviation compared to the average task completion rate, which resulted in a higher desired return rate compared to the task completion rate. In some of the tests, the desired return rate continues to increase due to these properties, which should not be necessary as the real task completion rate stays relatively constant throughout the run.

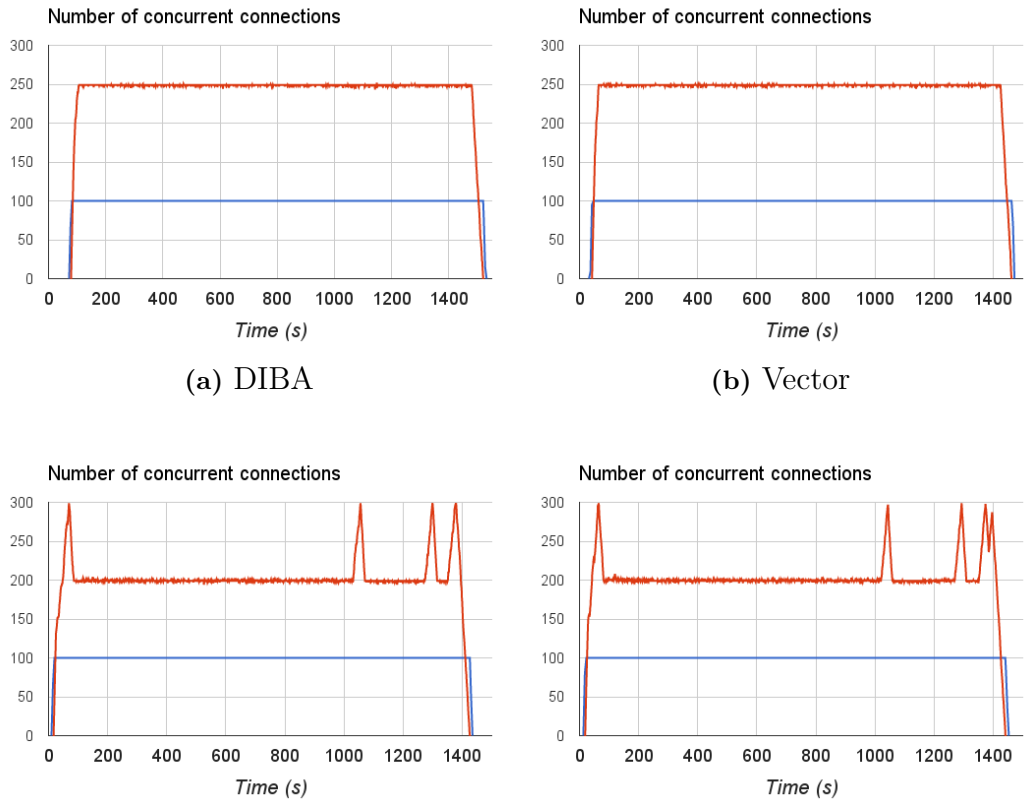
5.3.1 Utilization

The utilization graphs shows the number of concurrent working tasks, c, as well as the backlog level. When fully utilized, the number of concurrent tasks is expected to be at our chosen level c (c_{chosen}). Additionally, the backlog level has to be greater than zero. If the regulator queues requests in the virtual queue at a return rate equal to the task completion rate, the expected backlog level is AM, but as was discussed in Section 3.2, an overestimate is desired and likely when we take standard deviation

[8] https://gatling.io
[9] https://github.com/jbank/gatling
[10] Domain Specific Language
[11] https://github.com/RestExpress/RestExpress

into consideration. The expected backlog level for a higher return rate is β. For the fairness algorithms, the backlog is expected to fluctuate between *Prio 2* and *Prio 3* (~200 connections in the backlog), with peaks to HWM whenever a group of the top prioritized requests attempts to access the server.

Figures 5.1, 5.2 and 5.3 shows that our algorithms maintains full utilization for all our test scenarios, which validates our theorems. This is because there are always requests in the backlog (red line), and therefore the number of active tasks (blue line) stays fixed. Note how each backlog level peak (red line) in the fairness plots corresponds to one of the return levels that can be observed in the corresponding message cost tables in Section 5.3.2.



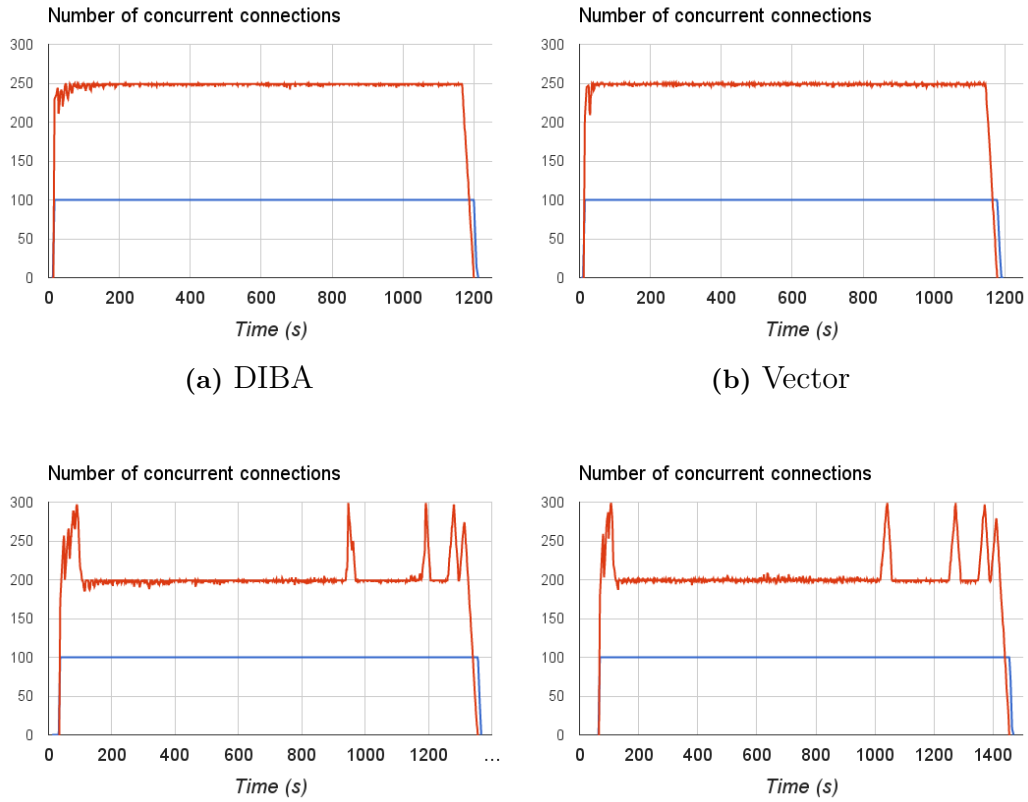
(a) DIBA

(b) Vector

(c) Fairness-DIBA

(d) Fairness-Vector

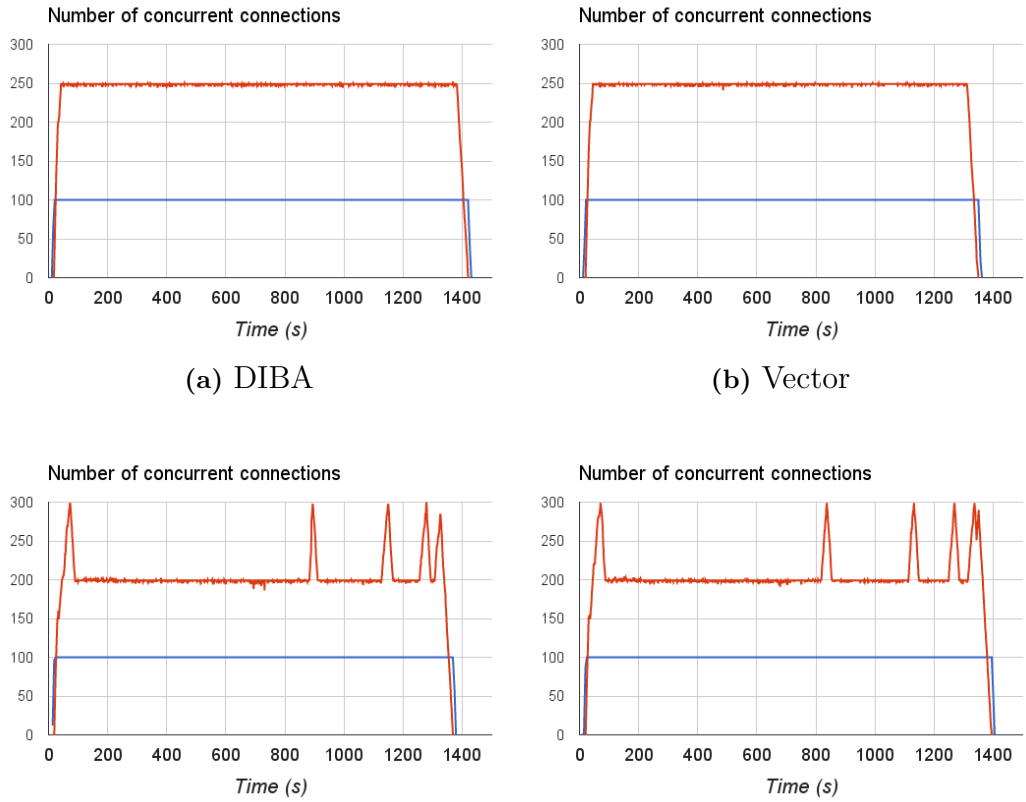
Figure 5.1: Constant flow: Red line represent the backlog level, the blue line is the level of active clients.



