AN INVESTIGATION OF A SERVER STATE-AWARE TASK DISPATCHING SOLUTION FOR CLUSTERED JAVA EE WEB APPLICATIONS AS A COMPETITIVE ALTERNATIVE TO INDUSTRY-STANDARD SOLUTIONS.

By

Nicola Crisologo de Sousa

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

2012/09/21

ABSTRACT

AN INVESTIGATION OF A SERVER STATE-AWARE TASK DISPATCHING SOLUTION FOR CLUSTERED JAVA EE WEB APPLICATIONS AS A COMPETITIVE ALTERNATIVE TO INDUSTRY-STANDARD SOLUTIONS.

By

Nicola Crisologo de Sousa

The task assignment policies that are available in current industry-standard server load balancing solutions for Java EE environments, such as the F5 Networks BIG-IP Local Traffic Manager and the Open Source Apache HTTPD Server with the mod_proxy plugin, do not consider performance and resource utilisation metrics from Java Virtual Machines and Java EE Application Servers in their task assignment decisions.

As a result these solutions may not be able to provide effective task assignments within clustered Java EE-based Web Application environments without the inclusion of specialized capacity planning skills and resources, complicating the realization of non-functional quality requirements such as availability, fault-tolerance, reliability, scalability, and performance.

This research has investigated the application of task assignment policies in a variety of capacity-sensitive Java EE environments with a prototype task dispatcher that provides implementations the policies available in the aforementioned solutions, and a proposed new policy that has been designed to overcome the limitations of these policies in Java EE environments.

These task assignment policies and the Java EE architecture was examined as a load-balancing and capacity planning problem domain with a focus on core Java EE resources such as JVM heap space, HTTP connection queue capacity, HTTP connection queue worker threads, and JDBC database connection pooling.

The evaluation results highlight the many challenges resulting from the use of task assignment policies in capacity-sensitive Java EE environments, where only 4 of the 12 policies in the evaluation produced no faults.

The static Weighted Round-Robin, Least Connections, and Weighted Least Connections policies produced excellent results. The proposed new server state-aware policy for Java EE environments also accomplished excellent results, while successfully preventing the over-utilization of core Java EE resources.

ACKNOWLEDGEMENTS

I would like to thank my family and friends for their support and encouragement throughout this dissertation.

I also thank and acknowledge Dr. Yanguo Jing for his guidance and assistance as my dissertation advisor and project sponsor.

Finally, I would like to extend my appreciation to the Java Open Source community and the many projects that directly contributed the development of the software produced for this dissertation.

TABLE OF CONTENTS

APPENDICES

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Chapter 1. INTRODUCTION

1.1 Scope

The realization of E-business Web Application quality requirements such as scalability, availability, fault-tolerance and performance, have resulted in a paradigm-shift from traditional single-server (scale-up) to distributed (scale-out) clustered Web Application server architectures.

Clustered Web applications, such as those that can be created with Java EE, require the use of a HTTP front-end task dispatcher (load balancer), where various task assignment policies (load balancing algorithms) can be used to distribute HTTP requests from clients to distributed server nodes for processing.

The appropriate selection of a task dispatcher and task assignment policy thus becomes a fundamental consideration in the realization of several critical E-business Web application quality requirements.

Current Open Source and commercial task dispatchers predominantly provide platform-generic task assignment policies that may not provide optimal results within clustered Java EE environments, thereby complicating the realization of scalability, availability, fault-tolerance and performance quality requirements.

1.2 Problem statement

The primary aim of this research is to evaluate the task assignment policies that are provided by the most utilised Open Source and industry-leading commercial server load-balancing solutions within a clustered Java EE environment; and to improve upon these solutions, if possible, through the research and development of a new server state-aware task assignment policy for clustered Java EE environments.

The secondary aim of the research is to produce a cross-platform Open Source server load-balancing solution for Java EE that provides a complete set of the industry-standard task assignment policies that would normally only be available from commercial hardware-based server load balancing products, where we hope that the addition of the proposed new policy could provide a free, but competitive, alternative to the use of commercial products.

1.3 Approach

The study will carry out a literature review of the task assignment policies that the industry-leading Open Source and commercial load-balancing solutions provide, and will then move on to identify the load balancing problem domain within a clustered Java EE Web application environment.

This will be followed by the proposal of a new task assignment policy for use with clustered Java EE Web applications that is able to support the realisation of scalability, availability, fault-tolerance and performance requirements.

The study will conduct a thorough evaluation of the various task assignment policies within controlled Java EE environments as part of the development of a prototype task dispatcher that provides implementations of these task assignment policies.

1.4 Outcome

The contributions of this research are:

1. An investigation of the task assignment policies that the most utilised Open Source and industry-leading commercial load-balancing solutions provide, with a focus on the Open Source Apache HTTPD mod_proxy (Apache, 2012b) and the commercial F5 Networks BIG-IP Local Traffic Manager (F5 Networks, 2011).

2. The collection of performance and utilisation metrics during the evaluation of the task assignment policies, that the above solutions provide, as part of a comprehensive study into the aforementioned policies in addressing the task dispatching problem domain within clustered Java EE Web applications.

3. The development of an Open Source Java task dispatcher that provides Open Source implementations of the aforementioned policies as well as the proposed new policy. The task dispatcher will assist in the conduct of further analysis into the problem domain, and the resulting evaluation of the various task assignment policies within Java EE environments.

4. The proposal of a new Java EE server state-aware task assignment policy that is able to base its task assignment decisions on Operating System, Java EE Application Server and Java Virtual Machine metrics, as a competitive alternative to the aforementioned solutions that are currently unable to consider all of these metrics in their task assignment decisions.

5. The OOAD and prototype implementation of the proposed task dispatcher and task assignment policy with UML and an Agile-based software development methodology.

6. The conduct of an evaluation for all of the implemented task assignment policies based upon an industry-standard workload specification that will exercise the Apache Geronimo DayTrader "end-to-end Java EE benchmark and performance sample application" (Apache Geronimo, 2010) in Java EE environments with varying capacity configurations for core resources.

1.5 Summary

The study within this dissertation will provide a theoretical introduction to the problem domain, with a strong focus on the theory and background that will support the realization of the research outcomes.

The exploration of these outcomes through literature and related research (Chapter 2) and the discussion of the industry-relevant background information and task assignment policies (Chapter 3) will make a significant

contribution to this research; where it is hoped that a critical understanding of these outcomes is presented, such that the reader will be better informed of the problem domain that is being considered as part this research, and the resulting justification (Chapter 6) for the research and development of the task dispatcher and the proposed Java EE task assignment policy (Chapters 4 and 5).

Chapter 2. Background and Review of Literature

2.1 The task assignment problem domain in a clustered Java EE environment

2.1.1 Introduction to the Java EE architecture

Multitier architectures support the development of applications within a layered architectural style that encourages the separation of concerns within the system, such that architectural advantages with regards to non-functional quality requirements such as scalability, availability, security, reusability, flexibility and maintainability, can be realised. (Grundy and Liu, 2000)

The Java Enterprise Edition (Java EE) SDK, formerly known as J2EE, realizes these architectural advantages through a set of community-driven technology specifications and Open Source implementations, as depicted in Figure 1.

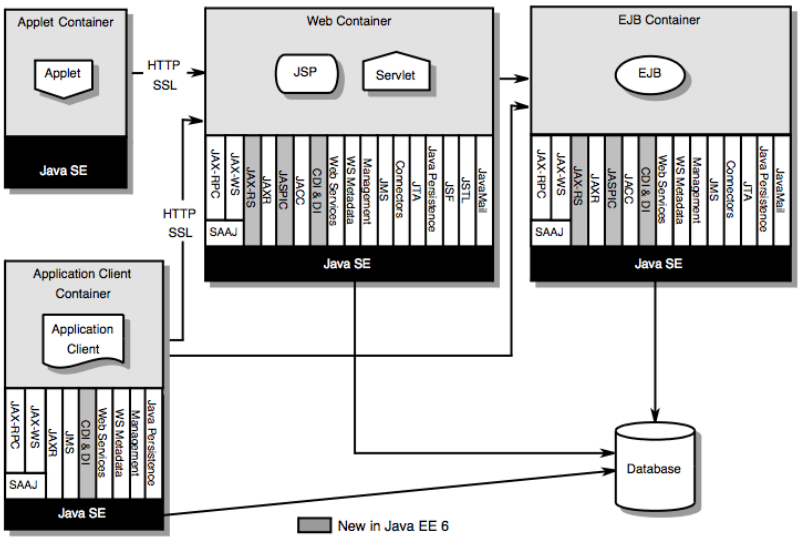


Figure 1 - The Java EE architecture. (Java Community Process, 2009)

The Java EE 6 architecture (Figure 1) is realised in the form of specification-complaint Application Servers that provide implementations of the standardised Java SE and EE APIs that have been defined in the Java EE specification (Java Community Process, 2009).

The Open Source community-driven nature of the Java EE specification and its technologies enable vendors such as Oracle, IBM, Apache, Redhat, and many more, to produce their own specification-compliant Java EE Application Servers.

Oracle, as the custodian of Java standards and technologies, provides a free Open Source Application Server reference implementation, known as GlassFish Server, for the Java EE specification(s).

GlassFish (GlassFish, 2012) is Oracle's reference implementation of the Java EE 6 specification and could thus be viewed as the industry-standard Application Server for Java EE 6, as other vendors often take extended periods of time to provide similarly compliant implementations of the latest Java EE standards.

Application Servers provide the lifecycle management, security, deployment and runtime resources and services that Java EE applications require, where Java EE applications are developed according to standardised interface specifications that are then supported by compliant Application Server implementations.

Java EE Application Server services include the management of resources, transactions, security, client access, and other functionalities that are required within the Java EE platform.

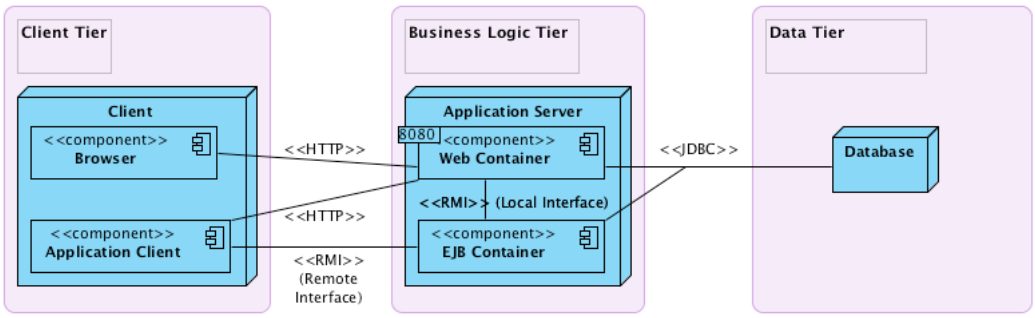


Figure 2 - The default deployment architecture of 3-tier Java EE Web applications.

In this study, as illustrated in Figures 1 and 2, the primary focus will be on a Java EE 3-tier (i.e. client, business logic and data\EIS tier) architecture where end-users interact with the system via HTTP Web application user interfaces that are provided in the Application Server's Web Container.

Web Containers are provided by Application Servers and host deployed Java EE Web applications. These Web applications then consume the various services that are provided by the Application Server, such as transactional access to business logic in the form of Enterprise Java Beans components (i.e. Entity, Session and Message Beans), and other core resources, such as HTTP request queues, connectors and JDBC database connection pools.

2.1.2 HTTP request task dispatching in clustered Web application environments

Classic distributed server environments are implemented with task dispatching mechanisms where a dispatcher is used for the assignment of tasks from a queuing mechanism to distributed servers. (Grundy and Liu, 2000)

A Web Server cluster, as illustrated in Figure 3, is an example of a distributed Web server environment where a task dispatcher queues and then assigns client HTTP requests to server cluster nodes through the use of a queue and a task assignment policy. (Andreolini, Colajanni and Nuccio, 2003)

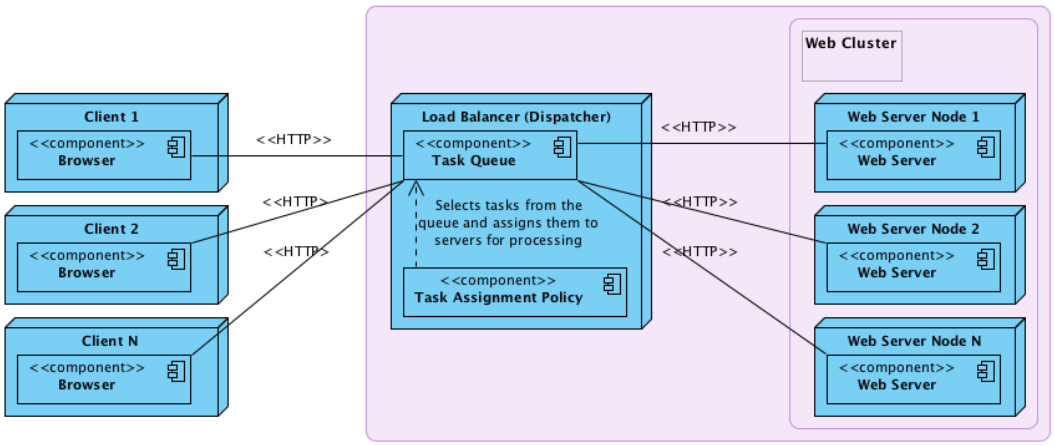


Figure 3 - HTTP request load balancer (task dispatcher) in a Web server cluster.

In the above environment, a task dispatcher enables HTTP clients to access a distributed Web Server cluster through the use of a single Virtual IP address (VIP), where the dispatcher (similar to a proxy server) accepts HTTP requests from a client, and then transparently assigns those HTTP requests from a queue to a destination Web server node for processing.

The use of a task dispatcher in such environments provides support for the realization of scalability, availability, fault-tolerance and performance requirements.

As a consequence, however, the selection of an appropriate task assignment policy will play a significant role in the realization of these quality requirements – especially so within specialized environments, such as Java EE. (Andreolini, Colajanni and Nuccio, 2003)

The use of a task dispatcher and predominantly static task assignment policies has therefore become an industry-standard solution for HTTP request dispatching (load balancing) in clustered HTTP server environments.

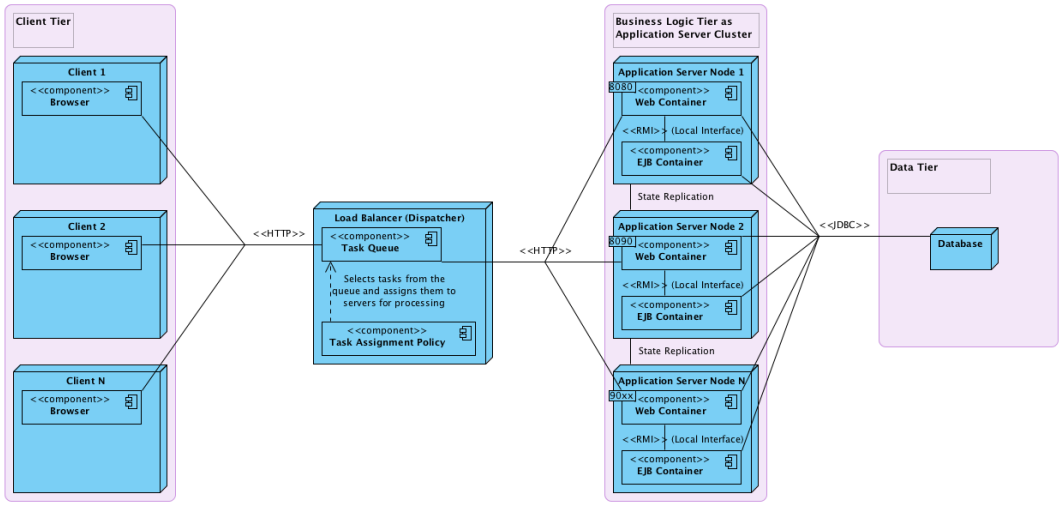


Figure 4 - HTTP task dispatcher in a Java EE clustered environment.

Industry-standard task dispatching solutions, however, are often very generic (environment-independent) as illustrated by the strong similarity between Figure 3 and Figure 4, where the failure to consider the performance and utilisation of core Java EE resources may thus complicate the realisation of quality requirements.

2.2 Metrics used in the performance evaluation of task assignment policies

2.2.1 Introduction

For the purposes of this study an evaluation of the task assignment policies will be conducted from a task dispatcher's perspective, with a focus on HTTP request throughput, failures and latency (as well as other metrics) within Java EE environments where capacity limitations are created.

The efficiency of task assignment policies will be evaluated, where:

- **Latency** represents the average response time that the system under test (cluster) took to complete a task,
- **Throughput** represents the amount of tasks that were completed successfully,
- and **Capacity** that represents the resource availability that a system requires while adhering to demand, latency and throughput requirements.

Studies of task assignment policies in Web clusters, such as Casalicchio and Tucci (2001), Cardellini et al. (2002), Casalicchio, Cardellini and Colajanni (2002), and Challenger et al. (2004) also identified response time (latency) and throughput as the primary metrics in the performance evaluation of task assignment policies.

Latency is then analyzed in terms of waiting time, running time and slowdown, where:

- **Waiting time** is the amount of time that a task would be delayed by before being accepted for processing by a server. Waiting time includes the processing lag that results from the tasks' relative position (and priority, etc.) in the dispatching queue, as well as the efficiency of the task assignment policy in selecting a destination server, and the task dispatcher's delay in sending the task to the selected server for processing.

- **Running Time** is the processing time of a task that has been sent to a cluster node for processing,

- and **Slowdown** is defined (for the purposes of this study) as the waiting time + running time, divided by the running time, reflecting the performance characteristics of the task dispatching function.

2.2.2 The application of performance evaluation metrics in the study of task assignment policies

The slowdown metric is of great importance in the study of task dispatchers and task assignment policies, where advanced task assignment policies attempt to reduce slowdown with various strategies.

Server state-aware task assignment policies, for example, are able to reduce running time by assigning a task to a server with the least amount of system load (e.g. CPU utilization), but will incur a certain amount of slowdown due to the consideration of server state information (waiting time).

More advanced task assignment policies, such as state-of-the-art Quality of Service task classification policies utilize service (task) differentiation

and server partition strategies, where tasks are classified into different processing priorities, and then dispatched on the basis of server utilization.

Older works, such as Denning and Buzen (1978), provided a detailed analysis of these metrics as part of a study into network queuing models. These metrics, with a particular focus on waiting time, have also been examined in great detail by many more recent task dispatching studies, as a reduction in waiting time improves latency and thus throughput.

Research by Shan et al. (2002), Cherkasova and Phaal (2002), Xiong and Yan (2005), Schroeder and Harchol-Balter (2006) and Andreolini, Casolari and Colajanni (2006), for example, investigated the application of Quality of Service (QoS) principles in their task assignment policies, where task classification and server partitioning schemes were found to be very successful in creating a compromise between end-user perceived latency and service classes.

2.3 Overview of task assignment policies

Research by Andreolini, Colajanni and Morselli (2002, p.2-3) organized task assignment policies into two distinct classes:

- **State-blind** (static) policies that do not consider system state in their task assignment decisions,
- and **State-aware** (dynamic) policies that consider state information, such as server load, in their task assignment (or scheduling) decisions.

Static policies are generally considered as the fastest (in terms of waiting time) task assignment policies, as they do not consider state information in their task assignment decisions. As a result, these policies could provide poor performance (increased latency and failures) when workloads include varying task sizes, such as HTTP requests for dynamic content, as they could assign tasks to a server that has more utilization than another, or in effect, to a server that is over-utilised and producing faults.

Dynamic policies are considered as being superior, with regards to availability and fail-over, and potentially decreased latency, to static policies in that they base task assignment decisions on system state information, such as server CPU utilization, at the cost of additional processing overheads (waiting time) due to the collection and analysis of system state information.

State-aware (dynamic) policies can be further classified with respect to the tier(s) at which they collect state information:

Client state-aware policies base their assignment decisions on the state of client-tier information, such as the client's IP Address, and are limited to the information that is available from a network connection or protocol, such as HTTP request headers (in content-aware policies).

Server state-aware policies base their assignment decisions on the system state of servers.

Client and Server state-aware policies combine client and server state information in their assignment decisions, and may produce a good sacrifice between latency and waiting time.

Research by Cardellini et al. (2002) grouped static and dynamic policy classes into **Content-blind** and **Content-aware** categories, where content-aware policies inspect tasks (such as HTTP requests) as part of their task assignment decisions.

Content-aware policies may also consider the task size of requests, either in terms of file sizes for static content (e.g. SITA-E), or as the processing requirements of the request in terms of server resources such CPU, disk or network I/O (as with the CAP policy), where service differentiation (QoS) is then made possible. These policies fall outside of the scope of our research focus, but have been briefly described in Appendix A.1.

As discussed in Hunt et al. (1998) and Schroeder and Harchol-Balter (2006), the above-mentioned content-aware policies are indicative of the difference between load balancing, and load scheduling, where this study will focus on the equalization of server load.

2.4 Overview of the task assignment policies that are available in the Open Source Apache HTTPD mod_proxy and F5 LTM commercial solutions

Many of the most popular load-balancing solutions currently available are based upon various implementations of the content-blind policies that are discussed in section 3.2.1.

The Open Source Apache mod_proxy (Apache, 2012b) solution supports the use of content-blind, client state-aware policies, including the:

- Random,
- Round-Robin,
- and Weighted Round-Robin policies.

The F5 Networks BIG-IP Local Traffic Manager series of hardware load balancing devices (F5 Networks, 2012) support many of the content-blind (Layer-4), content-aware (Layer-7), and server state-aware policies including the:

- Round-Robin (default policy),
- (static and dynamic) Weighted Round-Robin,
- Least Loaded,
- Fastest Server,
- Least Connections,
- Weighted Least Connections,
- Least HTTP Sessions,
- and the (patent-pending) F5 Dynamic Ratio policy.