(a) DIBA **(b)** Vector

(c) Fariness-DIBA **(d)** Fairness-Vector

Figure 5.2: Initial burst: Red line represent the backlog level, the blue line is the level of active clients.



(a) DIBA **(b)** Vector

(c) Fairness-DIBA **(d)** Fairness-Vector

Figure 5.3: Constant flow, that is followed a large burst: Red line represent the backlog level, the blue line is the level of active clients.

5.3.2 Message Cost

Tables 5.2, 5.3 and 5.4 shows the return level distributions for the different tests, which we here refer to as the message cost. Note that the average message cost will vary slightly between different executions of the same test. We can see that the Vector algorithm has a consistently higher message cost than the DIBA algorithm. We can also see that without the fairness extension, each algorithm has an exponential tail in the return level distribution. This tail is effectively cut with the fairness extension.

The average message cost correlates directly with the ratio between the return rate and the task completion rate. An average message cost of 1 would mean that the return rate from the virtual queue is just enough to match the task completion rate, which is too low (due to crashing clients and fluctuations in task completion rate etc, as discussed in Section 3.2). All our tests maintained an average message cost greater than 1, while not exceeding 2. At the same time, a low average message cost means low traffic intensity, which indicates a low risk of congestion.

Table 5.2: Message cost for *Constant flow*

	DIBA	Vector	Fairness-DIBA	Fairness-Vector
AVG	1.59	1.60	1.57	1.64
0	386	400	300	290
1	4886	4876	4241	3998
2	2011	1959	3233	3341
3	832	842	538	602
4	294	307	281	247
5	109	131	7	122
6	51	42	0	0
7	16	26	0	0
8	10	12	0	0
9+	5	5	0	0

Table 5.3: Message cost for *Initial burst*

	DIBA	Vector	Fairness-DIBA	Fairness-Vector
AVG	1.62	1.92	1.49	1.7
0	300	300	250	250
1	5045	4202	5637	3853
2	1857	2033	1591	3409
3	837	1010	689	616
4	325	522	246	272
5	137	274	187	200
6	57	135	0	0
7	27	61	0	0
8	8	32	0	0
9+	7	31	0	0

Table 5.4: Message cost for *Constant flow, that is followed by a large burst*

	DIBA	Vector	Fairness-DIBA	Fairness-Vector
AVG	1.57	1.76	1.58	1.68
0	377	398	299	294
1	5058	4723	5116	4753
2	1871	1766	1822	2050
3	775	806	813	789
4	309	461	343	385
5	125	238	207	251
6	55	118	0	78
7	17	56	0	0
8	9	15	0	0
9+	4	19	0	0

6
Conclusion

We have developed a system to manage the flow of incoming client connections in a distributed system, in order to maintain an even workload on an application server. The system is based on an external component, the regulator, which monitors server load and notifies clients of when they should connect. For the regulator, we have developed two different scheduling algorithms. We have also developed an extension to improve the fairness of our two scheduling algorithms. We have made an experimental evaluation of our preliminary analysis by implementing the algorithms against a real world distributed system. The experimental results show good performance under static conditions. Due to our preliminary analysis, we believe our algorithms will work well under some dynamic scenarios as well.

Our system deals with scheduling in a system where the scheduler has no direct control over tasks. Instead of starting jobs when the server has free resources, our scheduler tries to estimate when there will be free resources on the server, and tells clients that they should try to access the server at that time. This means that the there is potentially a long period between the allocation of a time slot to a client to the arrival of a request from that client to the server.

Bibliography

[1] I. Adan and J. Resing. *Queueing Theory: Ivo Adan and Jacques Resing*. Eindhoven University of Technology. Department of Mathematics and Computing Science, 2001. URL: https://books.google.se/books?id=dAViMwEACAAJ.

[2] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. *Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model*. Ed. by Yoram Moses. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 341–355. ISBN: 978-3-662-48653-5. DOI: 10.1007/978-3-662-48653-5_23. URL: http://dx.doi.org/10.1007/978-3-662-48653-5_23.

[3] Michael A. Bender et al. "Adversarial Contention Resolution for Simple Channels". In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: ACM, 2005, pp. 325–332. ISBN: 1-58113-986-1. DOI: 10.1145/1073970.1074023. URL: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1073970.1074023.

[4] Michael A. Bender et al. "How to Scale Exponential Backoff: Constant Throughput, Polylog Access Attempts, and Robustness". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '16. Arlington, Virginia: SIAM, 2016, pp. 636–654. ISBN: 978-1-611974-33-1. URL: http://dl.acm.org/citation.cfm?id=2884435.2884482.

[5] D. Cederman et al. "A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems". In: *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 2013, pp. 1309–1320. DOI: 10.1109/IPDPS.2013.91.

[6] U. Kanade. "Performance of work conserving schedulers and scheduling of some synchronous dataflow graphs". In: *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*. 2004, pp. 521–529. DOI: 10.1109/ICPADS.2004.1316134.

[7] Dan Kegel. *The C10K problem*. 2014. URL: http://www.kegel.com/c10k.html (visited on 06/18/2016).

[8] Chao Li et al. "Chameleon: Adapting Throughput Server to Time-varying Green Power Budget Using Online Learning". In: *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. ISLPED '13. Beijing, China: IEEE Press, 2013, pp. 100–105. ISBN: 978-1-4799-1235-3. URL: http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=2648668.2648693.

[9] Mihai Rotaru. *How MigratoryData solved the C10M problem: 10 Million Concurrent Connections on a Single Commodity Server*. 2015. URL: https://mrotaru.wordpress.com/2015/05/20/how-migratorydata-solved-the-c10m-problem-10-million-concurrent-connections-on-a-single-commodity-server/ (visited on 06/18/2016).

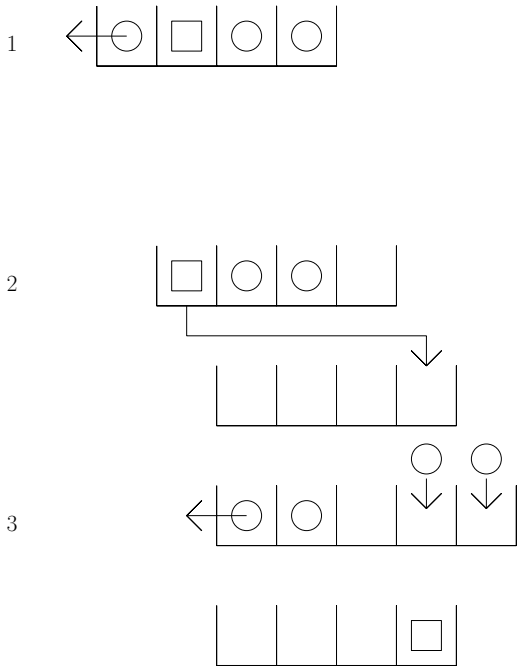
[10] Andrew S Tanenbaum. *Computer networks*. 4th ed. Prentice-Hall, 2003.

[11] J. T. Wen and M. Arcak. "A unifying passivity framework for network flow control". In: *IEEE Transactions on Automatic Control* 49.2 (2004), pp. 162–174. ISSN: 0018-9286. DOI: 10.1109/TAC.2003.822858.