2.5 Related Work

2.5.1 Related studies with performance evaluations of the Random, Round-Robin and other Weighted Round-Robin variations

Research by Casalicchio and Tucci (2001) provided a comprehensive evaluation of the Random, Round Robin, as well as several variations of the dynamic Weighted Round Robin policies, such as the Least Connections and Weighted Least Connections policies.

The policies were implemented as an OSI Layer-4 TCP (Webopedia, 2010) task dispatcher.

Evaluations were performed with an analytical model, where inverse Gaussian distributions were used for the number of requests per session, and Pareto distributions for user think time.

The results of the study indicated that the Weighted Round-Robin policy provided the best throughput and that the Random policy was comparable to its performance.

Research by Andreolini, Colajanni and Morselli (2002) conducted a study of several task assignment policies and presented a comparison of their performance characteristics in multitier clustered Web application architectures.

The study evaluated dispatching policies within client, server, and client-server state-blind and state-aware contexts. State-blind policies included the Random, Round Robin, and static Weighted Round Robin (WRR) policies.

State-aware policies included variations of the Least Loaded (LL) (in terms of connections, CPU, disk, and network usage, etc.), the dynamic Weighted Round Robin (WRR), and the Client Aware Policy (CAP) policies.

The policies were implemented in the Apache HTTPD Web Server (Apache, 2012a) with the mod_proxy (Apache, 2012b) and the mod_rewite (Apache, 2012c) plugin modules, with custom scripting code that was used to control the operation of the modules.

The evaluation of the policies was conducted with the httperf tool (httperf, 2011). The number of requests per session was defined as inverse Gaussian distributions and user think time as Pareto distributions.

A Python script was used to generate Pareto distribution workloads where 80% of the requests were for dynamic content, and the remainder as static content requests.

A motivation was provided that this distribution would place more stress on the destination servers. Observations in Java EE environments support this, due to the extensive use of servlets that include only a relatively small distribution of cacheable static content.

The results of the experiments concluded that the Round-Robin policy was relatively stable and provided the best overall response time from all of the evaluated policies.

This observation, however, could be challenged by the running time of the dynamic workload and the resulting capacities of each server. In this

case, it seems probable that identical capacities were used, and that the running time of dynamic requests were very similar, favouring a Round-Robin strategy.

The authors suggested that server state-aware policies (such Least Loaded) could be unreliable due to transient server node load, where LARD provided the highest response time, and CAP only performed slightly better.

2.5.2 Related research on task assignment policies

Research by Andreolini et al. (2001) explored the concept of applying Quality of Service (QoS) principles in a cluster-based Web application as a means to establish multi-class Service Level Agreements (SLAs) where predictable service delivery could be provided for selected classes of users through the use of dynamic server partitioning schemes.

A Quality of Web-based Services (QoWS) policy that uses request classification with admission control, performance isolation and high resource utilisation was described and then implemented as a task dispatcher with scheduling and resource management that was targeted towards cluster(s) that host a single Web Application.

OSI Layer-7 Load Balancing was achieved through the use of an Apache HTTPD Web Server (Apache, 2012a) with the mod_proxy (Apache, 2012b) and mod_rewrite (Apache, 2012c) plugin modules, where a variation of the Weighted Round Robin policy was used to select destination servers.

The QoWS policy was able to communicate with and control the functioning of the mod_rewrite plugin. In order to support dynamic WRR, a load-monitoring agent was installed on each node, which sent periodic server load information to the dispatcher via UDP.

An “active number of HTTP connections” metric was used as the primary server load metric and was then supplemented by other (Linux-based) OS metrics, such as those obtained from the Linux proc file system, such as the system load averages.

The proposed dispatcher was evaluated with the WebStone benchmark tool (Webstone, 1998). A workload generator was implemented that incorporated dynamic content requests with QoS classes that were classified in terms of different types of system users. User request rates were modelled as an inverse Gaussian distribution, and user think times as Pareto distributions.

The results of the experiment confirmed success in meeting all four of the defined QoS SLA targets through the use of dynamic partitioning, and provided a motivation for being a superior approach to other partitioning policies (like CAP), as described in Cardellini et al. (2002).

Related research by Sharifian, Motamedi and Akbari (2009) proposed the implementation of a load approximation-based policy (ALB) that provides admission control for requests.

A fundamental consideration in the study was to find an alternative approach to distributed server load feedback agents. Towards this objective

an evaluation of distributed agents was presented, and highlighted that feedback communication costs could be high (relative to the number of nodes), and that the precision of the server selections could be impacted by the transient nature of server load feedback reports.

This observation has been supported by the other studies included in this review and will become a critical consideration in the proposal of a new server state-aware policy and therefore warrants further investigation.

Server load feedback agents were replaced with an analytical model, similar to Dahlin (2000) and Dinda and O'Hallaron (2000), that was able to estimate the load of servers dynamically by predicting the CPU demands of request classes as well request queue lengths.

A proportional integral (PI) controller was then used as a running time feedback mechanism that was able to reduce analytical workload estimation errors.

The authors described the server state-aware CAP policy as being able to deliver good results with dynamic content, whereas other state-blind policies were described as having low performance with dynamic content requests.

The Weighted Round Robin (WRR), Content Aware Policy (CAP), and proposed ALB policy were evaluated with the (deprecated) RUBiS benchmark (OW2 Consortium, 2009), where the proposed policy (ALB) showed a significant increase in throughput as well as a stabilization of mean response times in contrast to the CAP and the WRR policies.

2.5.3 Server load prediction and load feedback sample ages

Research by Dinda and O'Hallaron (2000) evaluated linear time series models within the context of load scheduling and determined that host load prediction was consistently possible to a very useful degree.

Improved host load-predictions directly contribute towards better predictions for the running time of computationally intensive workloads (e.g. dynamic content requests), and could therefore be applied in server state-aware policies.

The study collected 1 Hz samples of a given hosts load history and was able to successfully predict 1 to 30 second future load averages through the use of a predictive autoregressive model (Bourke, 1998).

Related research by Dahlin (2000) examined the problem domain associated with the consideration of stale feedback information that was obtained from distributed server load feedback agents and presented several algorithms that were suitable for solving such problems.

The primary focus of the study was in the definition of a series of Load Interpretation (LI) policies that were able to schedule workload distribution evenly under varying server load feedback sample ages.

Server load was defined as request queue length in the paper but could be substituted with other metrics as well. Periodic, Continuous, and Update on Access delay models were also applied successfully.

An increase in sampling frequency, however, may yield more accurate results than predictions can, at the expense of increased communication costs. This study will therefore attempt to use a solution where feedback samples are embedded into HTML responses in order to reduce the communication costs of high frequency feedback samples.

2.6 Summary

This chapter has examined load balancing in Java EE environments as a problem domain and defined the metrics that will be used in the evaluation of the task assignment policies that are available from the leading Open Source and commercial load balancing solutions.

The related research section presented an overview of the current state-of-the-art research being done on Quality of Service based task assignment policies, and discussed the lack of accuracy in server assignment decisions, due to out-dated server state feedback information.

Chapter 3. THEORY

3.1 The primary non-functional considerations for successful Web applications

3.1.1 Introduction

On the 1st of January 2012 the Twitter (Twitter, 2012) social network experienced several outages, as it was unable to support a service demand of approximately 16,000 “tweets” per second (Daily Mail, 2012).

This is one of many recent E-business website outages that highlight the ever-increasing non-functional challenges, such as scalability, availability, fault-tolerance, performance and reliability that many E-business organizations encounter as a result of conducting business on the Internet.

The consequences of failing to achieve these non-functional requirements could be severe:

- Research by from Amazon indicated that “every 100ms delay [in latency] costs 1% of sales” (Linden, 2006).

- Research by Eric Schurman (Yahoo and Microsoft Bing) and Jake Brutlag (Google Search) demonstrated a strong relationship in the “reductions of distinct queries, query refinements, revenue per user, amount of clicks, and overall satisfaction” as a result of end-user perceived latency, where customers often switched to competitors sites whenever services were “slow”. (Shurman and Brutlag, 2009)

- The Aberdeen Group (Aberdeen Group, 2008) conducted a study of more than 160 E-business organizations and presented statistics that suggested a 7% loss of income due to an additional 1-second delay in latency:

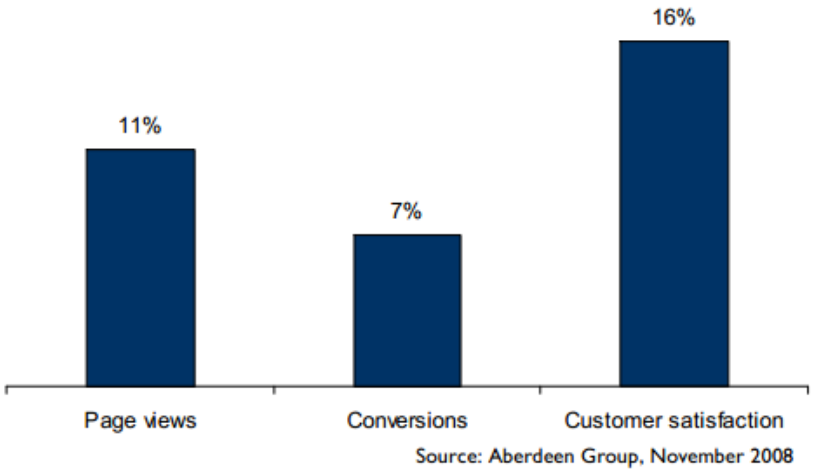


Figure 5 - The average impact of an additional 1-second delay in response times. (Aberdeen Group, 2008)

The report listed the following considerations for successful Web applications:

- "Application availability,
- Success rate in preventing issues with the performance of Web applications before end-users are impacted,
- Improvements in response times of Web applications"

More formally these considerations transition directly into the most important user-perceived non-functional quality requirements that form the norm for competitive E-business services, including: availability, fault-tolerance, reliability, scalability, and performance.

The Java EE architecture provides availability, scalability, fault-tolerance and performance for EJB-related services, where the realization of quality goals beyond the middle-tier layer is largely dependant on the application server implementations that vendors provide and support.

Application server vendors thus often compete on the basis of the scalability, performance, fault-tolerance and reliability goals that their middle-tier implementations support, with a primary focus on middle-tier services and not the dispatching of HTTP requests to nodes.

This establishes a very significant concern leading to the use of 3rd party load balancing solutions, because very few Application Server vendors include advanced HTTP request load balancing solutions in their implementations.

Equally concerning, however, is the introduction of a master-node architectural pattern where a single (“domain controller”) node is responsible for the dispatching of HTTP requests in a cluster that it represents, usually with the Round-Robin or static Weighted Round-Robin policies, as is the case in the current version of GlassFish Server (Figure 6).

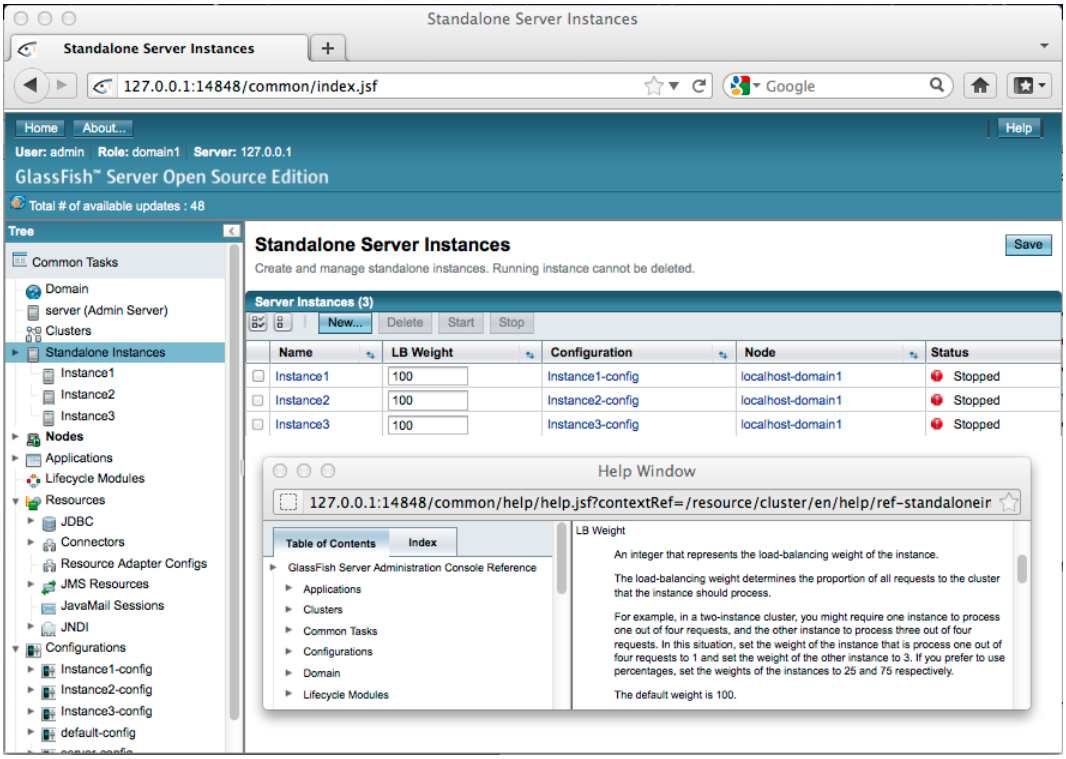


Figure 6 - The static WRR-based configuration of clustered nodes on Glass-Fish Server.

The realization of scalability, availability, fault-tolerance, reliability and performance requirements for Java EE Web applications are thus constrained by vendor provided support and 3rd party load balancing solutions that fall outside of the Java EE specification, often introducing vendor lock-in where a transition from one application server vendor to another can be very challenging and costly.

The introduction of single-point of failure situations, as is the case with the domain controller node (if present), can be addressed with high-availability (redundant) load balancing solutions, where HTTP requests are dispatched by an external load balancing solution that dispatches the client HTTP requests it receives directly to cluster nodes on their respective HTTP listener ports.

3.2 Taxonomy of content-blind policies

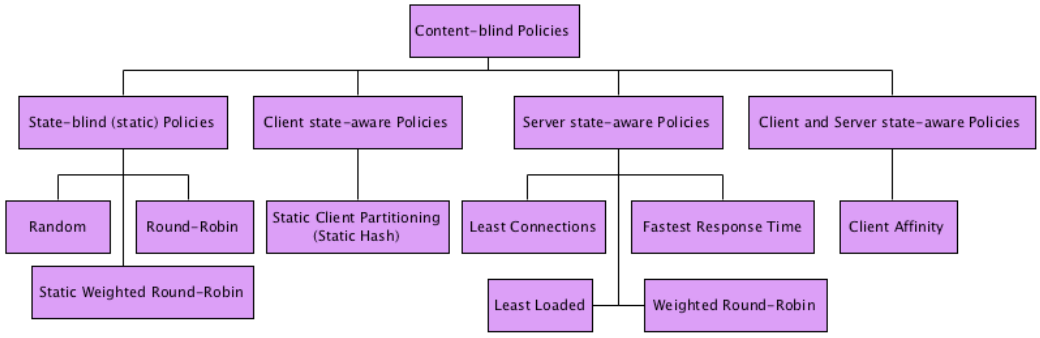


Figure 7 – An overview of content-blind task assignment policies.

3.2.1 State-blind (static) policies

The **Random** (RND) policy assigns tasks to a server in a (pseudo) random order, where each server in a cluster of size *N*, has the probability of *1 / N* to receive a task.

The **Round-Robin** (RR) policy assigns tasks to servers in a cyclic order, where each task is sent to a server *S* in a cluster of size *N*, with the probability of *S mod N*.

The **Static Weighted Round-Robin** (static WRR) policy is a variation of the Round-Robin policy where each server is assigned a ratio or weight for the assignment of tasks in a cyclic round-robin order.

The RAN, RR and static WRR policies attempt the uniform distribution of incoming tasks to servers, and do not consider task size variation or system state information in their task assignment decisions.

As a result, task size variability would have an impact on latency, especially so if we consider heavy-tailed request arrivals as discussed in Harcol-Balter (2002).

While these policies are able to support scalability, availability and performance requirements, they may be unable to support fault-tolerance and reliability requirements, as they may assign requests to servers that are over-utilized or that have failed.

3.2.2 Client state-aware policies

Static Client Partitioning policies assign tasks to a server through the use of a static server partitioning mechanism. The client's IP Address is converted to a number *H* with a hashing function, and the task is then assigned to a server in a cluster of size *N*, with *H mod N*.

As discussed in Cardellini et al. (2002), this policy guarantees that a request from a certain client IP Address will always be dispatched to the same server, and thus maintains client-server affinity, encouraging the use of (local) caching mechanisms for static requests in order to increase performance.

The use of this policy on Java EE environments would not make sense since the large majority of requests would be for dynamic content, where most browsers would already cache the static content for well-designed applications.

This policy also exhibits the same concerns as the state-blind policies, and would break the client-server affinity rule, should the clients IP Address change during the course of server interactions.

3.2.3 Server state-aware policies

The **Least Loaded** (LL) policy assigns a task to a server that has reported the least amount of "load" during an observation period.

As described in Andreolini, Colajanni and Morselli (2002), the LL policy could be adapted to characterise load in various ways, and is the most common type of dynamic policy provided by commercial load-balancing solutions, such as the F5 LTM.

As a result, there are several known variations:

The **Least Connections** (LL-CONN) policy assigns a task to a server with the least amount of active connections during an observation period.

The **Weighted Least Connections** (LL-WLC) policy is an adaptation of the LL policy where tasks are assigned to a dynamically ordered list of servers with a static ratio. The order of the servers is based upon the least amount of active connections (as a percentage of total capacity) during an observation period.

The **Least HTTP Sessions** (LL-SESSIONS) policy assigns a request to a server with the least amount of active HTTP sessions during an observation period.

For the purposes of this study, LL-SESSIONS is considered as a duplication of the Least Connections policy, since a performance-testing tool would receive a new session identifier (as a HTTP response header) for each connection that it establishes.

The **Fastest Response Time** (FRT) policy assigns a task to a server that has achieved the fastest response time during an observation period.

The **Weighted Round-Robin** (WRR) policy, as described by Hunt et al. (1998), is a dynamic variation of the static Weighted Round-Robin policy, and assigns a request to a server based on a dynamically evaluated server weight (load) that is proportional to the load information that each server has reported during an observation period, and includes many of the same variations as the LL policy.

Server state-aware policies attempt to address the availability, fail-over, performance (and in some cases, the reliability) concerns of state-blind policies, at the expense of additional processing overheads (waiting time), and represents the group of policies that may provide the best results in a clustered Java EE environment where capacity planning can become challenging.

A study by Andreolini, Colajanni and Morselli (2002), however, suggested that the state-blind Round-Robin policy performed more consistently than server state-aware policies, as the periodic update nature of load status reports often resulted in server saturation before the next update interval, and introduces a valid and significant concern. (Load feedback sample age problems were examined in Chapter 2.)

3.2.4 Client and Server state-aware policies

The **Client Affinity** policy is able to maintain client affinity based on previous server selection decisions with the initial sever selection based on load information.

This policy is best described through the examples that were presented in Cardellini et al. (2002, p.290), where SSL-based services were considered. The initial client request, before an SSL handshake is completed, would be routed to a server based on the amount of load, where the selected sever would create an SSL session and process the initial request.

It would thus be beneficial that subsequent requests are also routed to the same sever in order to realise the performance benefits of reusing the same SSL session. Support for HTTP/1.1 persistent connections would be another good example of this policy.

The primary disadvantage of this approach is that client-affinity rules ignore subsequent server load state information after the initial request, and would thus exhibit the same concerns as state-blind policies.

3.2.5 The F5 Networks Dynamic Ratio policy

The (patent pending) F5 Dynamic Ratio policy, based upon the server state-aware WRR policy, assigns a dynamic weight to each server based on CPU, memory and disk I/O utilization metrics that are obtained from SNMP-enabled servers.

Weight of Node =

(Number of Nodes in Pool) ^ (Memory Coefficient ((Memory Threshold - Memory Utilization) / Memory Threshold))

+ (Number of Nodes in Pool) ^ (CPU Coefficient ((CPU Threshold - CPU Utilization) / CPU Threshold))
+ (Number of Nodes in Pool) ^ (Disk Coefficient ((Disk Threshold - Disk Utilization) / Disk Threshold))

Where:
- The minimum and maximum weight of a node is 1 to 100.
- Utilization metrics and thresholds are percentages.
- The 'Number of Nodes in Pool' is always defaulted to 10.
- Coefficients are the weights for each metric.
- Thresholds are user-defined values.

Equation 1 - F5's Dynamic Ratio algorithm. (F5 Networks, 2010)

Node 1				
Number of Nodes in Pool	**Memory Coefficient**	**Memory Threshold**	**Memory Utilization**	***Total***
10	1	70%	20%	**5.18**
Number of Nodes in Pool	**CPU Coefficient**	**CPU Threshold**	**CPU Utilization**	
10	1.5	80%	20%	**13.34**
Number of Nodes in Pool	**Disk Coefficient**	**Disk Threshold**	**Disk Utilization**	
10	2	90%	20%	**35.94**
			Node 1 weight	**54**
Node 2				
Number of Nodes in Pool	**Memory Coefficient**	**Memory Threshold**	**Memory Utilization**	***Total***
10	1	70%	40%	**2.68**
Number of Nodes in Pool	**CPU Coefficient**	**CPU Threshold**	**CPU Utilization**	
10	1.5	80%	40%	**5.62**
Number of Nodes in Pool	**Disk Coefficient**	**Disk Threshold**	**Disk Utilization**	
10	2	90%	40%	**12.92**
			Node 2 weight	**21**

Table 1 - Example application of the F5 DR algorithm.