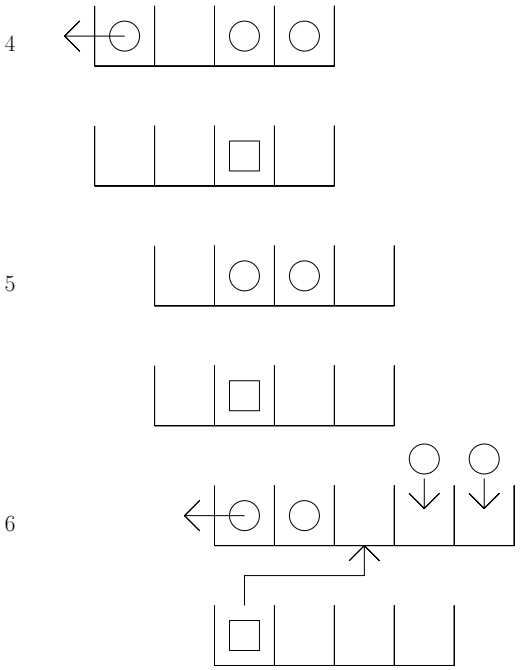
A

Examples of Upper Bounds for Fluctuating Task Completion Rates

When there is no overestimate but the task completion fluctuate, the upper bound of the is proportional to the number of clients in the worst case. We provide two examples for this, one for DIBA (Figure A.1), another for Vector (Figure A.2).

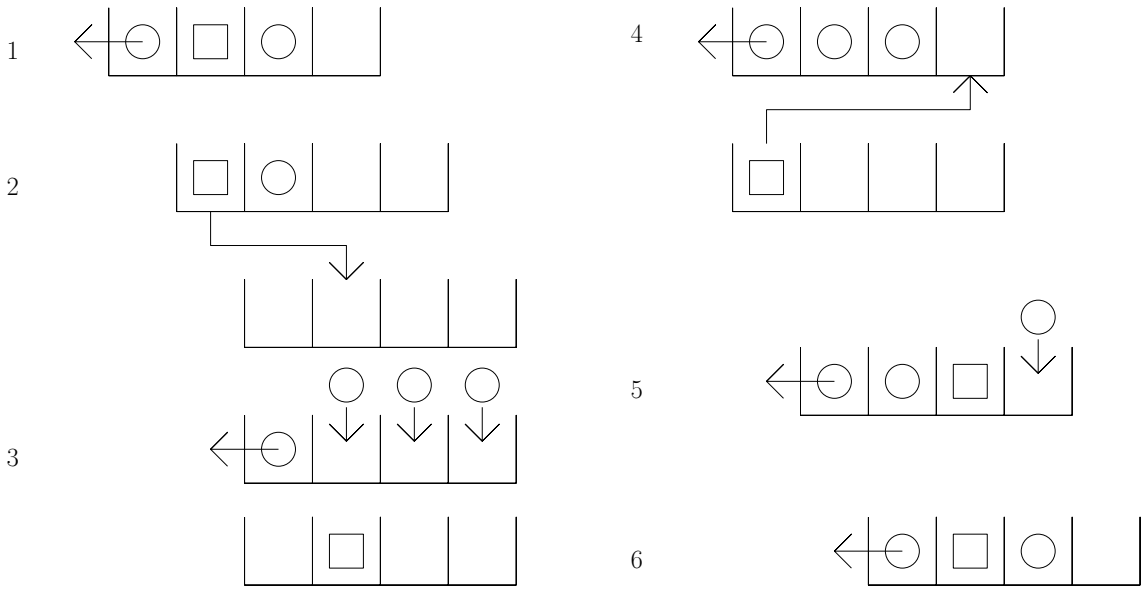


(a) The task completion rate is $\frac{1}{2}$ for step 1 and 2, and 1 for step 3.



(b) The task completion rate is $\frac{1}{2}$ for step 4 and 5, and 1 for step 6. One more step and we are back at step 1 were we started (given that the next time unit has task completion rate $\frac{1}{2}$)

Figure A.1: Example for unbounded return for DIBA when there is varying task completion rate. The upper queue within each step corresponds to a queue with return rate of 1, and if there is a lower queue within a step, the return rate of the other queue is $\frac{1}{2}$.



(a) The task completion rate is $\frac{1}{2}$ for step 1 and 2, and 1 for step 3.

(b) The task completion rate is 1 for step 4 and 5, and $\frac{1}{2}$ for step 6 (same as step 1).

Figure A.2: Example for unbounded return for Vector when there is varying task completion rate. The upper queue within each step corresponds to a queue with return rate of 1, and if there is a lower queue within a step, the return rate of the other queue is $\frac{1}{2}$.

B

Additional Results and Data Values

B.1 Task Completion Rate

The task completion rates for each test run is presented in Table B.1. Note that differences in task completion rates for the different tests does not depend on which algorithms that was used.

Table B.1: Task completion rate for the test for each test setup

	Test 1	Test 2	Test 3
DIBA	5.8	7.1	6.0
Vector	5.9	7.3	6.4
Fairness-DIBA	5.9	6.5	6.3
Fairness-Vector	5.9	6.1	6.2

The task completion rate for the application server was more or less constant within each test run. We show an example for that in Figure B.1, with the distribution of finished jobs for the test run on test setup 1 for DIBA. We do not provide more graphs on this, due to space limitations. In the figure, a trend line is drawn top of the data values, where the task completion rate is 7.1 tasks per second for the entire run.

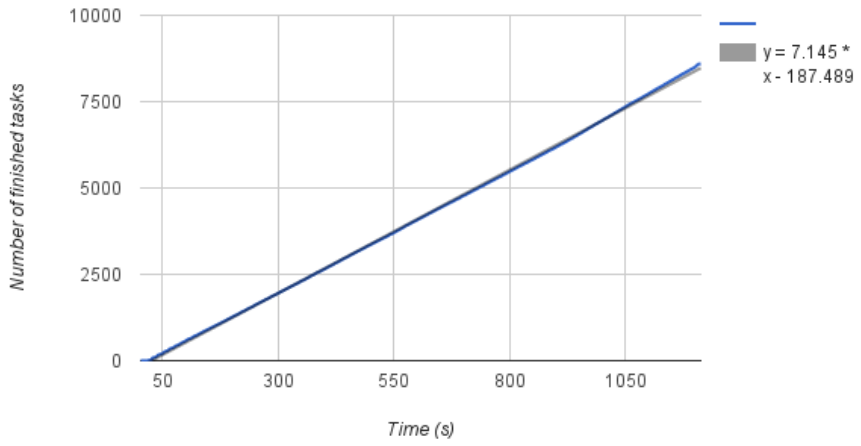


Figure B.1: Example that the task completion rate is constant. Number of finished tasks, where the x-coefficient corresponds to the task completion rate

B.2 Average Task Completion Time

The average task completion time from the application server for each test.