As per the calculation in Table 1, a connection ratio of 54:21 will be calculated, where Node 1 will receive 2.6 (~3) times as many connections as Node 2 in a cyclic round-robin order.

3.2.6 F5 LTM as the leading commercial load-balancing solution

The F5 Networks LTM is rated as the most comprehensive industry-leading commercial load balancing solution by Gartner (Gartner, 2009), as per Figure 8.

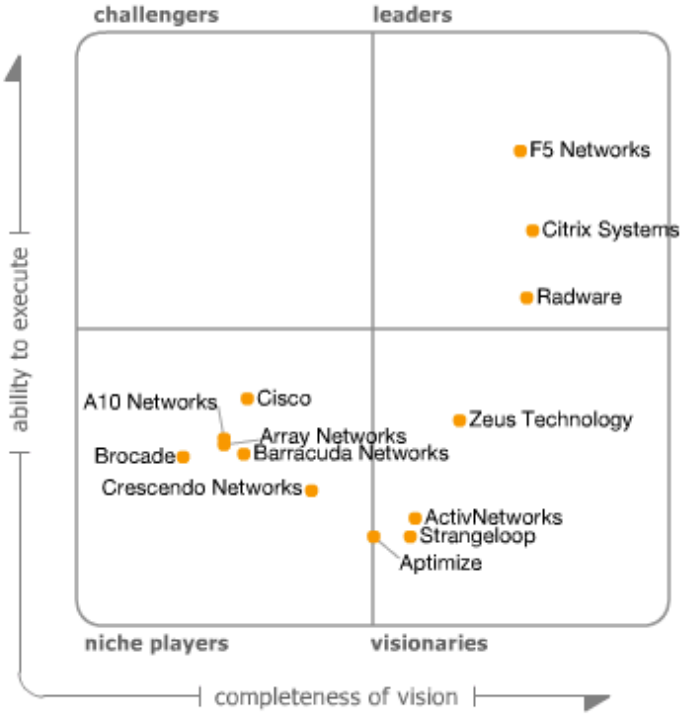


Figure 8 - Gartner's magic quadrant report for Application Delivery Controllers. (Gartner, 2010)

3.3 The consideration of Java EE platform-specific state information in task assignment decisions

The capacity and utilization of several core Java and Java EE resources and services will have a direct impact on the availability, scalability, reliability and performance that Java EE Web application deployments are able to achieve.

These core resources include (but are not limited to):

1. JVM Heap Space,

2. HTTP connection queue sizes and queue worker threads,

3. and JDBC database connection threads.

3.3.1 JVM Heap Space

JVM Heap Space defines the maximum amount of OS memory that has been reserved for use by a JVM process at runtime. Each Application Server will thus have a fixed amount of memory within which it must provide all Java EE services, containers and deployed Web applications.

Exceeding the amount of available Heap Space will result in *java.lang.OutOfMemoryException*'s, where the JVM process will halt.

The scalability and performance of JVM's are also impacted by garbage collection cycles and strategies, where the relative utilization of heap space may trigger garbage collections quite often in order to reclaim the memory that was used by orphaned objects, resulting in JVM delays.

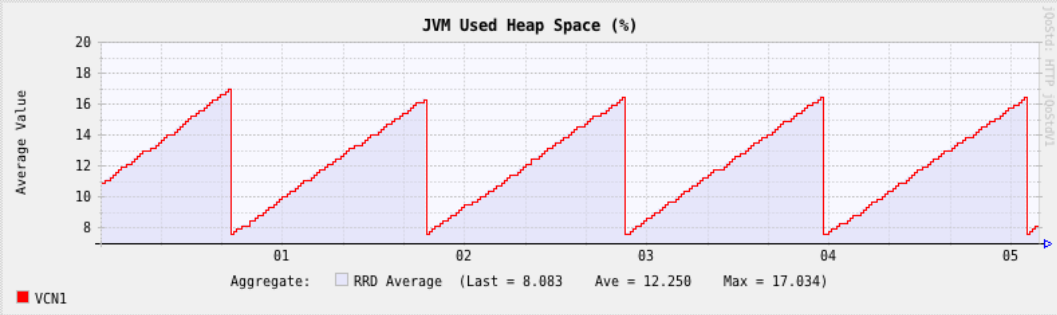


Figure 9 - Example of JVM garbage collections with low heap space utilization.

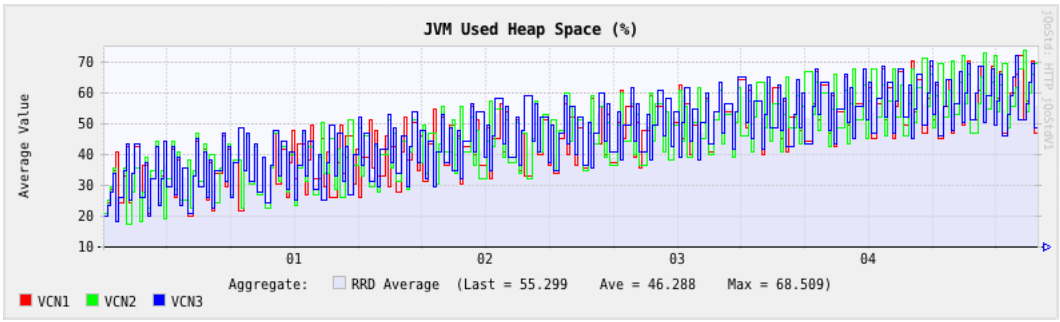


Figure 10 - Example of high-frequency JVM garbage collections with high heap space utilization.

An application server that is not experiencing high heap space utilization, as illustrated in Figure 9, may perform garbage collections much less often than compared to application servers with higher heap utilization, as per Figure 10.

3.3.2 Application Server HTTP connection request queue, worker threads and JDBC connection pools

Application Server Web Containers manage HTTP client connections asynchronously with HTTP connection listeners and connection processing queues that are processed by a limited number of queue processing (worker) threads.

The configuration of an Application Server defines the maximum number of client connections (connection queue size), and available worker threads.

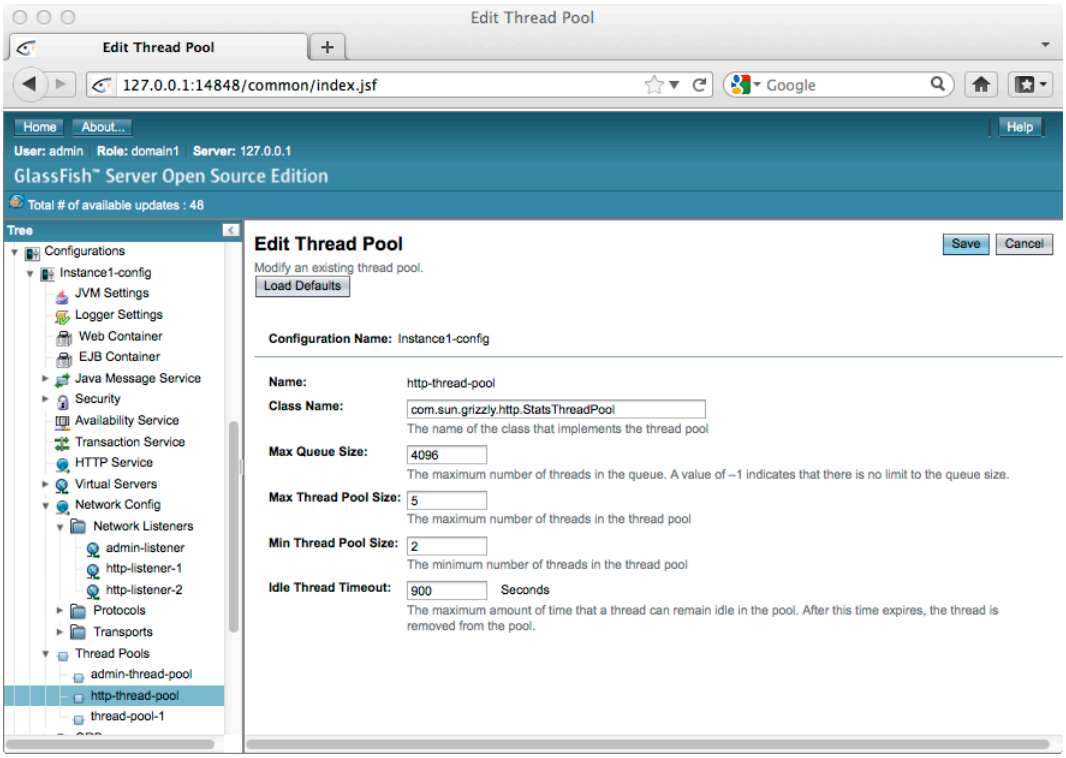


Figure 11 - The default "http-thread-pool" configuration of a node in GlassFish.

In GlassFish Server, for example, the default configuration will accept a maximum of 256 requests in HTTP keep-alive client connections, and will queue a maximum of 4096 connections. The connection queue, however, is processed with a default maximum of 5 worker threads, possibly leading to a large amount of connection queuing in high-utilization situations that would increase latency.

A shortage of worker threads for new HTTP connections will result in connection queuing whenever all of the worker threads are busy, and may therefore result in denied connections once the maximum connection queue size is exceeded.

Also important is the consideration of requests in keep-alive connections, where a client may pipeline multiple requests (as per HTTP\1.1) in an es-

tablished connection. The amount of permitted requests in keep-alive connections may result in an increased number of connections being made.

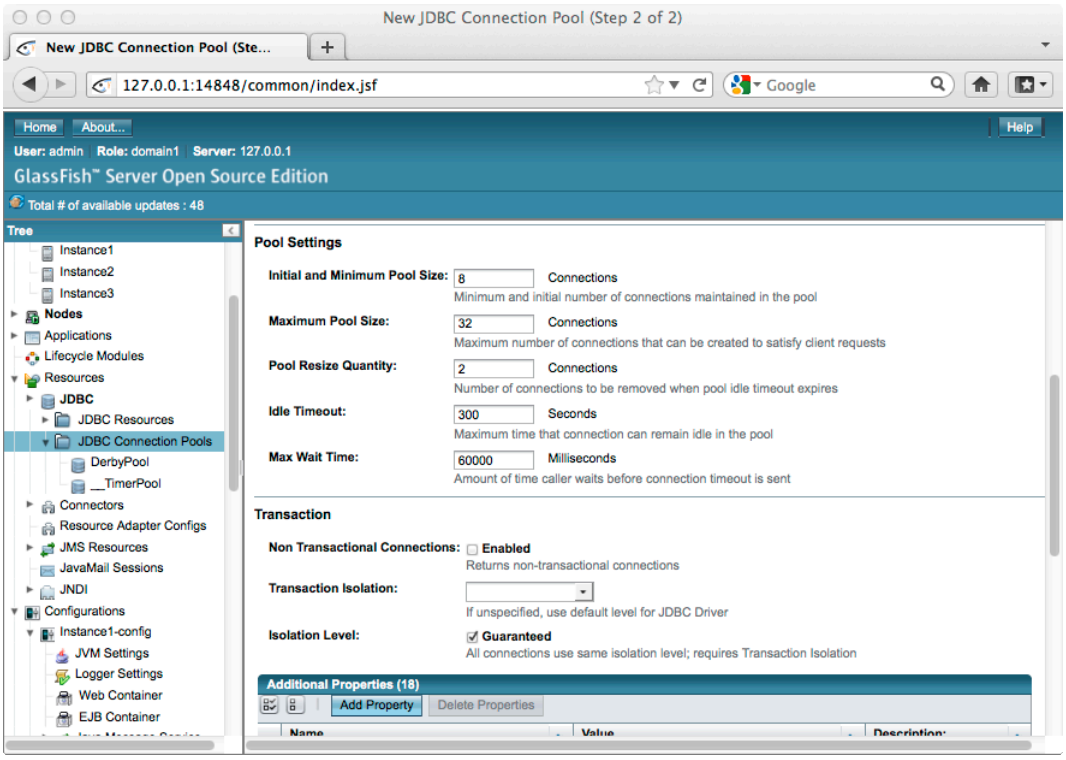


Figure 12 - The default JDBC Connection Pool settings for a new database connection pool.

JDBC database connection pools would present the same challenges as the HTTP connection queue worker threads. In Glassfish a new JDBC connection pool is defaulted to 32 connection threads, as per Figure 12.

3.3.3 Capacity planning tools

Capacity planning for these core Java EE resources can become a very complex and challenging task, and may lead to substantial investments in performance benchmarking, monitoring and analysis tools, depending on the size of the problem domain.

Several vendors, such as Foglight (Quest Software, 2012), for example, have specialised Application Performance Monitoring (APM) products that are able to monitor and instrument the utilization of these core Java EE and other server and network (etc.) resources, without which capacity planning may become a costly trial and error exercise.

3.4 Summary

The application of an effective task assignment policy, as part of a load balancing solution for Java EE, should include the consideration of core Java EE resources (and their peak demands, etc.) in order to support the realization of several non-functional quality requirements that are expected of E-business Web applications on the Internet.

Chapter 4. SPECIFICATION AND DESIGN

4.1 Software Requirements Specification

4.1.1 Introduction

4.1.1.1 Purpose

This chapter enumerates and describes the requirements for the Task Dispatcher (outcome 3) and the policies that it will provide (outcome 4) for the first release (prototype) of the software product as required to realise all of the software-based outcomes of this dissertation.

4.1.1.2 Product scope

The prototype task dispatcher and its task assignment policies address a problem domain where current industry-leading “load balancing” solutions, such as the F5 LTM, may be unable to provide optimal task assignments in Java EE due to their inability to consider Operating System, Application Server and JVM metrics that may be required to perform optimal task assignments in E-business clustered Java EE production environments (as discussed in Chapter 3).

4.1.2 Overall Description

4.1.2.1 Product functions

As per outcomes 3, 4 and 6 of this research, the software product shall:

- Provide a TCP (OSI Layer-4) and HTTP (OSI Layer-7) Java NIO reverse-proxy server task dispatcher, with implementations of the

task assignment policies as identified on page 15, in order to support the evaluation of these policies.

- Propose and provide an implementation of a new server state-aware task assignment policy that considers Operating System, Application Server and JVM metrics in order to produce optimal dynamic task assignments for clustered Java EE Web applications.

- Collect and report OS, Application Server and JVM metrics through Web-based monitoring interfaces.

- Provide administrators with a Web-based user interface to control and configure the software with.

4.1.2.2 User classes and characteristics

The software is intended to become an Open Source solution that can be used by Java developers and other technical support staff, such as Solution Architects and Network\Server Administrators.

As such, the software is intended for use by end-users that already possess technical knowledge and experience within the problem domain that the software addresses.

4.1.2.3 Operating environment

The task dispatcher (outcome 3) shall:

- Only support a single active instance of a Virtual Cluster at runtime.

- Run on any Operating Systems that are capable of running the Java Standard Edition version 1.5+ Java Virtual Machine and associated Java Runtime Environment.

- Support the use of static task assignment policies for the TCP (Layer-4) task dispatcher implementation without a dependency on the Node Monitoring Agent or Java EE Application Servers.

- Require the use of the Node Monitoring Agent, and Java EE Application Servers for all dynamic HTTP (Layer-7) policies that are implemented in the HTTP (Layer 7) task dispatcher implementation.

The Node Monitoring Agent shall:

- Require installation onto a Java EE version 5+ compliant Application Server, with support for GlassFish (v3.1.1) in the initial version.

- Only support Operating Systems as per the SIGAR API's official list of supported Operating Systems. (Sigar, 2012)

As described in section 2.1.2, the software, when configured as a HTTP (Layer-7) task dispatcher, shall require a clustered Java EE Web Application environment for deployed applications whenever session-stickiness is required.

Support for Java EE Application Server clustering is a vendor-based add-on, and as a result, the task dispatcher will not be able to verify, or interact with the clustering configuration, and shall not be dependant on this in order to provide implementations of the task assignment policies.

4.1.2.4 User documentation

The task dispatcher shall:

UD-01: Include installation and initial configuration documentation as HTML files.

UD-02: Include user-friendly context sensitive validations and configuration guidance as part of configuration sections in the task dispatcher's Web-based user interface.

4.1.3 Functional requirements

These sections detail the main features of the system by listing the actions to be taken and defining the requirements for those features.

4.1.3.1 Task dispatcher requirements (TDR)

The software shall:

TDR-01: Provide a Java NIO reverse proxy server with support for TCP (Layer-4) and HTTP (Layer-7) task assignment policies, as per the list of task assignment policies and respective requirements that are defined in subsequent sections.

TDR-02: Only support HTTPS connections with TCP (Layer 4) task assignment policies.

TDR-03: Provide a secure Web-based control, configuration and monitoring user interface.

TDR-04: Provide logging of all events to text files, with support for the configuration of logging detail levels (i.e. trace, debug, information, warnings and errors).

TDR-05: Collect the following performance metrics:

Metric Name	Description
TD_AVE_CONNS_ACT	Active Client Connections per second
TD_AVE_RQSTS_ACT	Active Client Transactions per second
TD_AVE_RSPS_TIME	Latency: Average response time for completed transactions (ms) per second
TD_AVE_THROUGHPUT	Throughput: Completed Transactions per second
TD_AVE_FAIL_RQSTS	Failed Transactions per second
TD_AVE_SRV_SELS	Number of Node Selections per second
TD_AVE_SRV_SEL	Waiting Time (WT1): Average Node Selection Time (ms) per second
TD_AVE_SRV_CONN_WAIT	Waiting Time (WT2): Average Node Connection Time (ms) per second
TD_AVE_RQ_PROCTIME	Processing Time (PT): Average Transaction Running Time (ms) per second
TD_AVE_SLOWDOWN	Slowdown: (WT1 + WT2 + PT) / PT
TD_AVE_TRAF_IN	Traffic Received (MB/s)
TD_AVE_TRAF_OUT	Traffic Sent (MB/s)

Table 2 - Metrics collected by the Task Dispatcher.

4.1.3.2 Task dispatcher configuration requirements (TDCR)

Configuration overview

A Virtual Cluster (VC) defines a listening IP Address and Port (listening interface) for a clustered service that will receive incoming connections from clients.

A VC is composed of one or more Virtual Cluster Nodes (VCN's), each providing a destination IP Address and Port (destination interface) that will receive incoming traffic from the listening interface, and through the use of a Task Assignment Policy (TAP), will select the destination interface for each client connection.

Each destination interface then provides a response for the incoming traffic, which is then returned, via the task dispatcher, to the client's connection on the listening interface.

TDCR-01: Each Virtual Cluster shall provide a listening IP Address and Port, as well as one or more destination IP Addresses and Ports, that are to be defined as Virtual Cluster Nodes.

Each Virtual Cluster must require the selection of a Task Dispatching Service Type (i.e. TCP Layer 4 or HTTP Layer 7), as well as the selection of a Task Assignment Policy that is supported for the selected Task Dispatching Service Type.

TDCR-02: Virtual Cluster configurations shall be managed at runtime by an instance of the selected Task Assignment Policy in order to switch incoming client traffic from the Virtual Cluster's listening interface to a selected destination Virtual Cluster Node interface.

An example TCP reverse proxy configuration:

A VC is configured with a listening interface 192.168.0.1:21, with two VCN's 192.168.0.2:21 and 192.168.0.3:21. In this example, each VCN represents an instance of a mirrored FTP server, each providing identical content.

A Round-Robin Task Assignment Policy (for example) is then applied to dispatch the incoming traffic for each client's connection to the VC's listening interface on 192.168.0.1:21, to the Virtual Cluster Nodes 192.168.0.2:21 and 192.168.0.3:21 respectively, where each subsequent

client's connection on the listening interface (VC) is assigned to a different destination interface (VCN) for processing – thus achieving load balancing.

TDCR-03: TCP (Layer-4) task assignment's shall be performed on a per client connection basis.

An example clustered Web application reverse proxy configuration:

A Virtual Cluster is configured with a listening interface 192.168.0.1:80, with two Virtual Cluster Nodes 192.168.0.2:80 and 192.168.0.3:80. Each Virtual Cluster Node represents a node from a clustered (session-replicated) Java EE Web Application.

A static Weighted Round-Robin Task Assignment Policy is then (for example) applied to switch the incoming HTTP connections for each client from the VC listening on 192.168.0.1:80 to the cluster nodes 192.168.0.2:80 and 192.168.0.3:80, where client connections (and thus their traffic) on the listening interface (VC) are assigned to each destination VCN based upon the static "weight" connection ratio parameter that was configured for each respective VCN.

TDCR-04: HTTP (Layer-7) task assignments shall be performed on a per connection basis, where a new destination VCN for a client connection will be selected when a server (VCN) connection is closed, or a client time-out occurs.

4.1.3.3 Task dispatcher User Interface requirements (TDUIR)

The User Interface shall:

TDUIR-01: Provide a Web-based configuration management and reporting interface.

TDUIR-02a: Provide a configuration option for the definition a Virtual Cluster, where a unique Name, valid IP Address, valid Port field, Task Assignment Policy, as well as one or more Virtual Cluster Nodes can be defined for both TCP and HTTP VC's.

TDUIR-02b: Provide a configuration option for the definition of Virtual Cluster Nodes, where a unique Name, valid IP Address, valid Port and connection weight ratio fields will be captured.

TDUIR-03: Provide an operational status and reporting interface where various metrics for the Task Dispatcher, Virtual Cluster and associated Virtual Cluster Nodes can be viewed as time-series graphs at runtime.

4.1.3.4 Quality of Service requirements (QoSR)

The task dispatcher shall:

QoSR-01: Close an inactive client's TCP connection (NIO handler channel) after 30 seconds of idle time.

4.1.3.5 Task assignment policy requirements (TAPR)

The Task Dispatcher shall:

Static policies

TAPR-01: Provide an implementation of the static Random policy, as per section 3.2.1.