B.2.1 Test 1

DIBA

Figure B.2: Total average task completion time

Vector

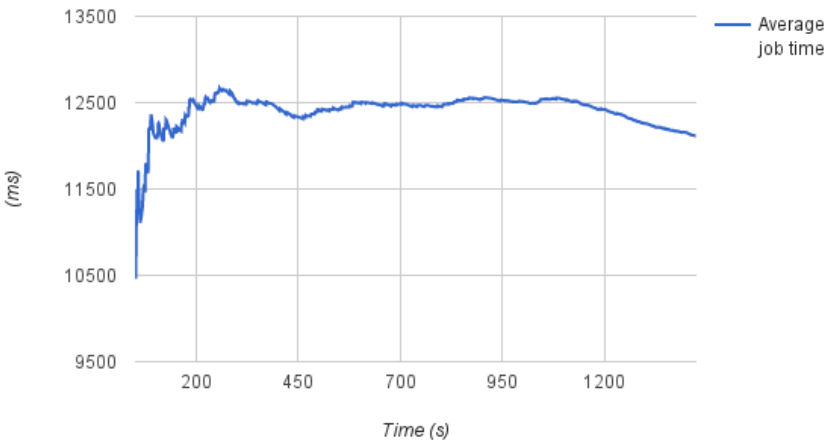


Figure B.3: Total average task completion time

Fairness-DIBA

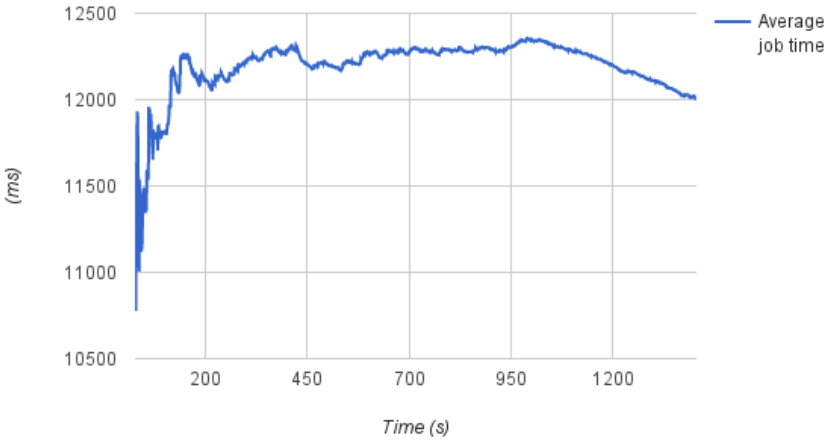


Figure B.4: Total average task completion time

Fairness-Vector

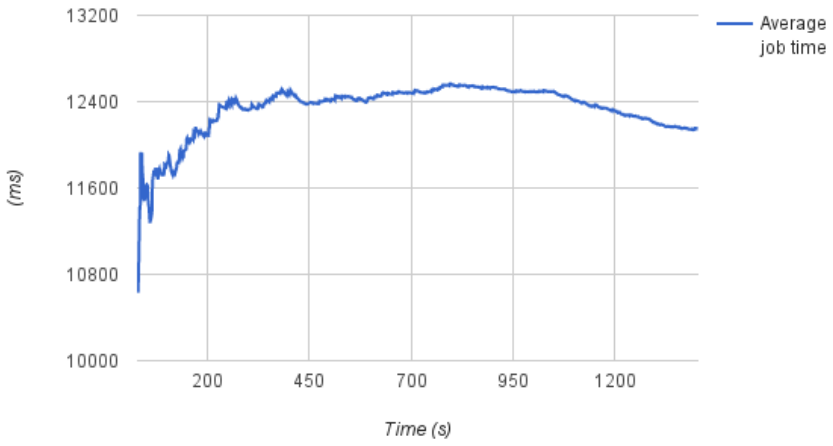


Figure B.5: Total average task completion time

B.2.2 Test 2

DIBA

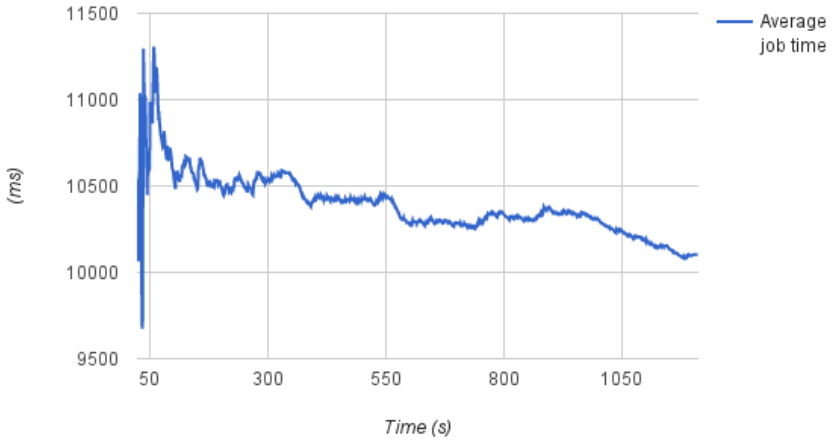


Figure B.6: Total average task completion time

Vector

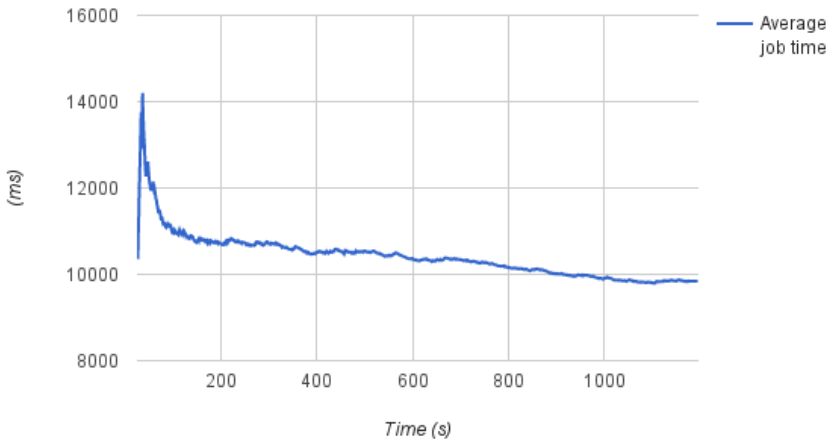


Figure B.7: Total average task completion time

Fairness-DIBA

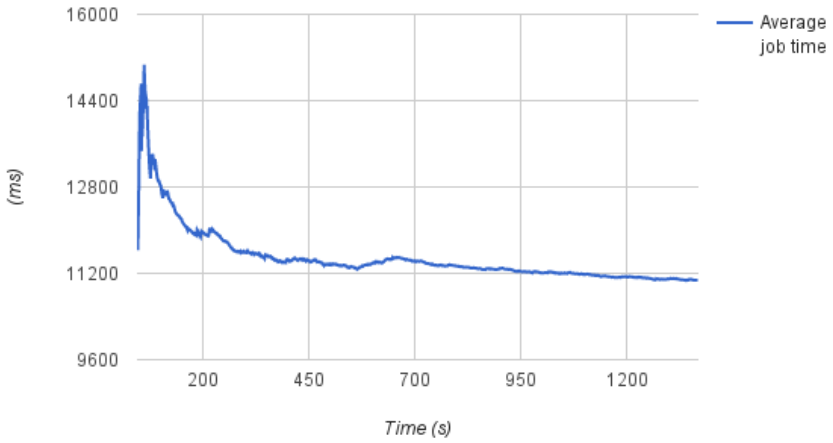


Figure B.8: Total average task completion time

Fairness-Vector

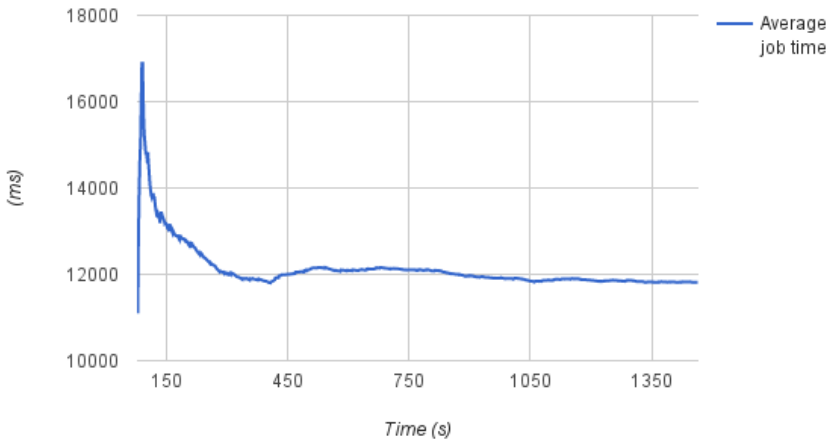


Figure B.9: Total average task completion time

B.2.3 Test 3

DIBA

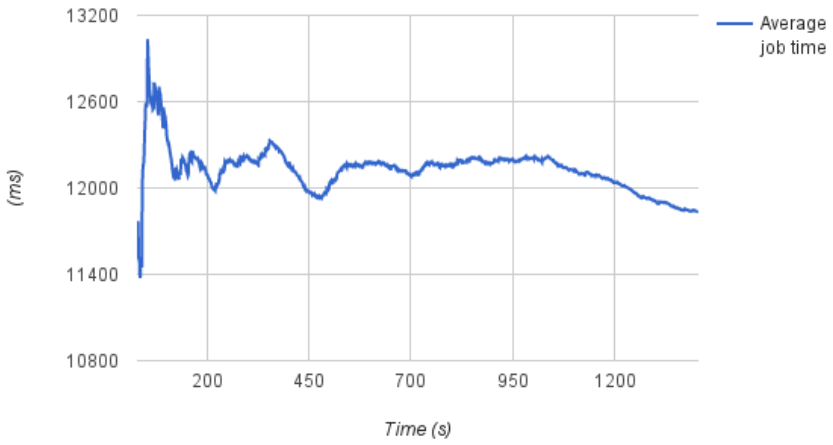


Figure B.10: Total average task completion time