TAPR-02: Provide an implementation of the static Round-Robin policy, as per section 3.2.1.

TAPR-03: Provide an implementation of the static Weighted Round-Robin policy, as per section 3.2.1, where 'weight' is defined as the number of connections that each Virtual Cluster Node will receive.

Dynamic policies (with a dependency on the Node Monitoring Agent)

TAPR-04a: Provide a thread-safe Node Monitoring Service that is able to retrieve NMA metrics from each Virtual Cluster Node that is managed by dynamic policies.

TAPR-04b: Obtain serialized NMA metrics every 1 second from each VCN's NMA.

TAPR-05: Set the 'healthy' indicator of a VCN to false, thereby excluding it from task assignments, when NMA metrics cannot be obtained and attempt to update the health status of the VCN every 60 seconds thereafter.

TAPR-06: Provide an implementation of the static Weighted Round-Robin policy, as per section 3.2.3, where 'weight' is defined as the number of HTTP connections that a Virtual Cluster Node would receive.

TAPR-07: Provide an implementation of the Least Loaded policy, as per section 3.2.3, where load is defined as the percentage of CPU utilization on a VCN.

TAPR-08: Provide an implementation of the Fastest Response Time policy, as per section 3.2.3, where response time is defined as the average amount of running time a Virtual Cluster Node has taken to process HTTP requests over the last second.

TAPR-09: Provide an implementation of the Least Connections policy, as per section 3.2.3, where connections are defined as the number of active HTTP connections to a VCN.

TAPR-10: Provide an implementation of the Weighted Least Connections policy, as per section 3.2.3, where Connections are defined as the active number of HTTP connections to a VCN.

TAPR-11: Provide an implementation of the F5 Dynamic Ratio policy, as per section 3.2.5, where the metrics are based upon the following Node Monitoring Agent metrics: CPU Utilization = OS_CPU_USED%, Memory Utilization = OS_MEM_USED% and Disk I/O Utilization = OS_CPU_IOWAIT%.

4.1.3.6 jQoStdV1 policy requirements (JPR)

The jQoStdV1 policy shall:

JPR-01: Provide a Java EE server state-aware task assignment policy for HTTP (Layer-7) requests, where requests are dispatched to a Virtual Cluster Node with the greatest amount of available capacity (as per JPR-02).

JPR-02: Apply the following server load calculation for each Virtual Cluster Node, and assign each new client HTTP connection to the VCN

with the least amount of load, excluding a VCN from assignment if the load is calculated as 100:

```
SET serverLoad = (OS_CPU_USED% + OS_CPU_IOWAIT% + AMX_CONNS_Q_USED% +
AMX_CONNS_Q_THRBSY% + AMX_JDBC_ACTC% + JVM_HEAP_USED%) / 6

IF (AMX_CONNS_Q_USED% > 55) THEN SET serverLoad = 100
IF (JVM_HEAP_USED% > 85) THEN SET serverLoad = 100

Note:
The uppercased variables are the respective metrics collected by the Node Monitoring Agent for each VCN.
```

Equation 2 – jCoStdV1 policy VCN load calculation.

4.1.3.7 Node Monitoring Agent requirements (NMAR)

The Node Monitoring Agent shall:

NMAR-01: Collect metrics from Operating Systems, Application Servers, and Java Virtual Machines each time the agent is called.

NMAR-02a: Provide the collected metrics through a Java Server Page that outputs the collected metrics as a serialized *Hashtable<String, Double>*.

NMAR-02b: Provide the collected metrics (for debugging) through a Java Server Page that outputs the collected metrics as an HTML table.

NMAW-03a: Be deployable onto an Application Server (that hosts a Virtual Cluster Node) as a standalone Web Application deployment archive (i.e. a WAR file).

NMAW-03b: Provide a JAR library that contains an annotated Server Filter that inserts a base64 encoded serialized *Hashtable<String, Double>* of the collected metrics into the header of each HTML response that is returned by a Web Application every 100 milliseconds, for collection by the HTTP task dispatcher.

NMAW-04: Collect the following Operating System metrics through the use of the SIGAR API (Sigar, 2012):

Metric Name	Description
OS_LOAD_MAX	OS Maximum Load (#CPUs)
OS_LOAD_AVE1	OS 1-Min Load Average
OS_LOAD_AVE5	OS 5-Mins Load Average
OS_LOAD_AVE15	OS 15-Mins Load Average
OS_CPU_AVAIL%	OS Available CPU (%)
OS_CPU_IDLE%	OS CPU Idle (%)
OS_CPU_IOWAIT%	OS CPU IOWait (%)
OS_CPU_SYSTEM%	OS Kernel-processes CPU Utilization (%)
OS_CPU_USED%	OS CPU Utilization (%)
OS_CPU_USER%	OS User-processes CPU Utilization (%)
OS_DISK_QUEUE	OS Disk-subsystem Queue Size
OS_DISK_READ_BYTES	OS Disk-subsystem Active Reads (MB)
OS_DISK_READS	OS Disk-subsystem Active Reads
OS_DISK_WRITE_BYTES	OS Disk-subsystem Active Writes (MB)
OS_DISK_WRITES	OS Disk-subsystem Active Writes
OS_MEM_FREE	OS Free Physical System Memory (MB)
OS_MEM_FREE%	OS Free Physical System Memory (%)
OS_MEM_TOTAL	OS Total Physical System Memory (MB)
OS_MEM_USED	OS Used Physical System Memory (MB)
OS_MEM_USED%	OS Used Physical System Memory (%)
OS_SWAP_FREE	OS Free Virtual Memory (MB)
OS_SWAP_FREE%	OS Free Virtual Memory (%)
OS_SWAP_PAGEINS	OS Virtual Memory Page-ins (MB)
OS_SWAP_PAGEOUTS	OS Virtual Memory Page-outs (MB)
OS_SWAP_TOTAL	OS Total Virtual Memory (MB)
OS_SWAP_USED	OS Used Virtual Memory (MB)
OS_SWAP_USED%	OS Used Virtual Memory (%)
NET_TCP_ACT_OPENS	OS Network-subsystem TCP ACTIVE-OPENS
NET_TCP_ATT_FAILS	OS Network-subsystem TCP ATTEMP-FAILS
NET_TCP_CUR_ESTAB	OS Network-subsystem TCP CURR-ESTAB
NET_TCP_ESTAB_RESETS	OS Network-subsystem TCP ESTAB_RESETS
NET_TCP_IN_ERRS	OS Network-subsystem TCP IN-ERRS
NET_TCP_IN_SEGS	OS Network-subsystem TCP IN-SEGS
NET_TCP_OUT_RSTS	OS Network-subsystem TCP OUT-RSTS
NET_TCP_OUT_SEGS	OS Network-subsystem TCP OUT-SEGS
NET_TCP_PASV_OPENS	OS Network-subsystem TCP PASSIVE-OPENS
NET_TCP_RETRANS_SEGS	OS Network-subsystem TCP RETRANS-SEGS

Table 3 - Operating System metrics collected by the Node Monitoring Agent.

NMAW-05: Collect the following Java EE Application Server metrics through the use of the Java Management Extension's (JMX) interface, when and as supported by the Application Server implementation:

Metric Name	Description
AMX_CONNS_OPEN	AppServer Open Connections
AMX_CONNS_MAX	AppServer Maximum Connections
AMX_CONNS_USED%	AppServer Maximum Connections
AMX_CONNS_DENIED	AppServer Denied Connections
AMX_CONNS_KEEPALVMAX	AppServer Max Requests in each Keep-alive Connection
AMX_CONNS_QD	AppServer Queued Connections
AMX_CONNS_QD_1MAVE	AppServer 1-Min Queued Connections
AMX_CONNS_QD_5MAVE	AppServer 5-Mins Queued Connections
AMX_CONNS_QD_15MAVE	AppServer 15-Mins Queued Connections
AMX_CONNS_Q_MAXSIZE	AppServer Maximum Connection Queue Size
AMX_CONNS_Q_USED%	AppServer Connection Queue Used %
AMX_CONNS_Q_THRDSMAX	AppServer Connection Queue Maximum Threads
AMX_CONNS_Q_THRDSMIN	AppServer Connection Queue Minimum Threads
AMX_CONNS_Q_THRBSY	AppServer Connection Queue Busy Threads
AMX_CONNS_Q_THRBSY%	AppServer Connection Queue Thread Utilization (%)
AMX_RQST_ERRS	AppServer HTTP Request Errors
AMX_RQST_MAXTIME	AppServer HTTP Request Maximum Processing Time (ms)
AMX_RQST_AVETIME	AppServer HTTP Request Average Processing Time (ms)
AMX_RQST_TOTAL	AppServer HTTP Requests
AMX_SESSIONS_ACT	AppServer HTTP Active Sessions
AMX_JDBC_MAXC	AppServer Max JDBC Connections
AMX_JDBC_ACTC	AppServer Active JDBC Connections
AMX_JDBC_ACTC%	AppServer Used JDBC Connections (%)

Table 4 - Application Server metrics collected by the Node Monitoring Agent.

NMAW-06: Collect the following Java Virtual Machine metrics from the Application Server's JVM:

Metric Name	Description
JVM_CLASS_COUNT	JVM Loaded Classes
JVM_COMPILE_TIME	JVM Total Compile Time (ms)
JVM_GC_TIME	JVM Total GC Time (ms)
JVM_HEAP_AVAIL	JVM Available Heap Space (MB)
JVM_HEAP_AVAIL%	JVM Available Heap Space (%)
JVM_HEAP_TOTAL	JVM Total Heap Space (MB)
JVM_HEAP_USED	JVM Used Heap Space (MB)
JVM_HEAP_USED%	JVM Used Heap Space (%)
JVM_THREADS	JVM Threads
JVM_THREADS_DAEMONS	JVM Daemon Threads

Table 5 - Java Virtual Machine metrics collected by the Node Monitoring Agent.

NMAW-07: Collect the following metrics for each server state-aware task assignment policy:

Metric Name	Description
NODE_NMA_RSPS_TIME	NMA: Response Time (ms)
NODE_AVAILABILITY%	NMA: Node Availability (%)
TAP_NODE_LOAD%	Task Assignment Policy: Calculated node load (%)
TAP_NODE_SEL	Task Assignment Policy: Number of Node selections
TAP_RQST_LASTTIME	Task Assignment Policy: Last HTTP Request Processing Time (ms)

Table 6 - Task Assignment Policy metrics collected by the task dispatcher.

4.1.4 Non-functional Requirements

4.1.4.1 Unit testing requirements (UTR)

The following JUnit4 tests shall be provided:

UTR-01: Each task assignment policy must have a corresponding unit test that validates the implementation of the policy.

4.1.4.2 Performance requirements (PR)

The software shall:

PR-01: Use Java NIO in the task dispatcher implementation.

PR-02: Provide a configurable event logging service, in order to reduce processing I/O and storage demands for the logging of system events.

4.1.4.3 Security requirements (SR)

The software shall:

SR-01: Require user authentication in order to prevent unauthorised access to administration console functions.

4.1.4.4 Software quality attributes (SQAR)

The software shall:

SQAR-01: Support the Operational Environment as per the requirements above.

SQAR-02: Provide User Documentation as per the requirements above.

SQAR-03: Follow the latest Java coding style guidelines, with JavaDoc source documentation, as published by Oracle. (Reddy, 2000)

4.1.4.5 Business rules (BR)

The software shall:

BR-01: Be implemented with the Java programming language.

4.1.4.6 Intellectual Property (IP)

The software shall:

IP-01: Abide by international laws for all intellectual property rights and adhere to the instructions of the respective IP owners when their intellectual property is included as part of the software product.

IP-02: Acknowledge the contributions of other entities that were used in the software product, as part of the user documentation.

IP-03: Only include Intellectual Property and other materials from Open Source Software or other initiatives that do not impose incompatible licencing conditions, distribution limitations, incur licencing costs, or any other form of royalties when distributed as part of Open Source software products.

4.2 Performance evaluation experiment design

4.2.1 Introduction

The study will perform an evaluation of the implemented policies with a workload specification and 5 different application server capacity configuration scenarios that will reduce the capacity for one the of servers in each experiment as per the table below:

Experiment Scenario	Server A	Server B	Server C
Baseline	Full capacity	Full capacity	Full capacity
JVM Heap Space	Full capacity	Full capacity	1/2 JVM Heap Space
HTTP connection queue worker threads	Full capacity	Full capacity	1/2 HTTP worker threads
JDBC threads	Full capacity	Full capacity	1/2 JDBC threads
All scenarios	Full capacity	Full capacity	1/2 JVM Heap Space 1/2 HTTP worker threads 1/2 JDBC threads

Table 7 - Experiment scenarios.

Every policy will be evaluated in each of the experiments, and an analysis of the results will then be performed.

4.2.2 Workload specification

The Apache DayTrader (Apache Geronimo, 2010) Java EE performance benchmarking application will be used as the Java EE Web application in the experiments as part of the workload specification.

"The Apache Software Foundation® DayTrader Performance Benchmark Sample provides a suite of workloads for characterizing performance of J2EE 1.4 Application Servers. The workloads consist of an end-to-end Web application and a full set of Web primitives. The applications are a collection of Java™ classes, Java servlets, Java ServerPages™ (JSP™) files and Enterprise JavaBeans™ (EJB™) built to open Java 2 Platform, Enterprise Edition (J2EE™) APIs. Together, the Trade application and Web primitives provide versatile and portable test cases that are designed to measure aspects of scalability and performance." (Apache Geronimo, 2010)

"DayTrader is an end-to-end Web application that is modeled after an on-line stock brokerage. DayTrader leverages J2EE components such as servlets, JSP files, enterprise beans, message-driven beans (MDBs) and Java database connectivity (JDBC™) to provide a set of user services such as login/logout, stock quotes, buy, sell, account details, and so on through standards-based HTTP and Web services protocols." (Apache Geronimo, 2010)

DayTrader includes a scenario servlet (workload driver) that automatically cycles through randomised DayTrader use case scenarios each time that it is called, as illustrated in Figure 13.

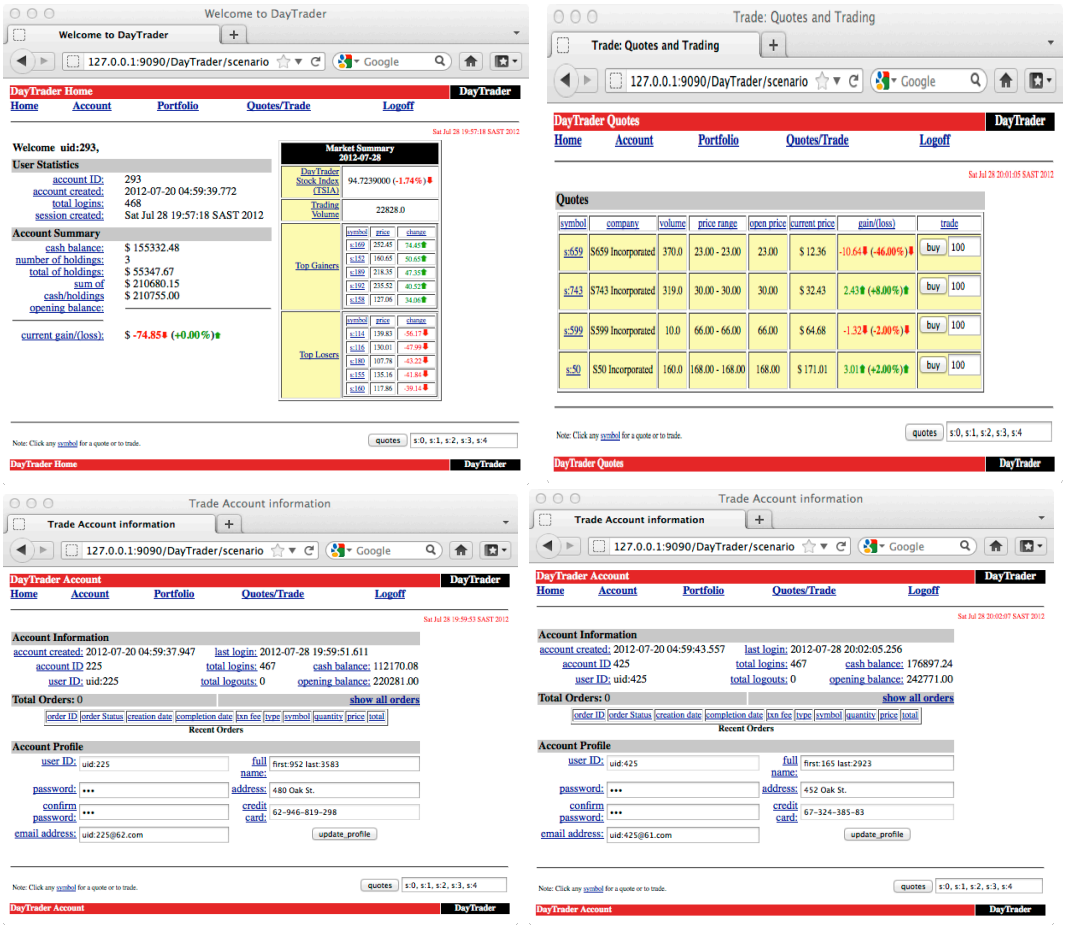


Figure 13 – Examples of DayTrader scenario servlet use cases, selected randomly each time that the servlet is called.

The workload specification will be implemented as a jMeter (Apache jMeter, 2012) test plan, as per Figure 14.

The DayTrader scenario servlet URL will be retrieved through the task dispatcher as the destination URL of the jMeter HTTP Request Sampler that is used in each of the sampler threads.

The workload specification will reset the DayTrader database at the start of each test, and will then call the http://127.0.0.1:9090/DayTrader/scenario servlet (continuously) from a maximum of 30 sampler threads, over a 5-minute time period.

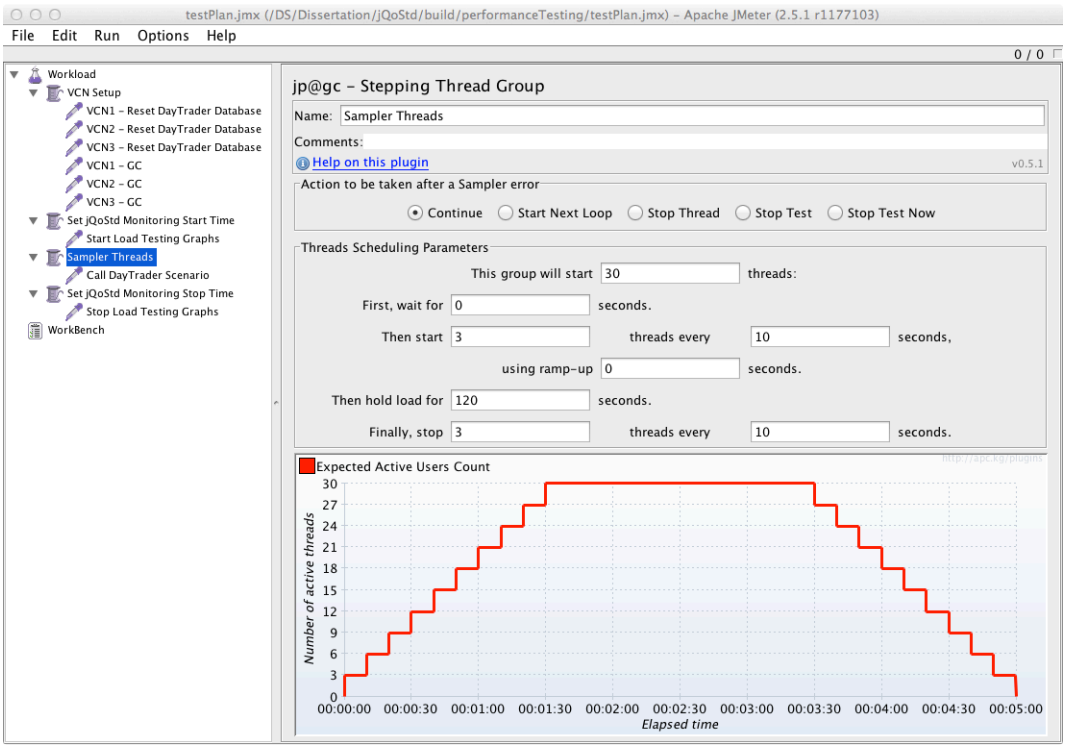


Figure 14 – The jMeter test plan and workload sampler threads specification.

4.2.3 Testing results and data

The Task Dispatcher shall collect the following metrics (as statistics and time series graphs) during the execution of the workload model for each task assignment policy, in each experiment:

Metric	Description
Samples	The total number of HTTP requests that were received by the task dispatcher during the test.
Active Conns	The average number of active client HTTP connections per second.
Throughput	The average number of completed HTTP requests per second.
Failed	The total number of failed HTTP requests during the test.
Failed %	The percentage of Samples that have failed.
Latency	The average latency of HTTP requests.
WT1	The average waiting time (per HTTP request) for server selections made by the task assignment policy.
WT2	The average waiting time (per HTTP request) for opening connections to the server that was selected by the task assignment policy.
PT	The average processing time (per HTTP request) on the selected server.
Slowdown	The average amount of slowdown per second.
Node Sels	The average number of server selections performed per second.

Table 8 - Metrics collected in the task dispatcher during a test.

The Node Monitoring Service metrics, for HTTP (Layer-7) policies, must also be collected and stored as time series graphs.

4.2.4 Experiment design

All of the implemented task assignment policies will be tested in 5 different scenarios on a Virtual Cluster that is composed of 3 application servers.

The execution of experiments will consume a lot of time, and could introduce many unexpected problems, where the experiments may need to be repeated numerous times in a consistent and predictable form.

The automation of experiment runs with a scripting language will thus be required.