Vector

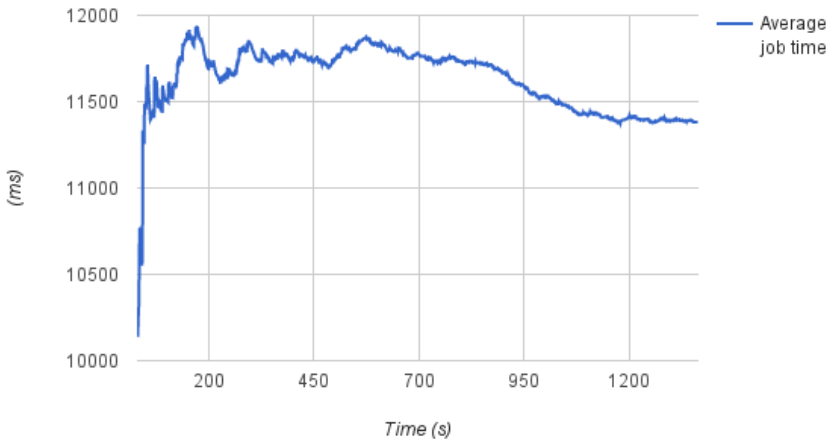


Figure B.11: Total average task completion time

Fairness-DIBA

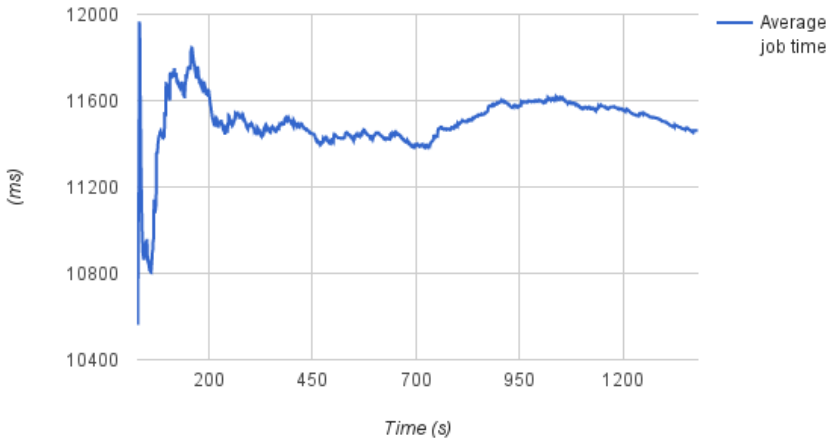


Figure B.12: Total average task completion time

Fairness-Vector

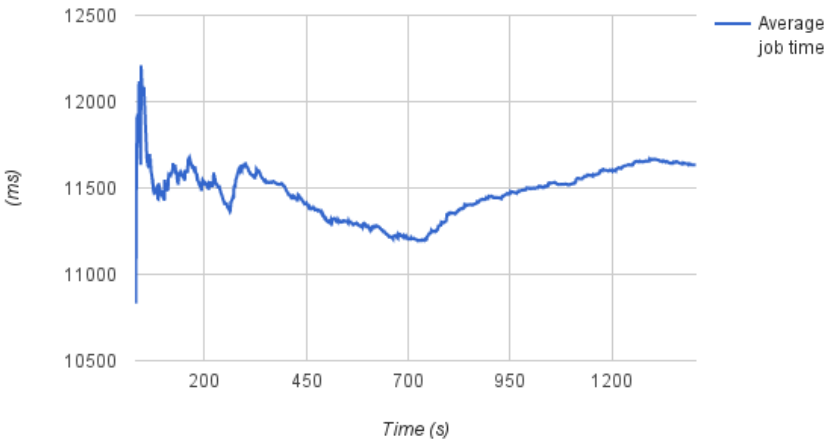


Figure B.13: Total average task completion time

B.3 Standard Deviation for the Task Completion Time

The standard deviation for task completion times from the application server for each test.

B.3.1 Test 1

DIBA

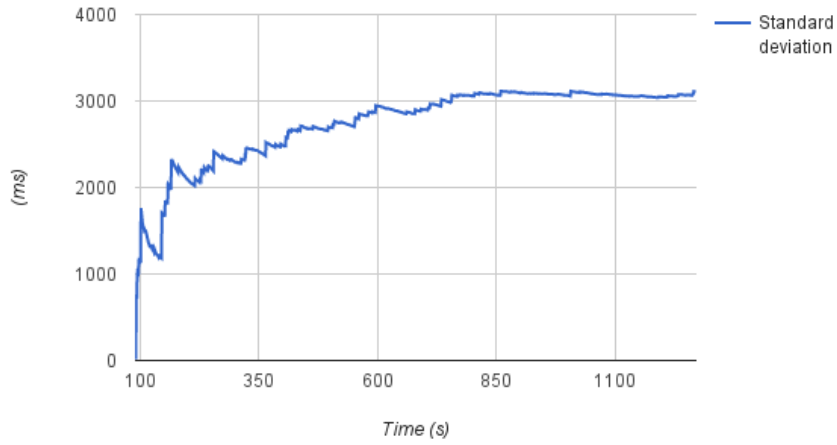


Figure B.14: Standard deviation for task completion time throughout the test.

Vector

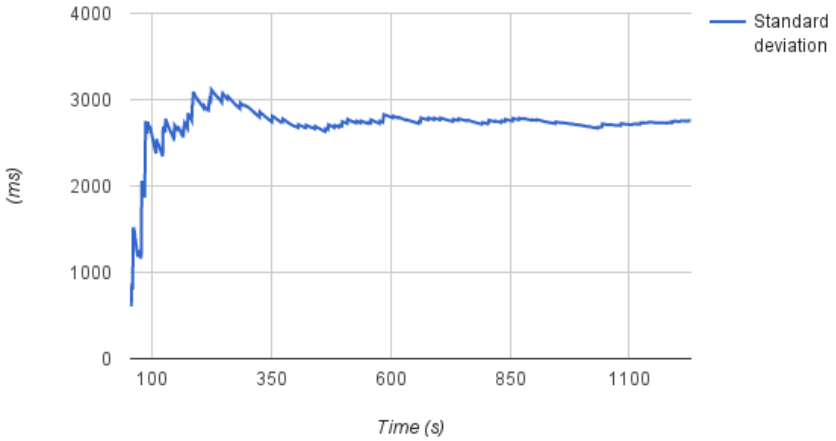


Figure B.15: Standard deviation for task completion time throughout the test.

Fairness-DIBA

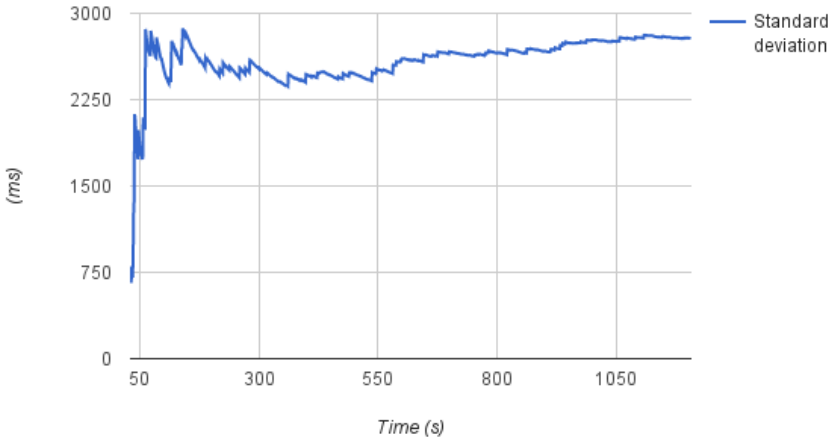


Figure B.16: Standard deviation for task completion time throughout the test.