Each experiment will include the following activities:

1. The creation and installation of the application server configurations for each experiment scenario.

2. The deployment of the DayTrader.war and jQoStdNMA.war web archive files on each application server.

3. The creation and installation of the respective task assignment policy configuration in the task dispatcher.

4. The starting of the Java Derby DBMS, and application servers.

5. The starting of the task dispatcher process.

6. The execution of a jMeter test plan.

7. The collection of statistics and metric graphs from the task dispatcher.

8. The stopping of the task dispatcher, Derby DBMS, and application server processes.

The automation of the experiments shall be implemented in a platform-independent manner, preferably with a Java ANT script, as that would enable the replication of the experiments on other platforms.

Chapter 5. METHODS AND REALIZATION

5.1 Overview

The software product has been implemented as a Java NIO reverse proxy server named: Java Quality of Service Task Dispatcher (jQoStd).

The implementation includes TCP (Layer-4) content and server state-blind, and HTTP (Layer-7) content and server state-aware task dispatcher implementations and policies.

The TCP implementation represents the Apache mod_proxy solution, and the HTTP implementation represents the F5 LTM.

The jQoStd solution is composed of (Figure 15):

1. A Task Dispatcher (reverse proxy server) with a Task Assignment Policy, Node Monitoring Service, and a Virtual Cluster that defines one or more Virtual Cluster Nodes.

2. An Administration Console that enables authorised administrators to configure and control the software via an embedded Tomcat server and a set of configuration and metric-reporting JSP pages.

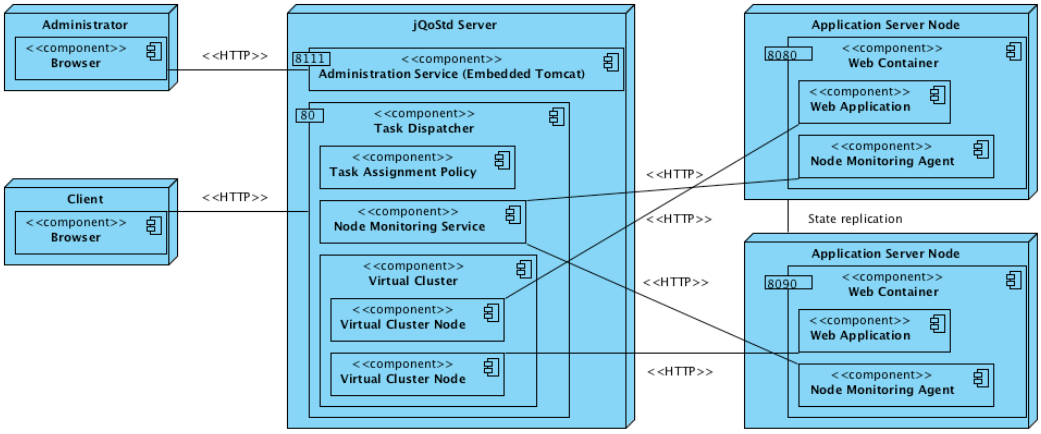


Figure 15 - Deployment diagram for the jQoStd HTTP task dispatching solution.

A Virtual Cluster represents a logical view of a clustered Java EE Web Application; where each Virtual Cluster Node represents a Java EE Application Server node that will participate in the virtual cluster as a virtual cluster node.

The Node Monitoring Service, only used for the HTTP (Layer-7) policy implementations, will obtain periodic samples of performance and utilization metrics from the Node Monitoring Agents that are installed on each of the Java EE Application Servers (Virtual Cluster Nodes).

These metrics will be used by the various HTTP (Layer-7) content and server state-aware task assignment policies in order to select the Virtual Cluster Node that will process HTTP requests from each client's connection to the task dispatcher.

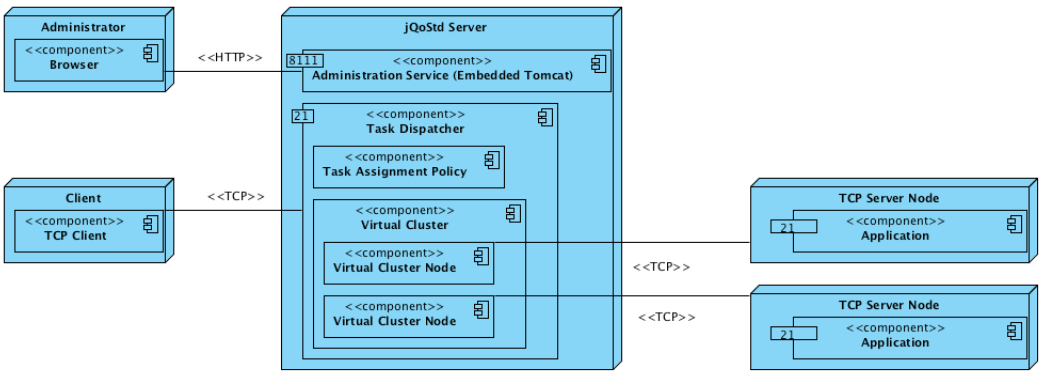


Figure 16 - Deployment diagram for the jQoStd TCP task dispatching solution.

JQoStd also provides a TCP (content-blind, server state-blind) task dispatcher implementation, as illustrated in Figure 16.

The Node Monitoring Agent(s) and Node Monitoring Service are not included in this solution, since these features are only supported with the HTTP content and server state-aware task assignment policies.

As such, the TCP implementation can be used to provide load balancing for any TCP-based service that uses a TCP port.

E.g. As per Figure 16, a load balancer has been created for a mirrored FTP service that is hosted on two back-end FTP servers, where jQoStd and the respective task assignment policies are not aware of the content or communications protocols being used or the server-state in the solution.

5.2 Design and implementation of the jQoStd TCP content and state-blind task dispatcher

The diagram in Figure 17 introduces the architectural pattern of a jQoStd TCP task dispatcher.

A TCP task dispatcher is only aware of the client-state (i.e. information that is available from a TCP connection) in its task assignment decisions.

As such, only static, content and server state-blind task assignment policies are supported.

The Apache HTTPD mod_proxy (Apache, 2012b) solution provides the Random, Round-Robin, and Static Weighted Round-Robin task assignment policies.

The jQoStd TCP task dispatcher implementation provides these policies as the:

- *RandomPolicy*,
- *RoundRobinPolicy*,
- and *StaticWeightedRoundRobinPolicy*

classes in the *org.symthoughts.jqostd.policies.tcp* package.

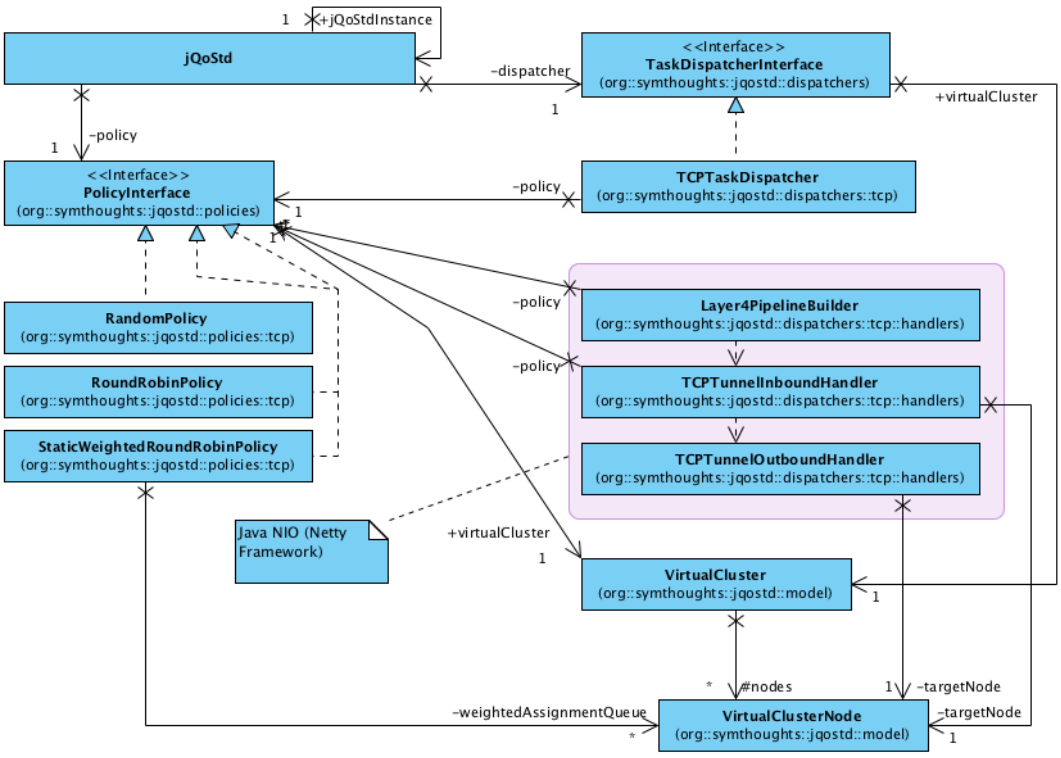


Figure 17 - Design overview of the jQoStd TCP task dispatcher.

The *jQoStd* class is the high-level (main) runtime class and assembles runtime instances of the *TaskDispatcherInterface* and *PolicyInterface* interface classes.

An instance of a *TaskDispatcherInterface* is required to manage the necessary sub-system classes that will provide the connection handling functionalities.

In jQoStd these functionalities are performed via Java NIO with the Netty NIO library (Netty, 2012). Netty's pipe and filter asynchronous event handling is configured with an event pipeline *Layer4PipelineBuilder* that will notify a connection handler *TCPTunnelInboundHandler* of client connection events that need to be processed.

Since jQoStd is a reverse proxy load balancer, an additional event handling class *TCPTunnelOutboundHandler* is required that will manage the communications to and from back-end servers (Virtual Cluster Nodes) for each client connection, as selected by the task assignment policy.

At runtime, a client thus establishes a connection (via Netty) to the *TCPTunnelInboundHandler*, which in turn applies a *PolicyInterface* instance to determine the destination server (a *VirtualClusterNode*) that will receive the traffic from the client's connection.

The *TCPTunnelOutboundHandler* is connected to the *TCPTunnelInboundHandler*'s pipeline, and will then establish a connection to the selected back-end server and stream traffic to and from the client's connection (via the pipeline) to the back-end sever connection.

All of the required runtime instances of the implementation classes are instantiated from the *jQoStd* class at runtime from an input XML configuration file that provides *jQoStd* with the necessary information to assemble the task dispatcher as a runtime instance.

```
<staticPolicyConfig>
    <configVersion>jQoStd Version 1.0.0</configVersion>
    <dispatcherClass>
         org.symthoughts.jqostd.dispatchers.tcp.TCPTaskDispatcher
     </dispatcherClass>
    <policyClass>
         org.symthoughts.jqostd.policies.tcp.StaticWeightedRoundRobinPolicy
     </policyClass>
    <virtualCluster>
       <name>FTP Load Balancer</name>
       <socket>
          <hostName>192.168.0.1</hostName>
          <port>21</port>
          <maxConnections>0</maxConnections>
       </socket>
       <nodes class="org.symthoughts.jqostd.model.VirtualClusterNode">
          <name>VCN1</name>
          <socket>
             <hostName>192.168.0.2</hostName>
             <port>21</port>
             <maxConnections>0</maxConnections>
          </socket>
          <weight>1</weight>
          <enabled>true</enabled>
       </nodes>
       <nodes class="org.symthoughts.jqostd.model.VirtualClusterNode">
          <name>VCN2</name>
          <socket>
             <hostName>192.168.0.3</hostName>
             <port>21</port>
             <maxConnections>0</maxConnections>
          </socket>
          <weight>1</weight>
          <enabled>true</enabled>
       </nodes>
    </virtualCluster>
</staticPolicyConfig>
```

Figure 18 - Example jQoStd XML configuration file for a TCP (Layer-4) FTP load balancer, configured with two nodes and the (TCP) Static WRR policy.

5.3 Design and implementation of the jQoStd HTTP content and server state-aware task dispatcher

The jQoStd HTTP (Layer-7) task dispatcher implementation repeats the architectural pattern illustrated above with some minor specializations that are required in order to support content and server state-aware task assignment policies.

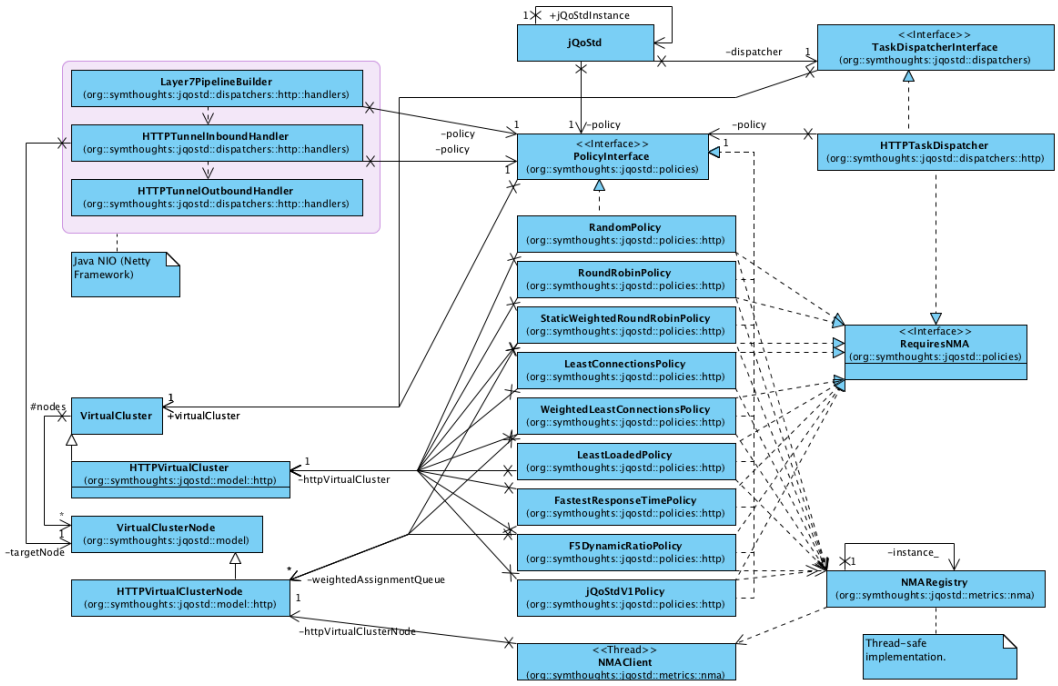


Figure 19 - Design overview of the jQoStd HTTP task dispatcher.

As per Figure 19, various implementation classes now have HTTP* specializations such as the *HTTPVirtualCluster*, *HTTPVirtualClusterNodes*, and the respective (Netty-based) NIO classes: *Layer7PipelineBuilder*, *HTTPTunnelInboundHandler*, and *HTTPTunnelOutboundHandler*.

The HTTP task dispatcher's architectural pattern also includes the Node Monitoring Service classes *NMARegistry* and *NMAClient* that obtain the various server-state performance and utilization metrics for each *HTTPVirtualClusterNode* via the *NodeMonitoringAgent* JSP that is installed on each instance of a Java EE Application Server, as illustrated in Figure 15.

The dependency on server state information for each implementation class is denoted by the *RequiresNMA* interface.

The F5 Networks LTM solution (F5 Networks, 2011) provides the Random, Round-Robin, static Weighted Round-Robin, Least Loaded, Fastest (re-

sponse time), Least Connections, Weighted Least Connections, and the (patent-pending) Dynamic Ratio task assignment policies.

The jQoStd HTTP task dispatcher implementation provides these (and the proposed jQoStdV1) policies as the:

- *Random,*
- *RoundRobin,*
- *StaticWeightedRoundRobin,*
- *FastestReponseTime,*
- *LeastConnections,*
- *WeightedLeastConnections,*
- *LeastLoaded,*
- *F5DynamicRatio,*
- *and jQoStdV1*

classes in the *org.symthoughts.jqostd.policies.http* package.

5.4 Design and implementation of the jQoStd Node Monitoring Agent

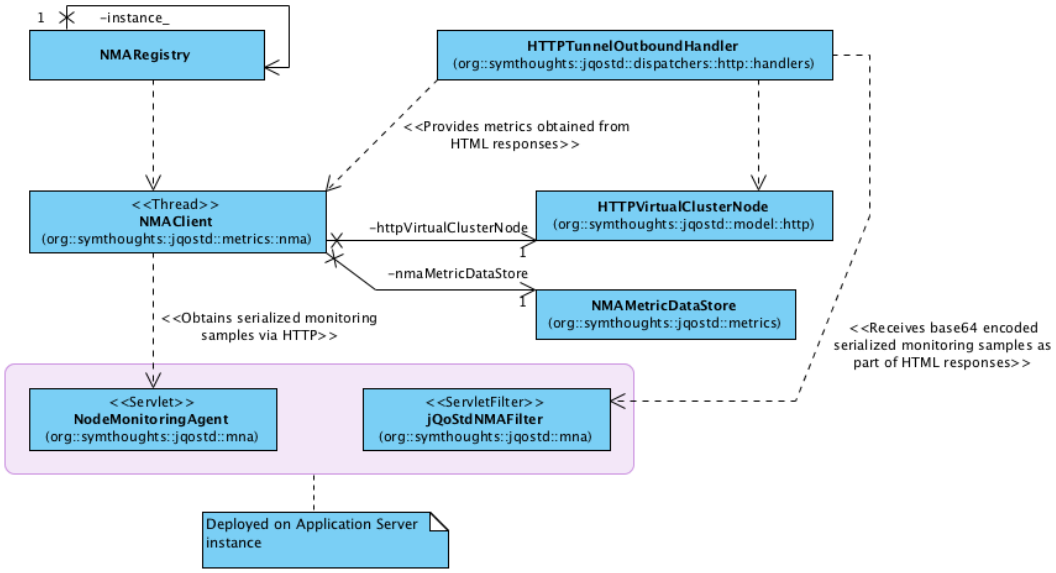


Figure 20 - Design of the jQoStd Node Monitoring Agent.

Node Monitoring is performed with a pre-packaged WAR file that is deployed into the Web Container of each Java EE Application Server instance that is configured as an *HTTPVirtualClusterNode*.

When jQoStd is started, the *NMARegistry* (singleton) will be used in order to register the *NMAClient*(s) that will retrieve serialized monitoring samples (metrics) from the *NodeMonitoringAgent* JSP for each *HTTPVirtualClusterNode*.

Server state-aware task assignment policies can then access the server state information for a node via the *NMARegistry* singleton.

Each respective *NMAClient* instance maintains the latest NMA monitoring sample in a thread-safe *Hashtable*. At each 1-second interval, this *Hashtable* is sent to the *NMAMetricDataStore*, where the data is then

stored in a time series RRD database (RRD4j, 2012) for the generation of the metric time series graphs in the Administration Console.

The *NodeMonitoringAgent* obtains various OS, Application Server, and JVM metrics through the use of JNI libraries and JMX interfaces.

The JNI libraries are provided by the Open Source Sigar API (Hyperic, 2012), and the JMX API (Oracle, 2012) is provided by Java EE compliant Application Servers.

The *jQoStdNMAFilter* is configured as an annotated Servlet Filter that is deployed as a JAR in the WEB-INF\lib folder of Java EE Web applications.

This class retrieves the same metrics as the *NodeMonitoringAgent* and inserts the metric samples into the HTML responses for some of the HTML requests that the Web Application processes, as an HTML header. The filter has been configured to insert a sample into the HTML response stream once every 100 milliseconds.

The *HTTPTunnelOutboundHandler* then intercepts and filters out these samples from the HTML responses it receives and updates the metrics for the node from where the sample was received.

This increases the effective node monitoring sample feedback rate to 10Hz, providing a substantial increase in accuracy as per the discussion in 2.5.3, while drastically reducing the inherent communication costs.

Figure 21 - Example of a HTML response with the inserted jQoStdNMA metric sample as a Base64-encoded HTTP response header.

5.5 Design and implementation of the jQoStdV1 policy

The *jQoStdV1* policy (as per Equation 2) was designed to provide optimal assignment of client connections to nodes, while preventing over-utilization states from resulting in connection and request processing failures.

The algorithm considers metrics that increase latency, and also metrics that result in failure conditions where the respective resources exceed their capacities.

The considered metrics that increase latency include:

- CPU and I/O Wait utilization from an Operating System and hardware performance perspective,
- HTTP connection queue worker threads utilization,

- and JDBC database connections.

The considered metrics that result in failures when demand exceeds maximum capacity include:

- HTTP connection queue maximum size,
- and JVM Heap Space.

The destination server with the least amount of calculated load for latency metrics is selected by the policy, where server selections are prevented whenever the thresholds for the metrics with limited capacities are exceeded.

5.6 Design and implementation of the Administration Console

The administration console was developed with JSPs, where each page is composed of reusable JSP fragment pages (includes) that provide the necessary contextual rendering for each required component.

Each page includes a header, footer, main menu, contextual menus, and main content section.

The administration console is provided from an embedded Tomcat Servlet container that is started within the jQoStd process.

Authentication and authorisation was implemented with a Servlet filter directive that establishes whether a HTTP client (jQoStd administrator) has created an authorised HTTP session. If not, an automatic redirection is

done to the Login page before a page request is processed, preventing unauthorised access to all pages.

In an attempt to reduce security-related issues, the jQoStd administration console provides user documentation with a recommended deployment configuration that includes a traffic (port) forwarding firewall that will filter and prevent access to the console (on the default port 8111) from untrusted zones.

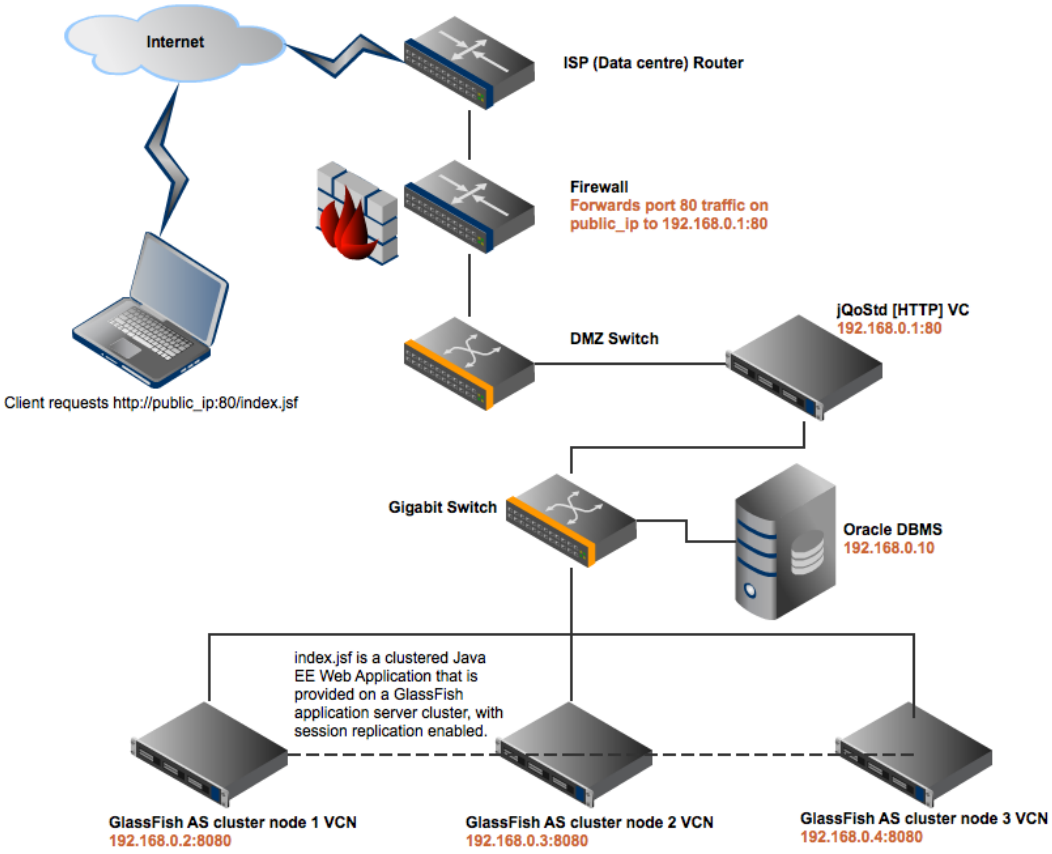


Figure 22 - Recommended production deployment of a jQoStd-based solution.

Walkthrough of the major use cases in the administration console:

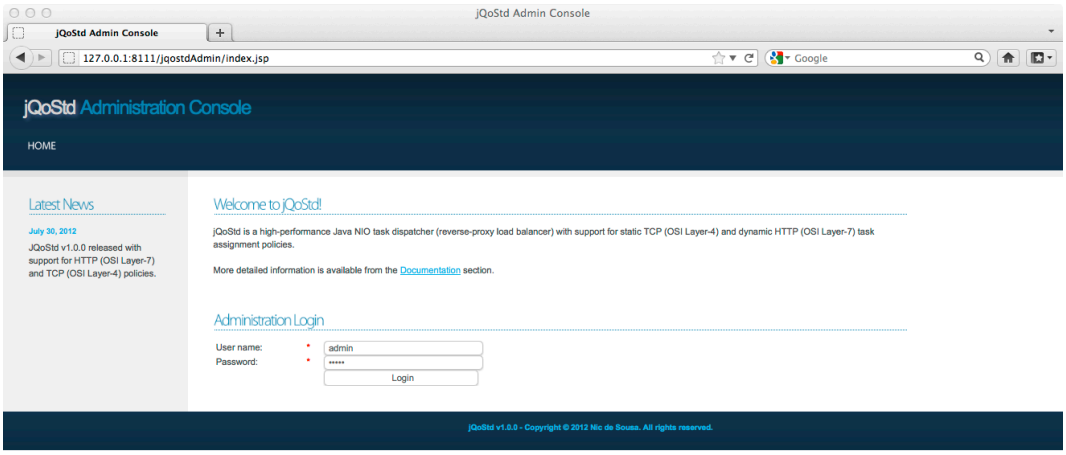


Figure 23 – Home: Authenticating a client (administrator) session with the jQoStd admin console.

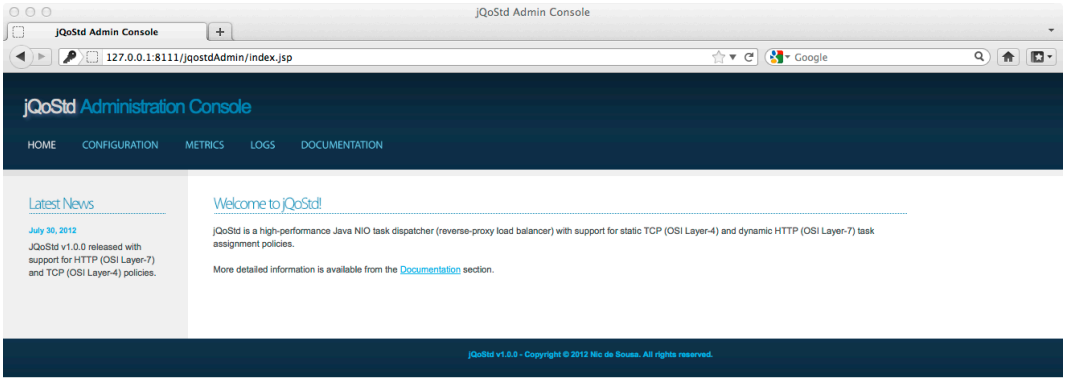


Figure 24 – Home: An authorised client session with access to the configuration, metrics, logs and documentation areas.

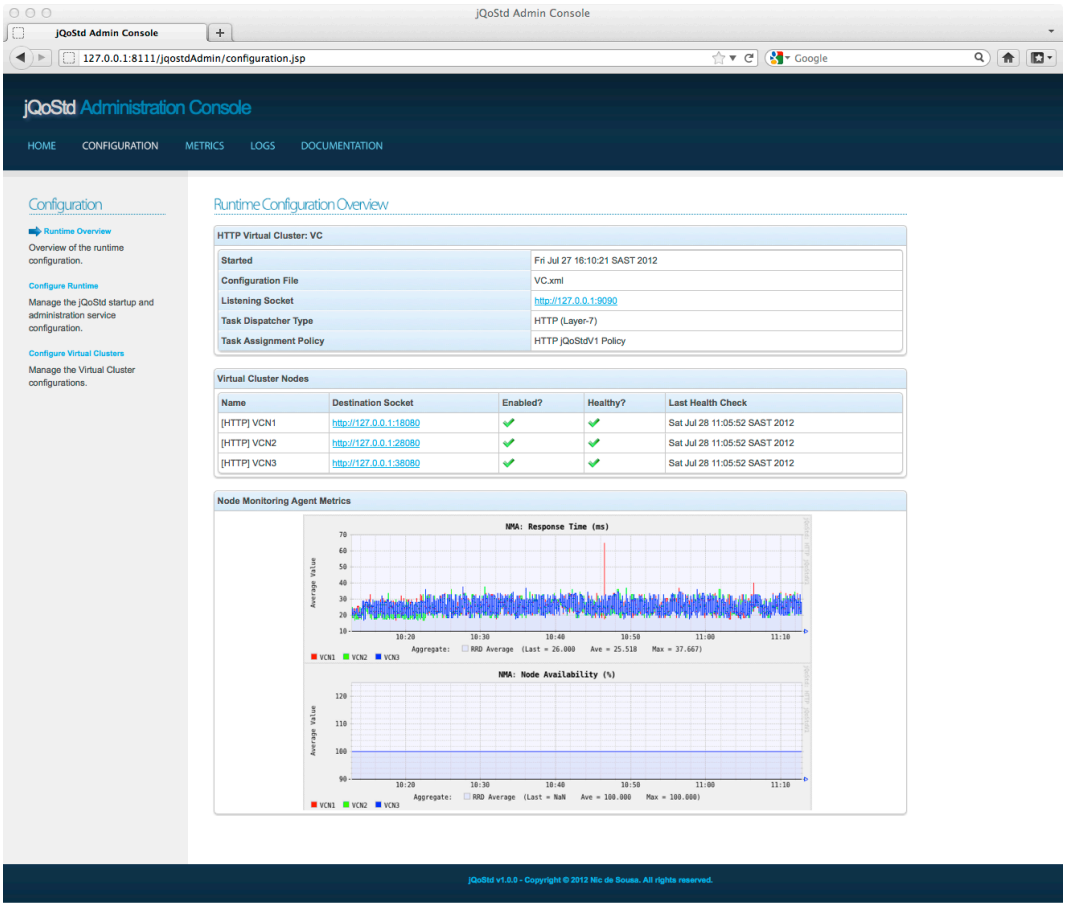


Figure 25 – Configuration: Configuration Runtime Overview of the 'VC.xml' HTTP task dispatcher configuration.

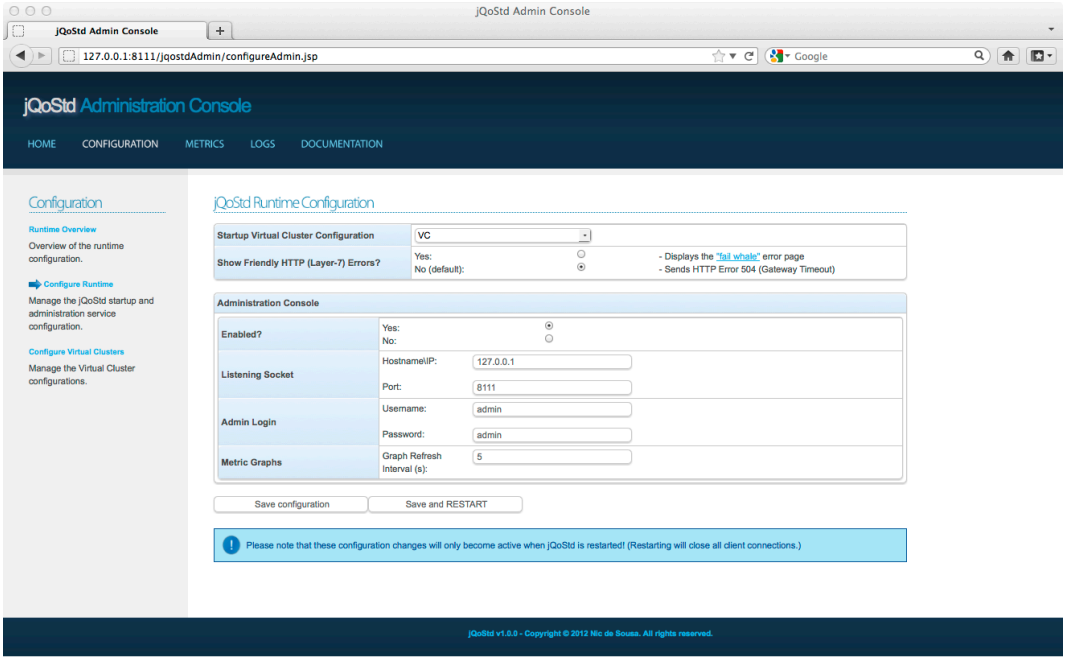


Figure 26 - Configuration: Configure Runtime for the currently active virtual cluster configuration, and general administration console settings.

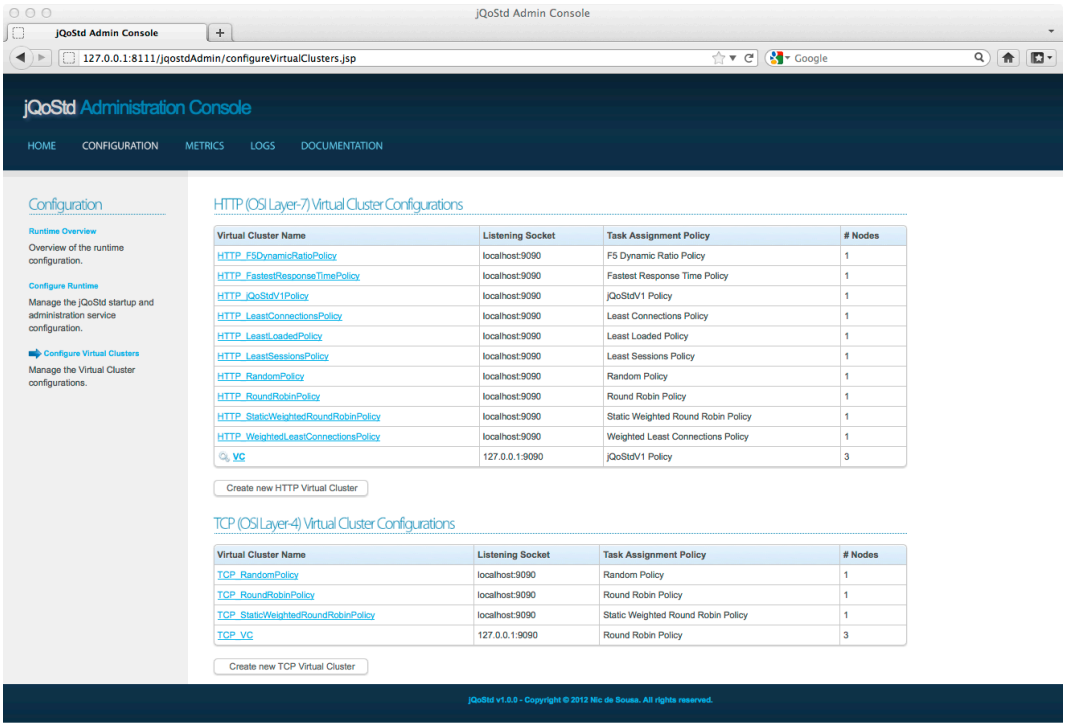


Figure 27 - Configuration: Configure Virtual Clusters, listing the available configurations, with hyperlinks that enable the maintenance of configurations.

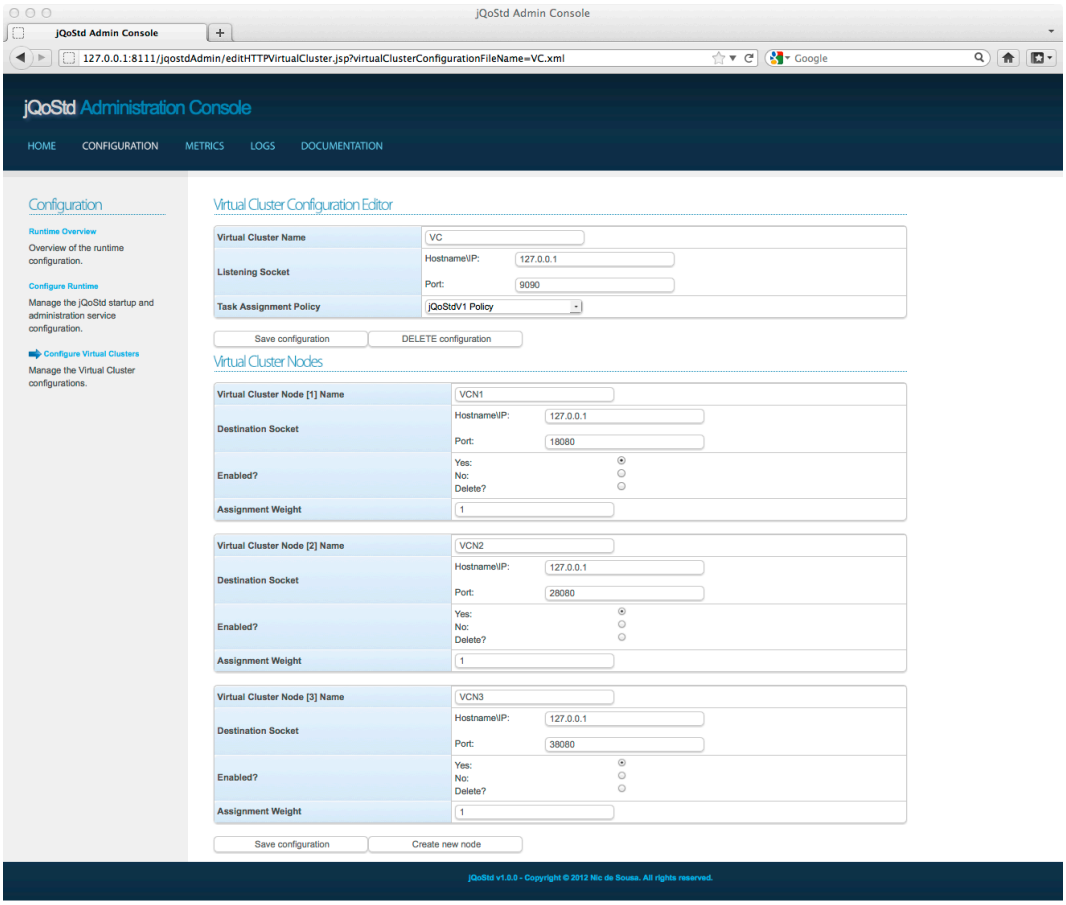


Figure 28 - Configuration: Editing the configuration of a virtual cluster. The creation of a new configuration reuses this screen with pre-populated default values.

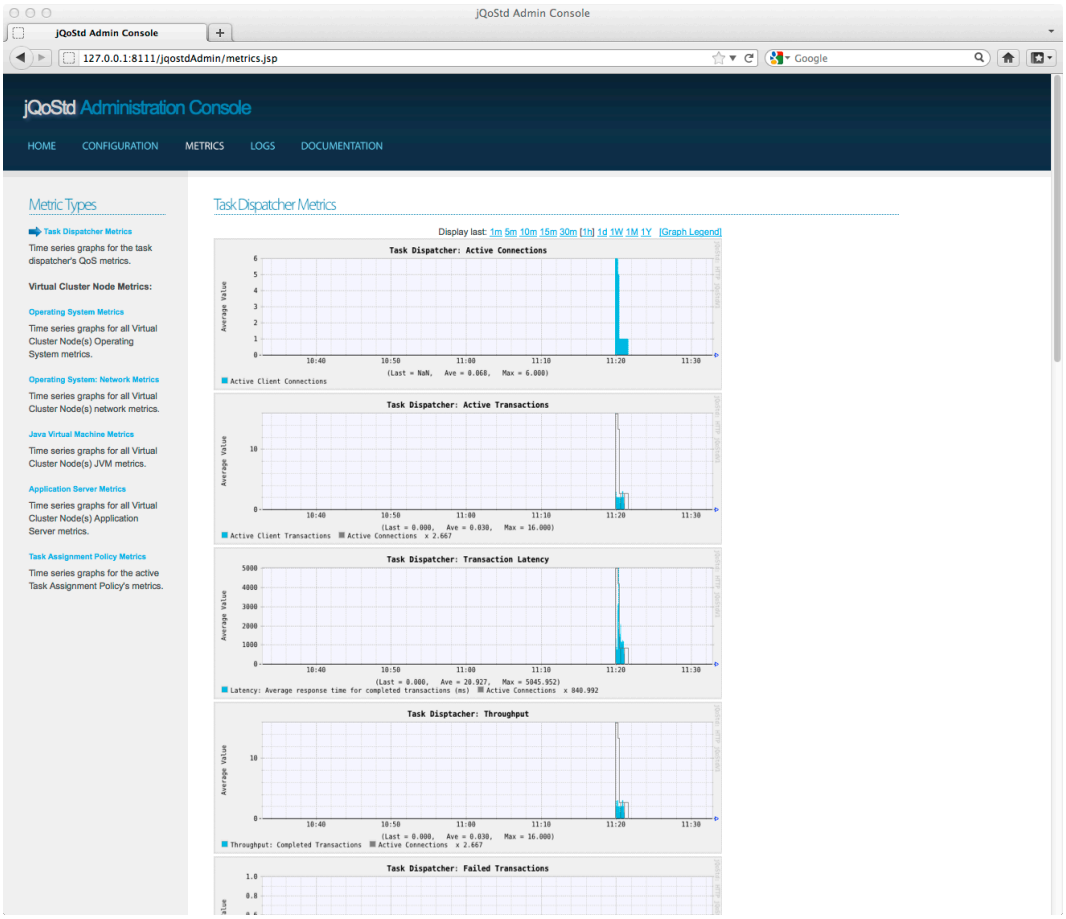


Figure 29 - Metrics: Task Dispatcher Metrics illustrating the task dispatcher time series graphs that were captured over the last 60 minutes.