Fairness-Vector

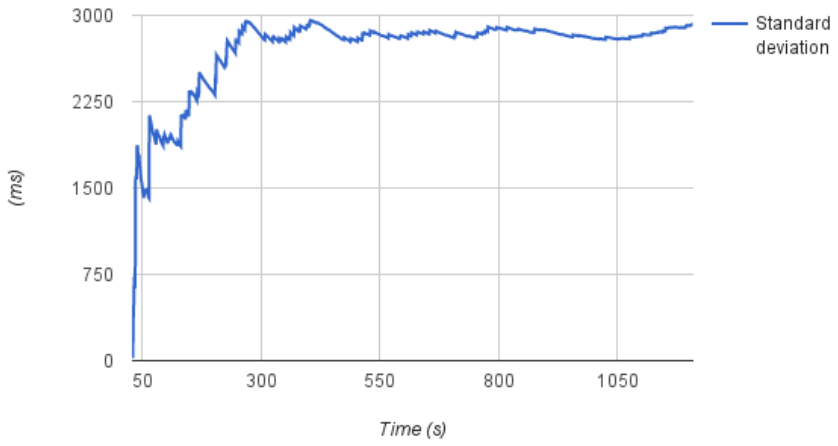


Figure B.17: Standard deviation for task completion time throughout the test.

B.3.2 Test 2

DIBA

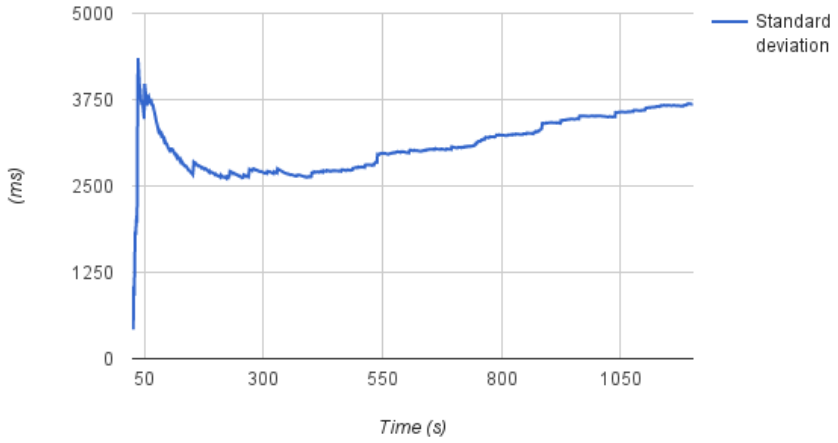


Figure B.18: Standard deviation for task completion time throughout the test.

Vector

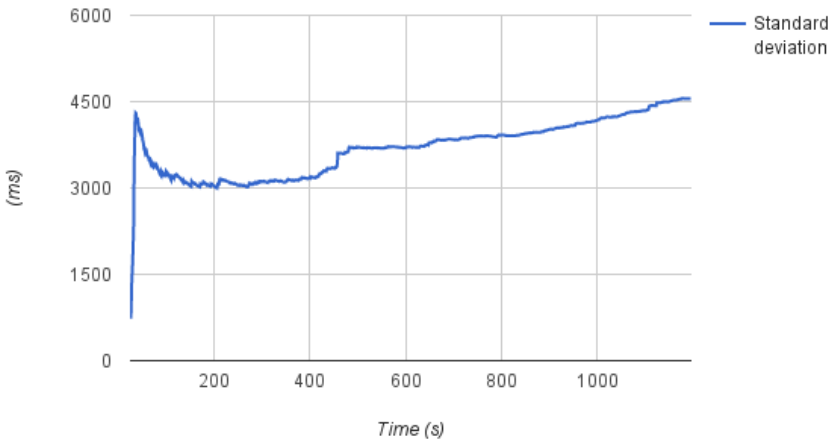


Figure B.19: Standard deviation for task completion time throughout the test.

Fairness-DIBA

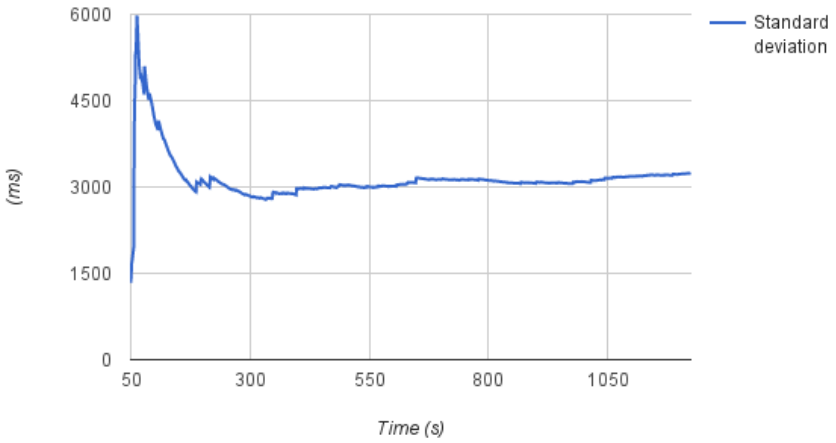


Figure B.20: Standard deviation for task completion time throughout the test.

Fairness-Vector

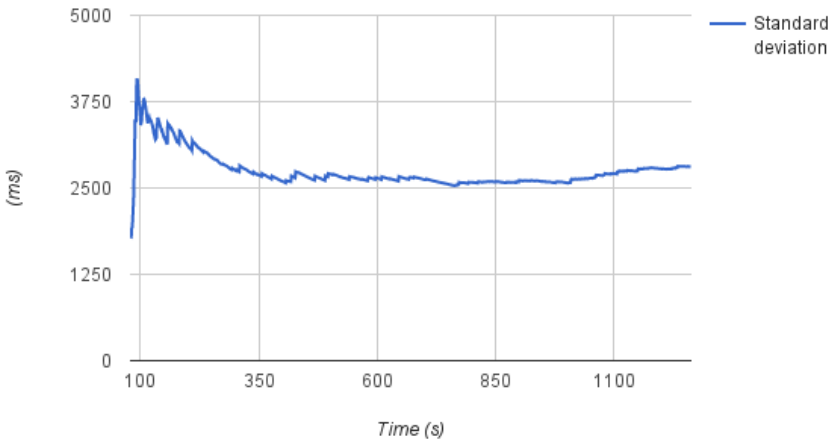


Figure B.21: Standard deviation for task completion time throughout the test.

B.3.3 Test 3

DIBA

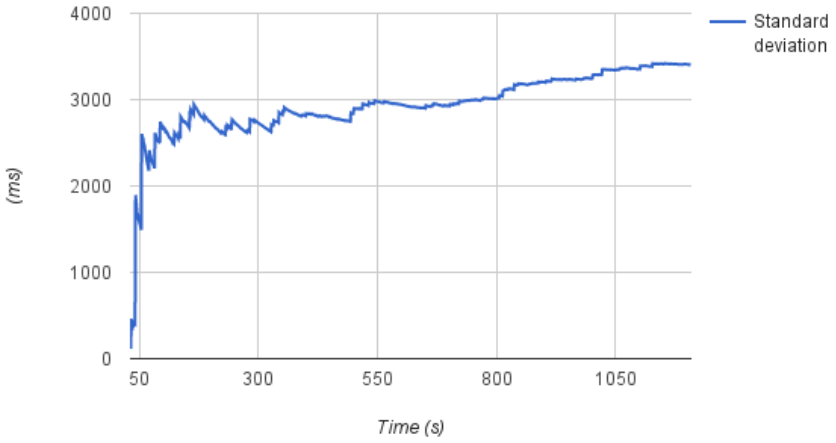


Figure B.22: Standard deviation for task completion time throughout the test.

Vector

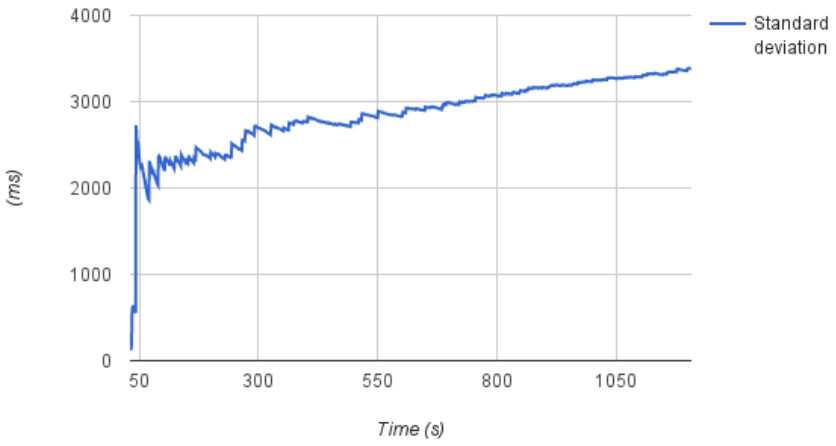


Figure B.23: Standard deviation for task completion time throughout the test.