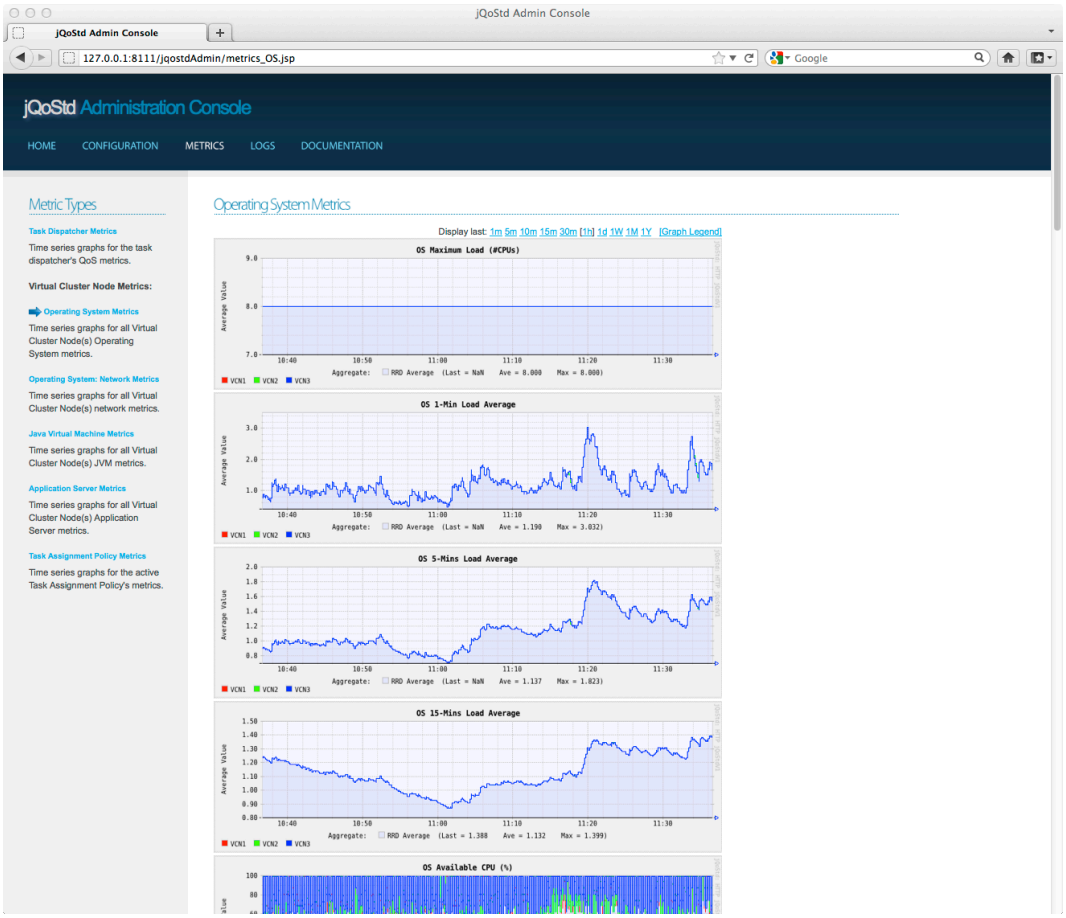


Figure 30 - Metrics: Operating System Metrics for all active Virtual Cluster Nodes.

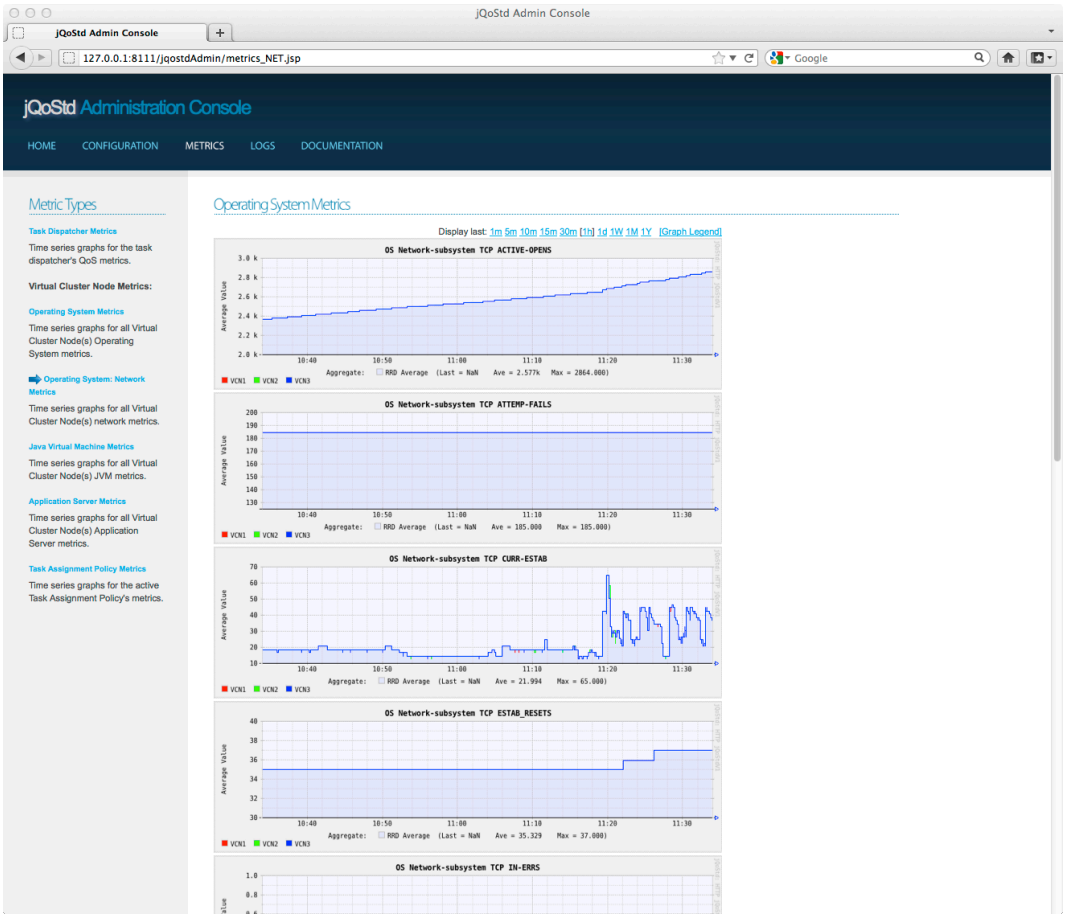


Figure 31 - Metrics: Operating System Network Metrics for all active Virtual Cluster Nodes.

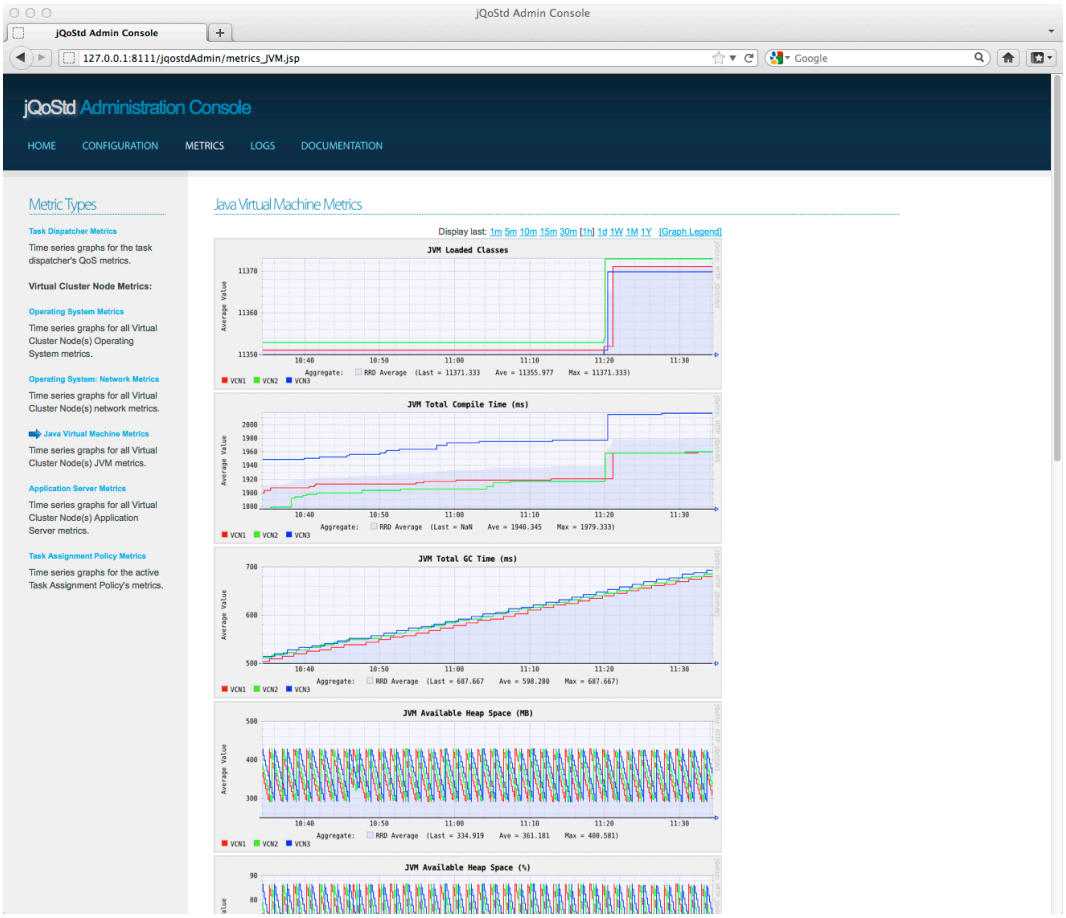


Figure 32 - Metrics: Java Virtual Machine metrics for all active Virtual Cluster Nodes.

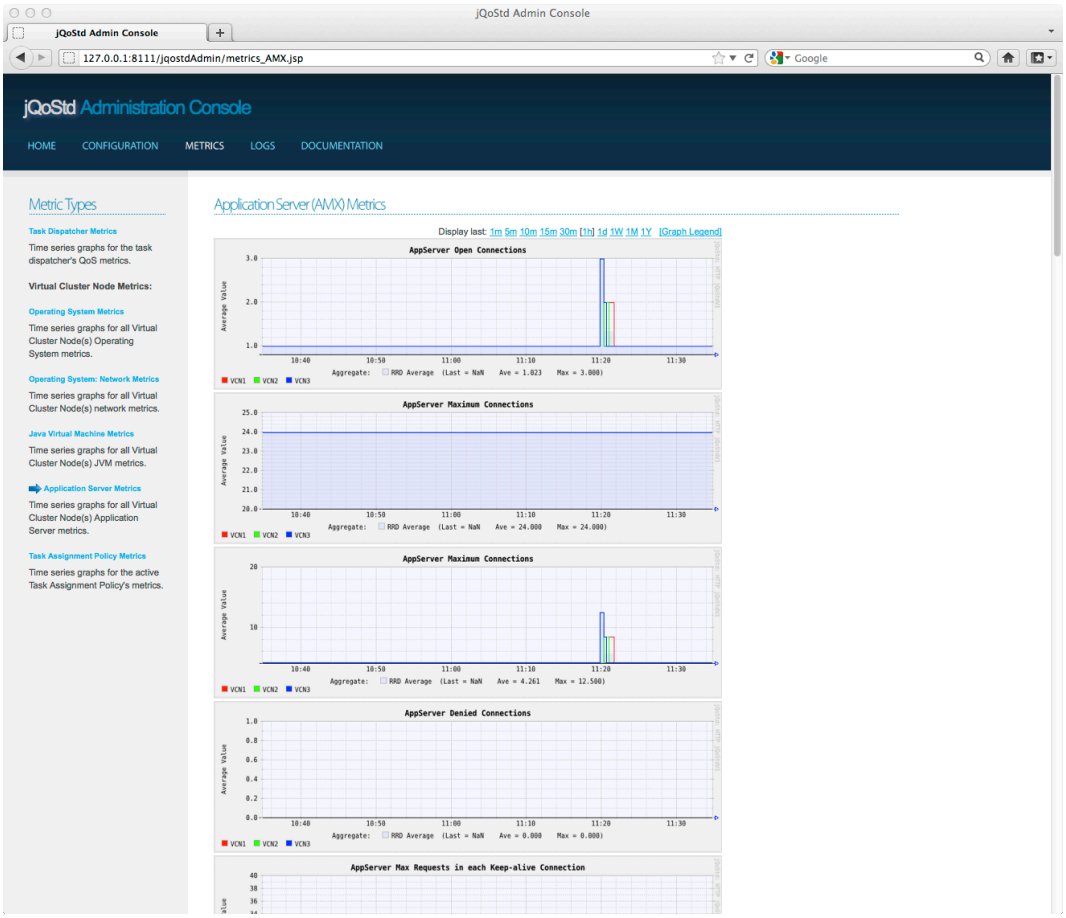


Figure 33 - Metrics: Application Server Metrics for all active Virtual Cluster Nodes.

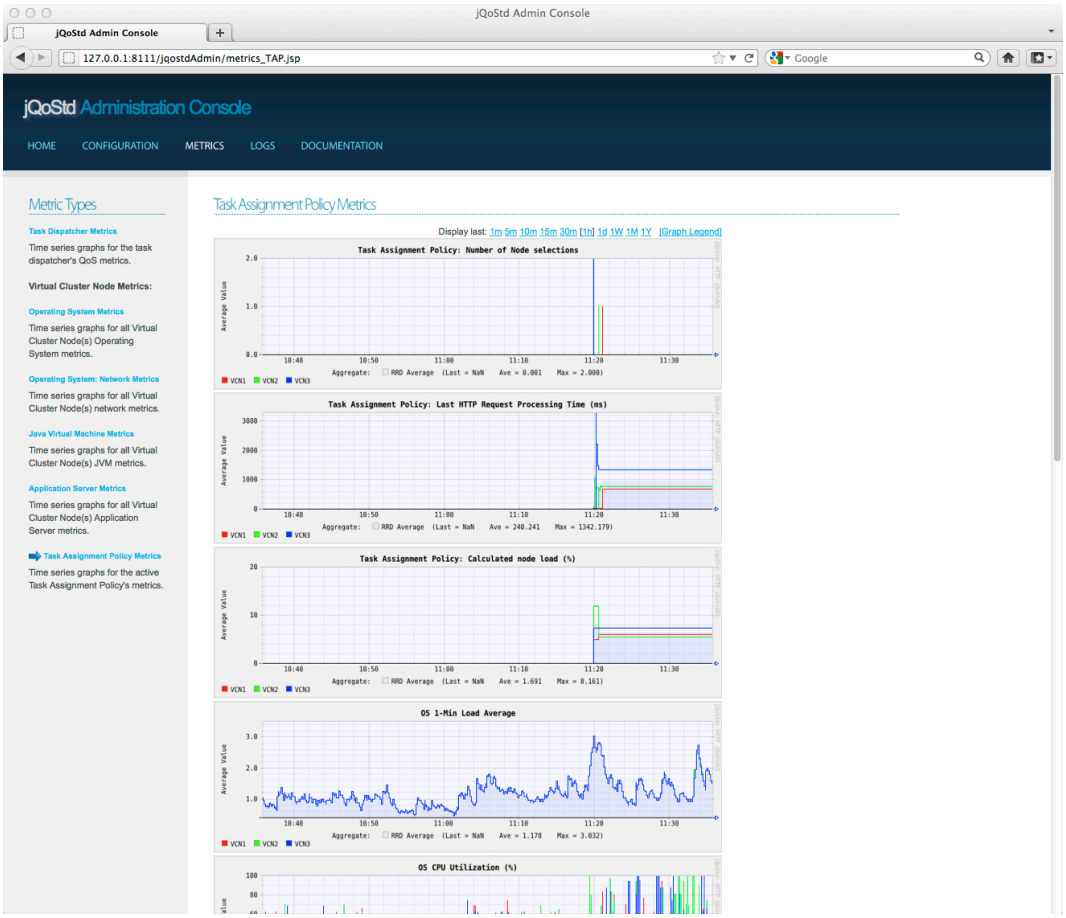


Figure 34 - Metrics: Task Assignment Policy Metrics for the active task assignment policy.

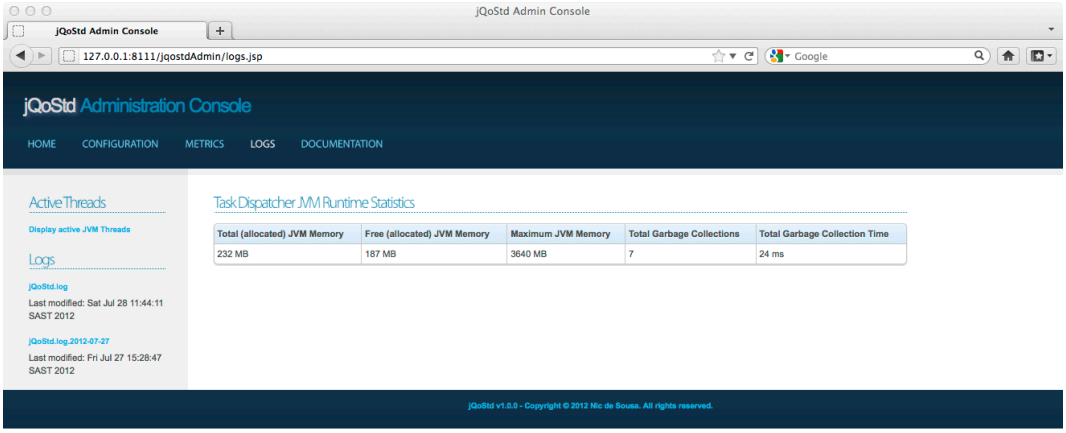


Figure 35 – Logs: Listing of the available logs, and the task dispatcher's JVM memory statistics.

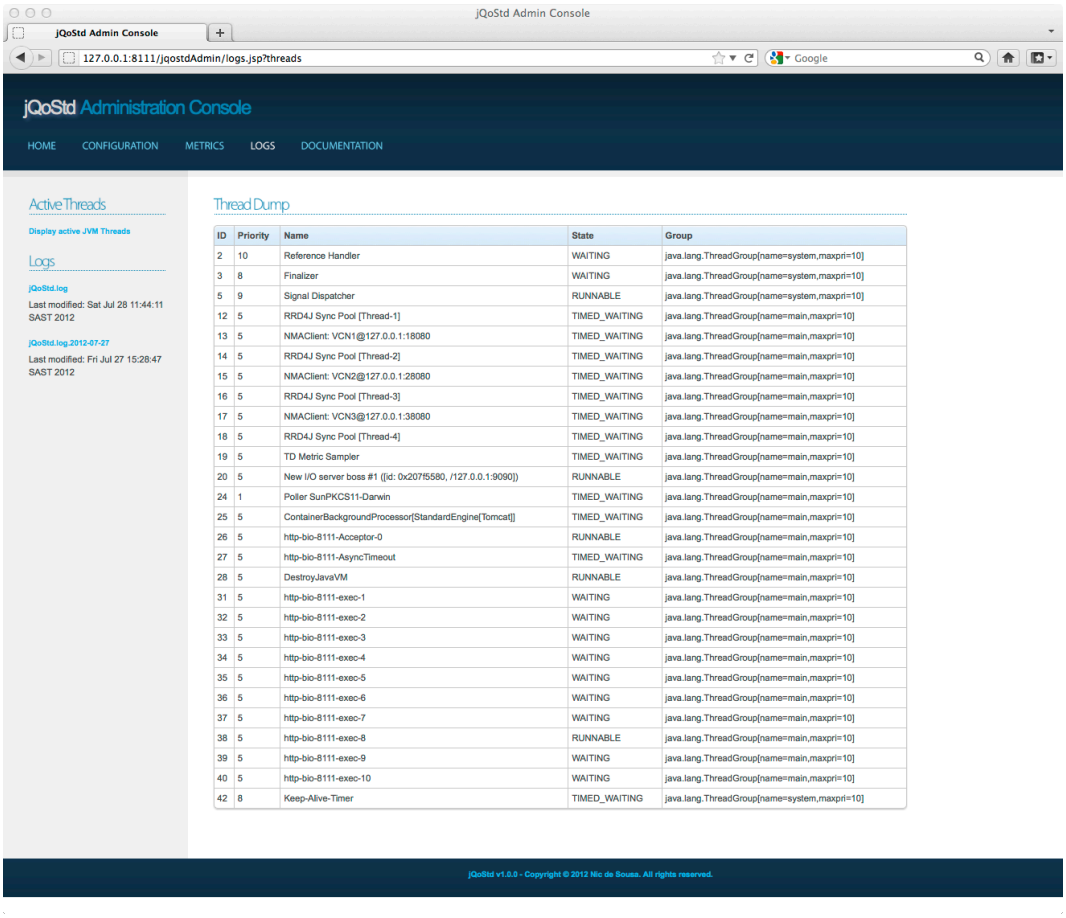


Figure 36 - Logs: Listing of the JVM threads in the task dispatcher.

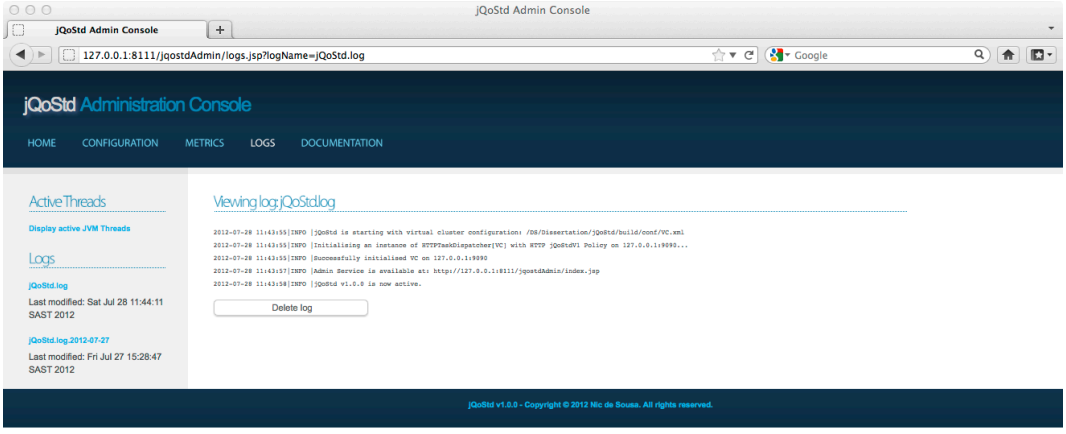


Figure 37 - Logs: Viewing of the jQoStd.log file.