Fairness-DIBA

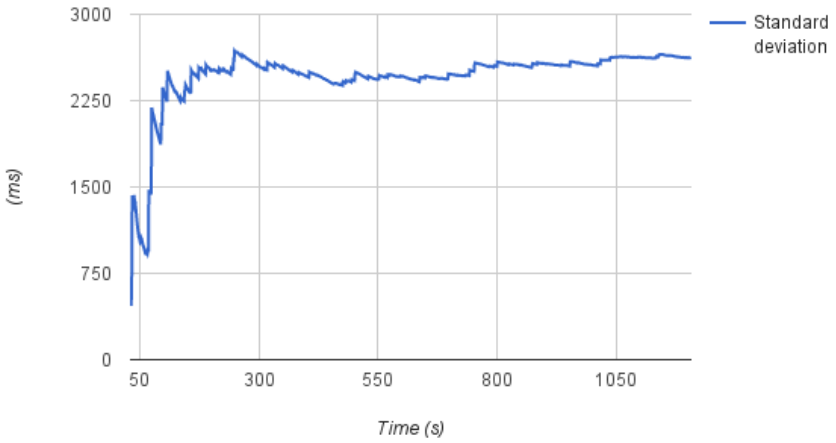


Figure B.24: Standard deviation for task completion time throughout the test.

Fairness-Vector

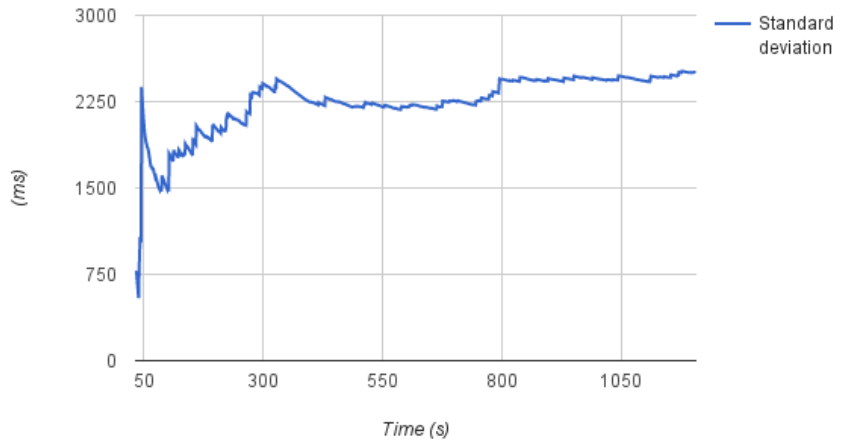


Figure B.25: Standard deviation for task completion time throughout the test.

B.4 Desired Return Rate

The graphs shows what we refer to as the desired return rate (see algorithms in Section 3). If the desired return rate is strictly less than the real task completion rate, we would risk under-utilization (explained in Section 3.2).

B.4.1 Test 1

DIBA

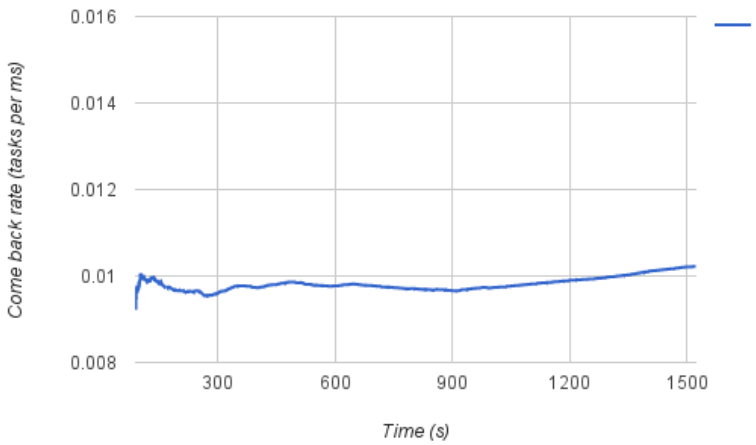


Figure B.26: The desired return rate that was sent to the return time computations.

Vector

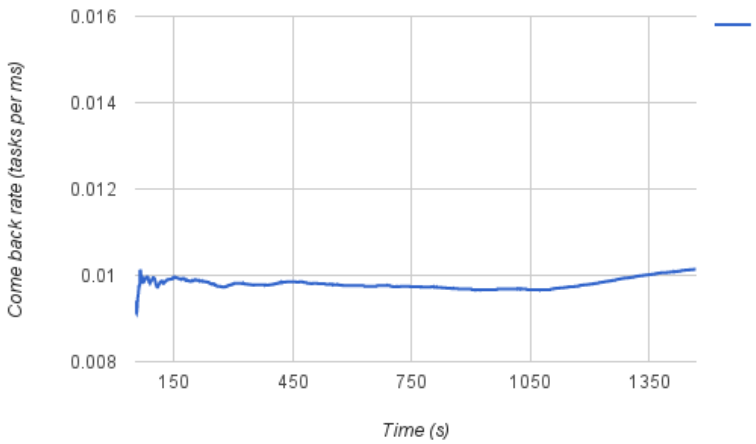


Figure B.27: The desired return rate that was sent to the return time computations.

Fairness-DIBA

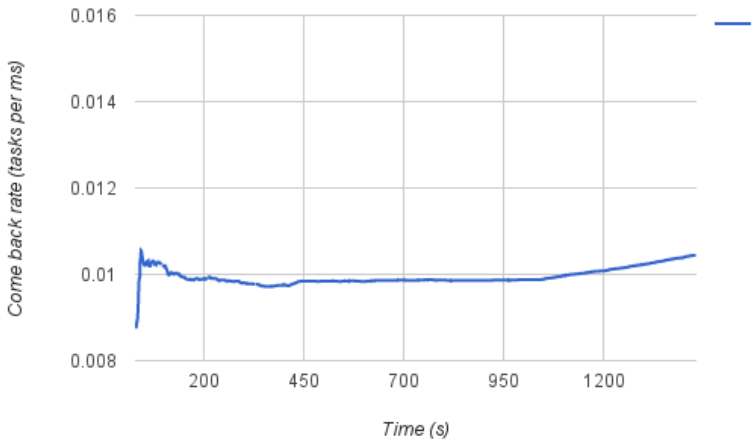


Figure B.28: The desired return rate that was sent to the return time computations.

Fairness-Vector

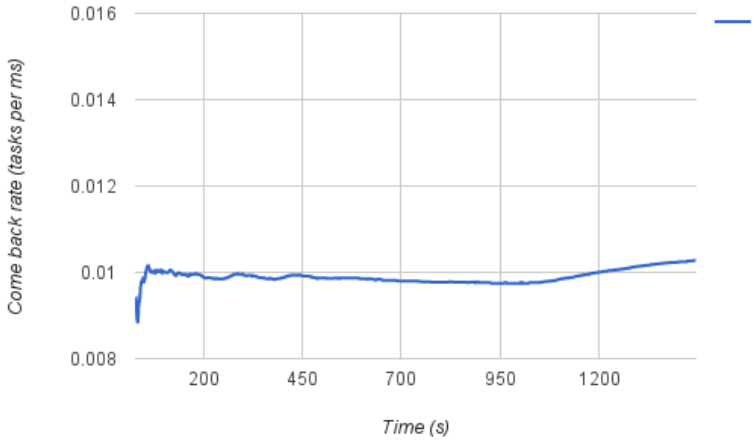


Figure B.29: The desired return rate that was sent to the return time computations.

B.4.2 Test 2

DIBA

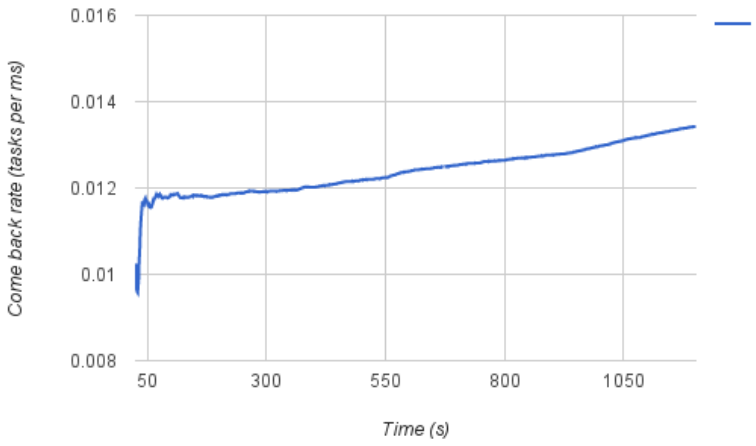


Figure B.30: The desired return rate that was sent to the return time computations.

Vector

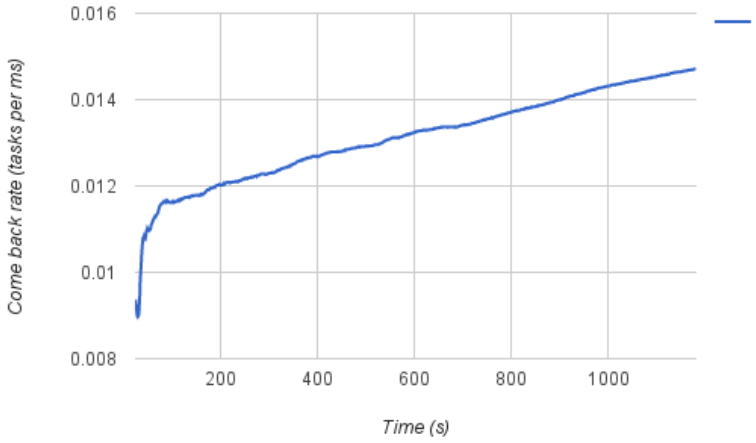


Figure B.31: The desired return rate that was sent to the return time computations.

Fairness-DIBA

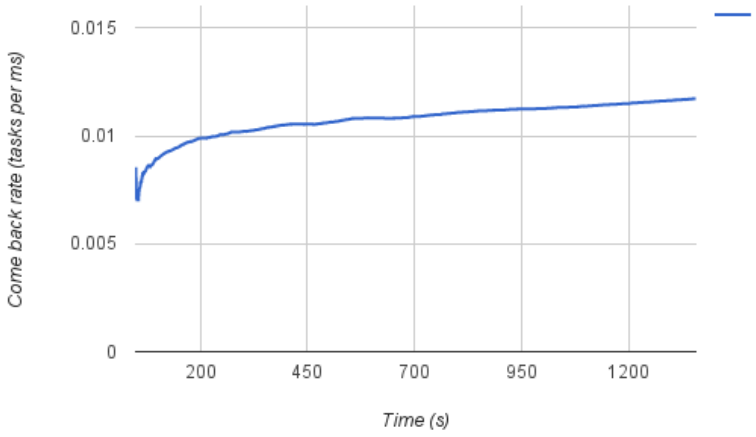


Figure B.32: The desired return rate that was sent to the return time computations.

Fairness-Vector

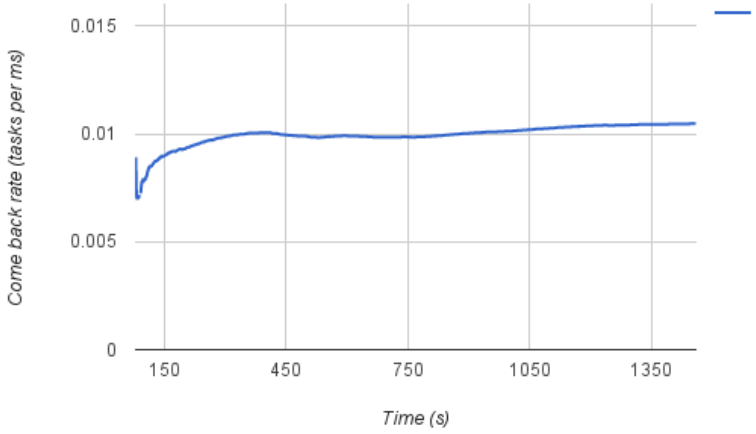


Figure B.33: The desired return rate that was sent to the return time computations.

B.4.3 Test 3

DIBA

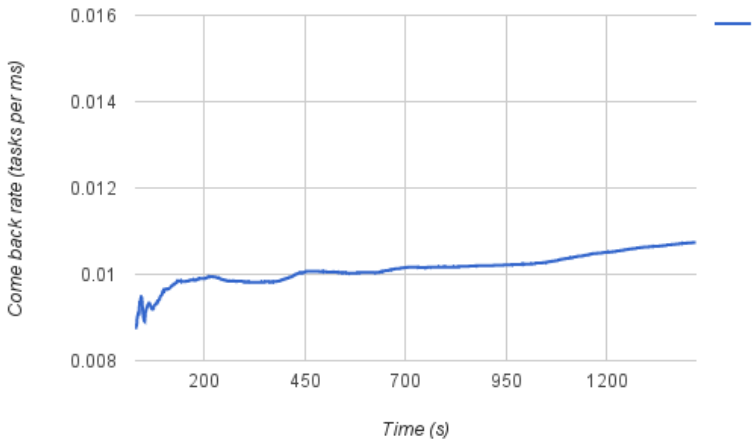


Figure B.34: The desired return rate that was sent to the return time computations.

Vector

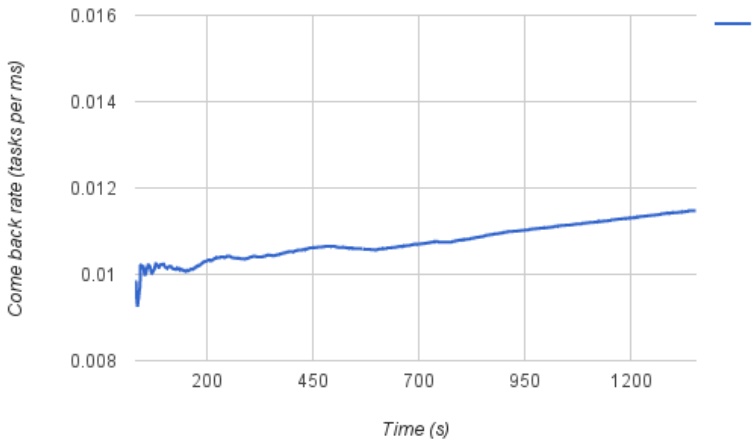


Figure B.35: The desired return rate that was sent to the return time computations.

Fairness-DIBA

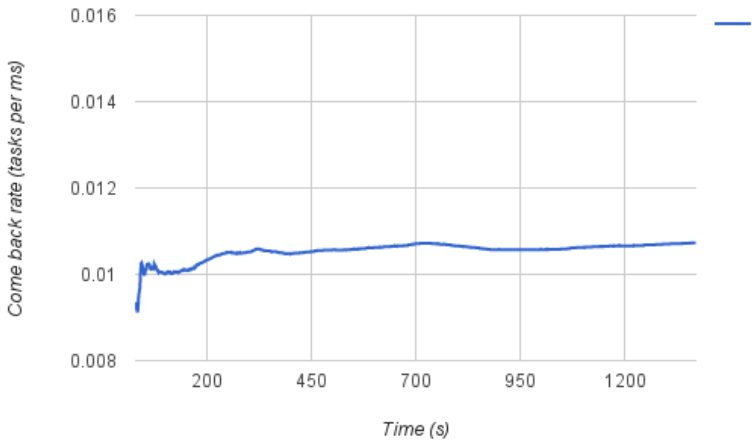


Figure B.36: The desired return rate that was sent to the return time computations.

Fairness-Vector

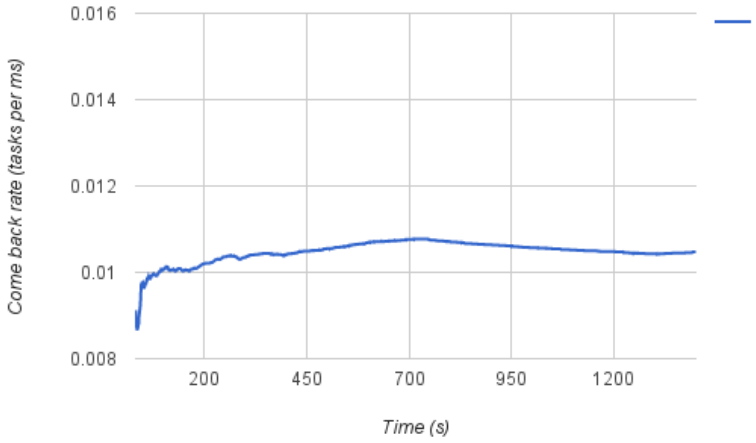


Figure B.37: The desired return rate that was sent to the return time computations.