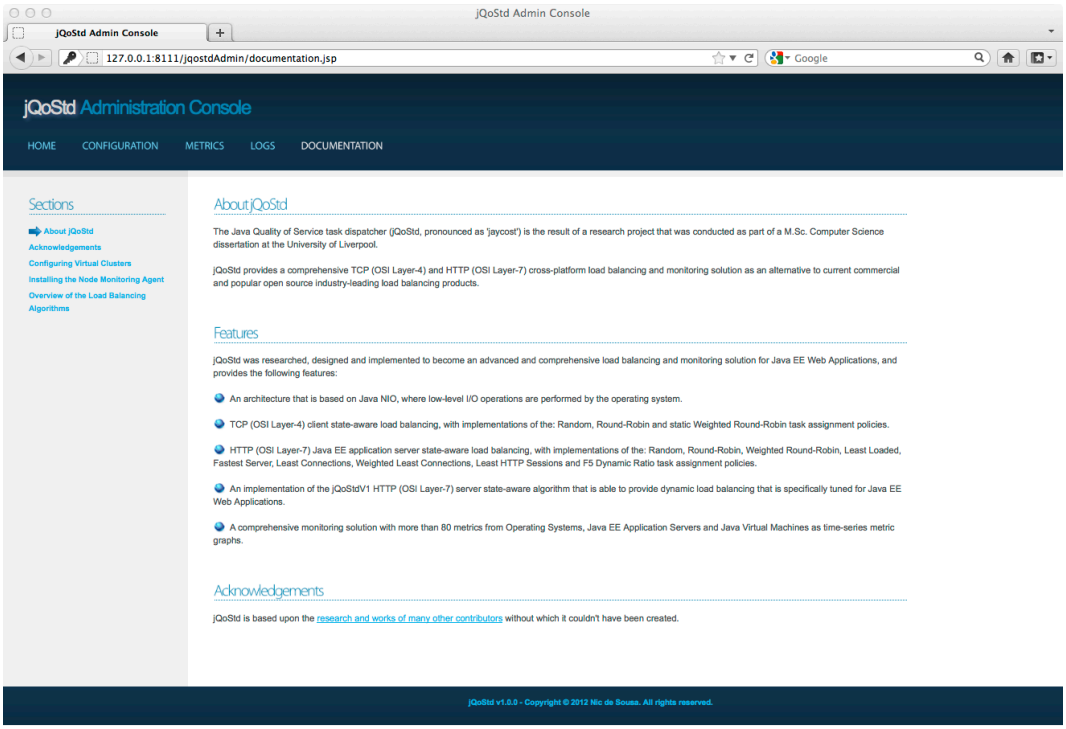


Figure 38 - Documentation: About jQoStd and its features.

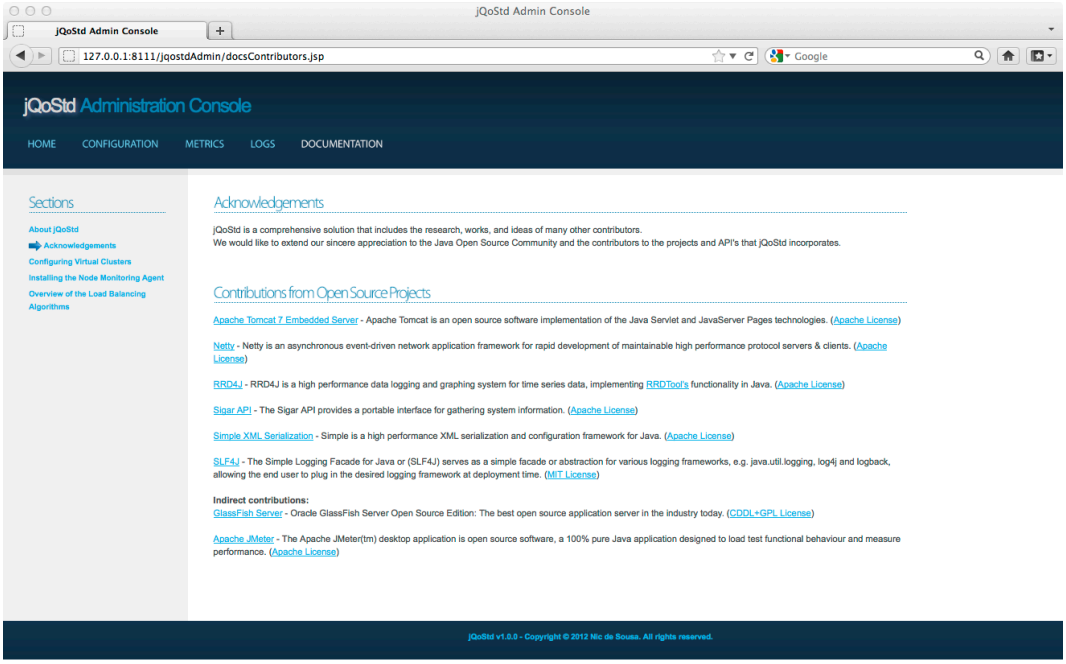


Figure 39 - Documentation: Acknowledgement of 3rd-party libraries that have been included in jQoStd.

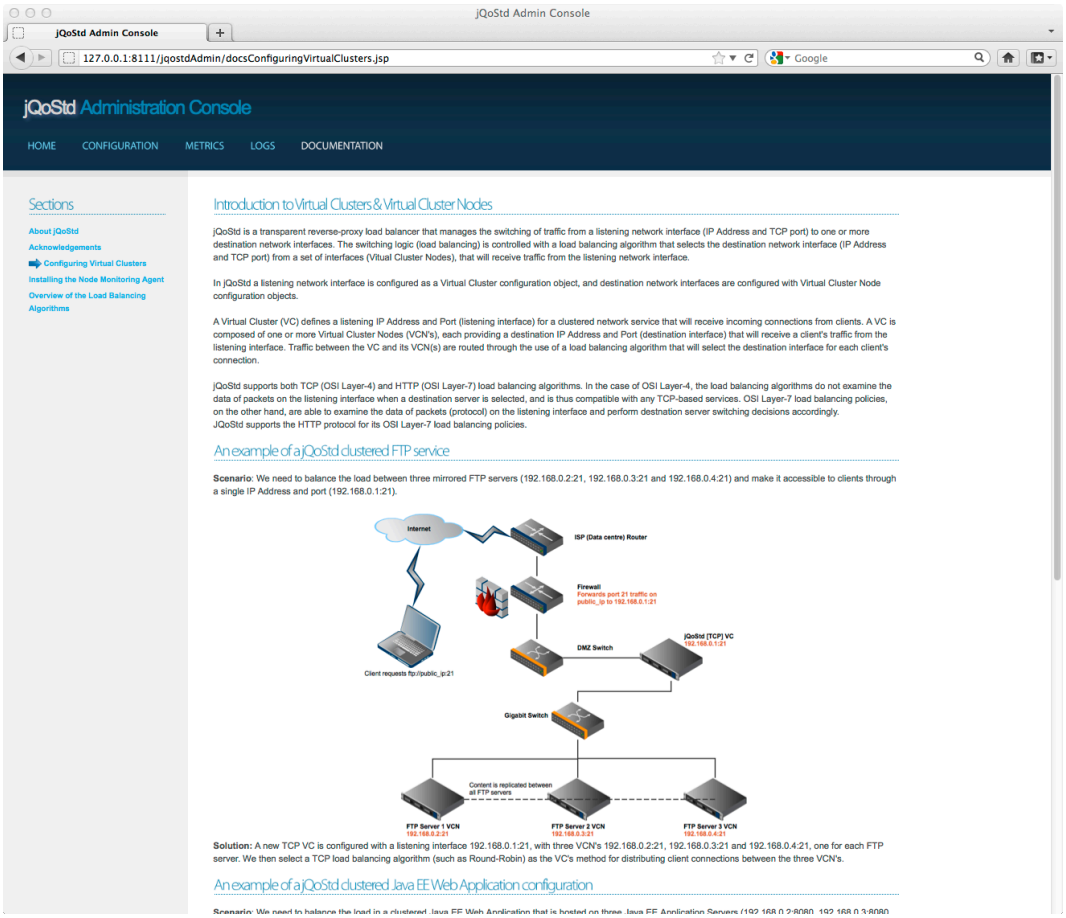


Figure 40 - Documentation: Discussion of Virtual Clusters, Virtual Cluster Nodes, and example configurations.

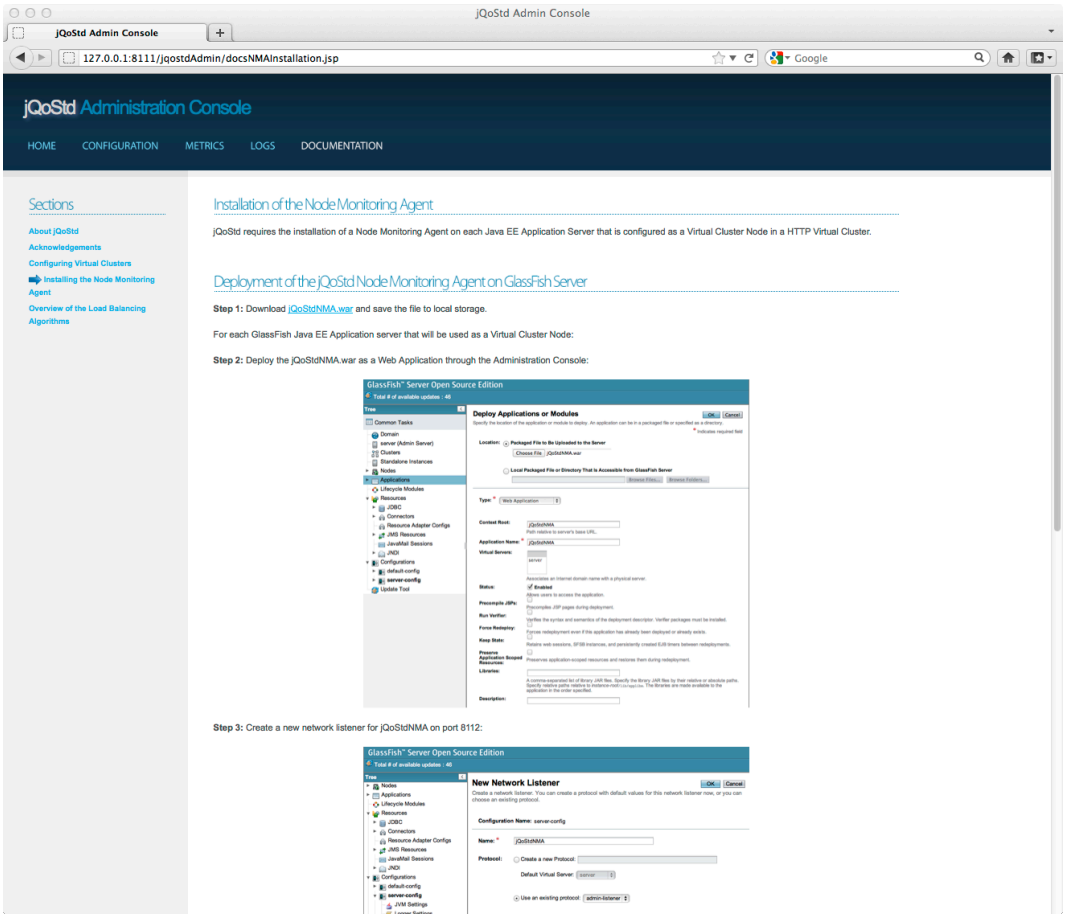


Figure 41 - Documentation: Step-by-step instructions for the installation of the jQoStd Node Monitoring Agent on GlassFish Server.

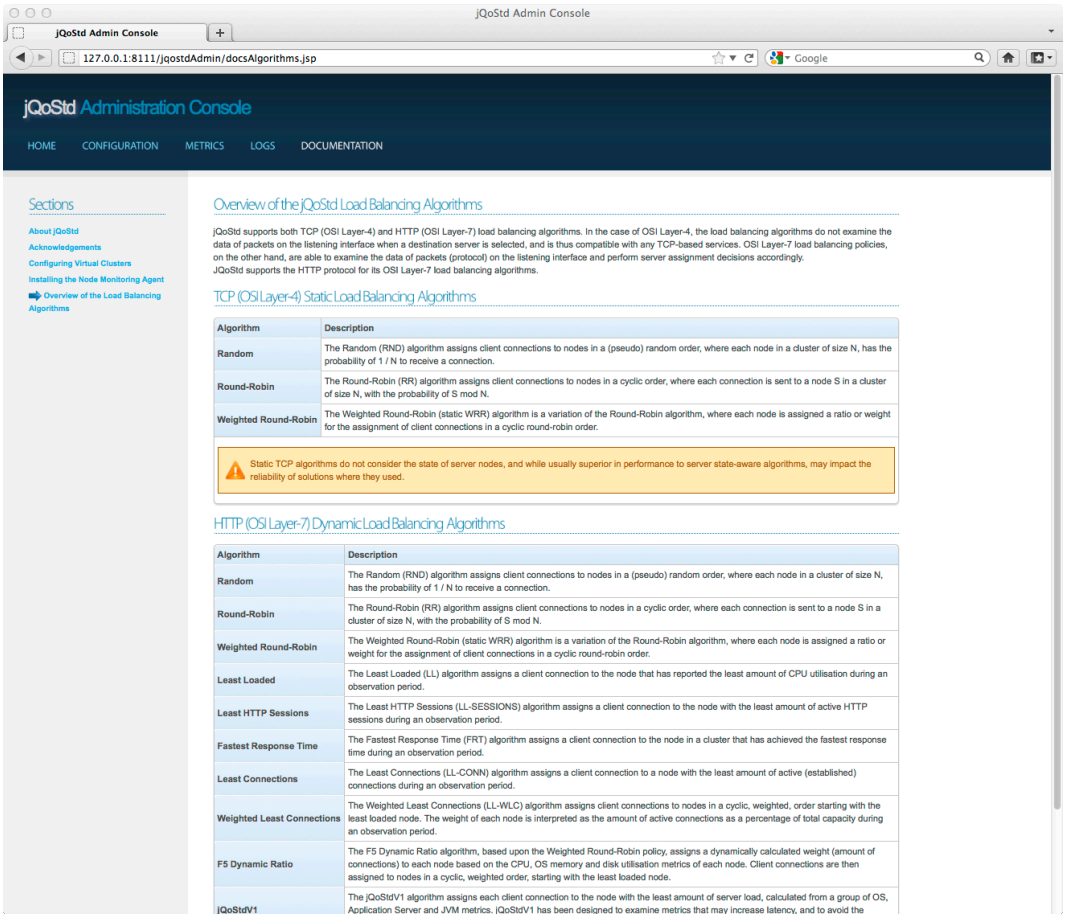


Figure 42 - Documentation: Discussion of the task assignment policies that jQoStd provides.

Chapter 6. RESULTS AND EVALUATION

6.1 Introduction

This chapter presents the results from the experiments that were conducted as part of the evaluation in this study.

The baseline configuration of the application servers, and the number of clients concurrently calling DayTrader scenarios (i.e. workload thread scheduling), were critical considerations in the study.

An iterative trial and error approach was used to establish the baseline environment configuration for use with all experiments, where the capacity of resources were gradually reduced on all servers until a 100% fault-free result was obtained with the HTTP Static Weighted Round-Robin policy with a weighted connection distribution ratio of 2:2:1.

The baseline configuration therefore represents an environment where a weighted static round-robin distribution ratio of 2:2:1 would provide consistent (fault-free) results in all configurations.

The HTTP Static Weighted Round-Robin policy (with a 2:2:1 distribution ratio) is thus used as the control for all experiments.

6.2 Overview of the experimental environment and configurations used for experiments

Figure 44 presents an overview of the environment that was used for all experiments.

A quad-core 2.2 GHz Intel Core i7 MacBook Pro, with SSD drives and 16GB of RAM was used as the test-bed computer.

Figure 43 - Configuration of the test-bed computer.

The test-bed computer hosts all of the required software components used in the experiments, including a Java Derby DBMS, the jQoStd task dispatcher, and 3 instances of the GlassFish application server, hereafter referred to as VCN1, 2 and 3.

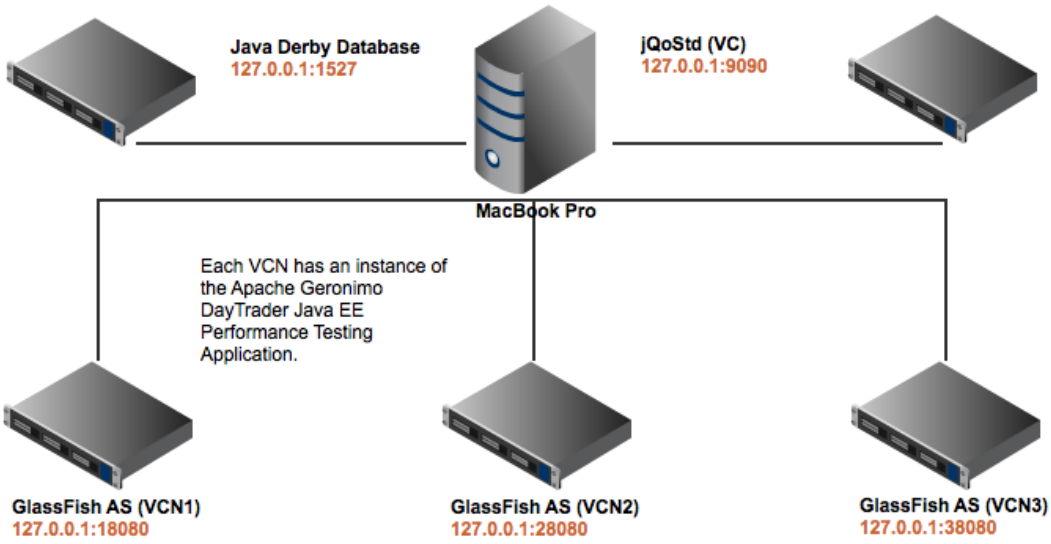


Figure 44 – Deployment of the experimental environment.

The baseline configurations that were used for the application servers in each of the experiment scenarios are defined in Table 9.

Figure 45 presents the workload thread scheduling configuration. The workload will start 3 concurrent sampler threads every 10 seconds for 1.5 minutes, sustain 30 threads for 2 minutes, and then stop 3 threads every 10 seconds thereafter.

No user think-time variability was incorporated, where sampler threads are only idle when they wait for responses. (I.e. user think time = latency)

This workload configuration results in a balanced distribution of the attack, sustain, and decay load curve, as illustrated by the active connections and throughput graphs in Figure 46 and Figure 47.

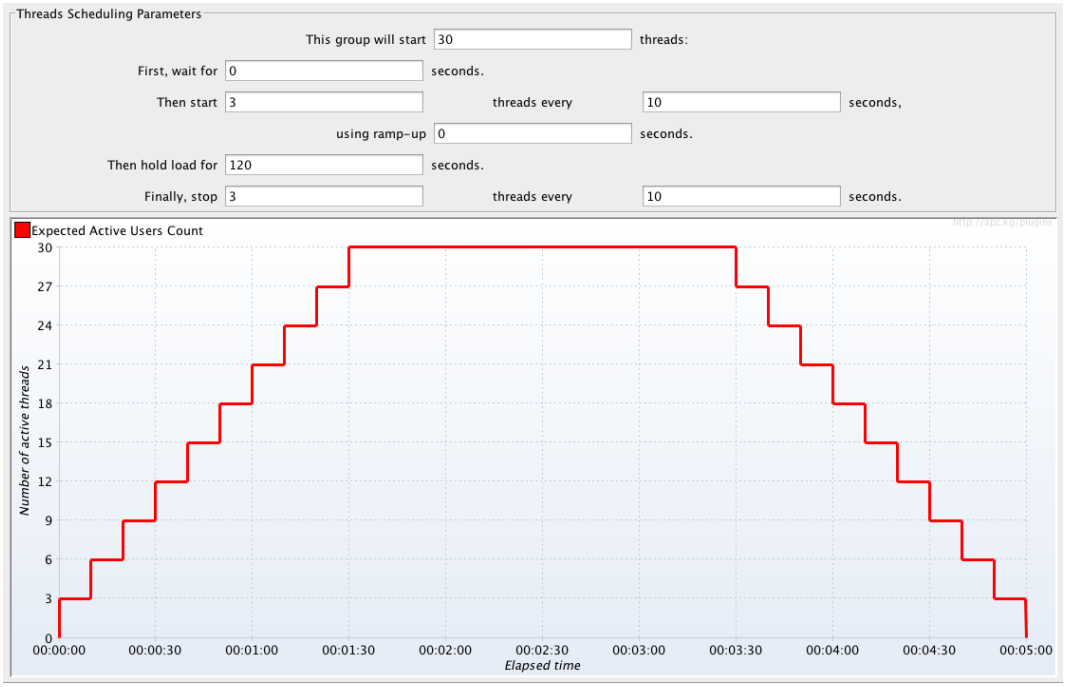


Figure 45 - The workload thread scheduling parameters.

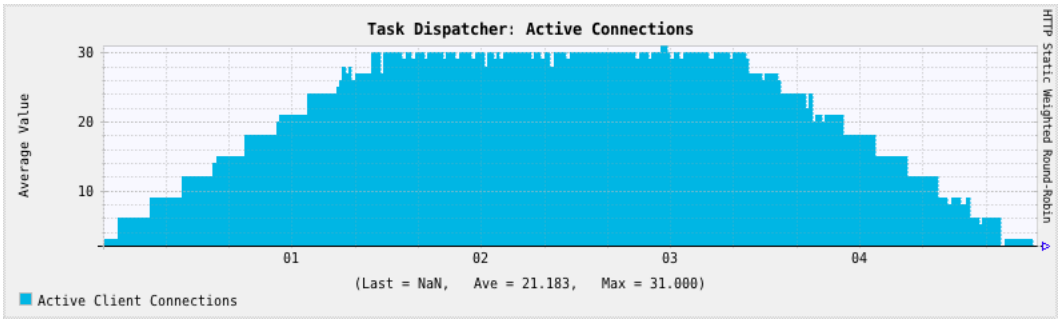


Figure 46 - Active Connections graph for the HTTP Static WRR policy.

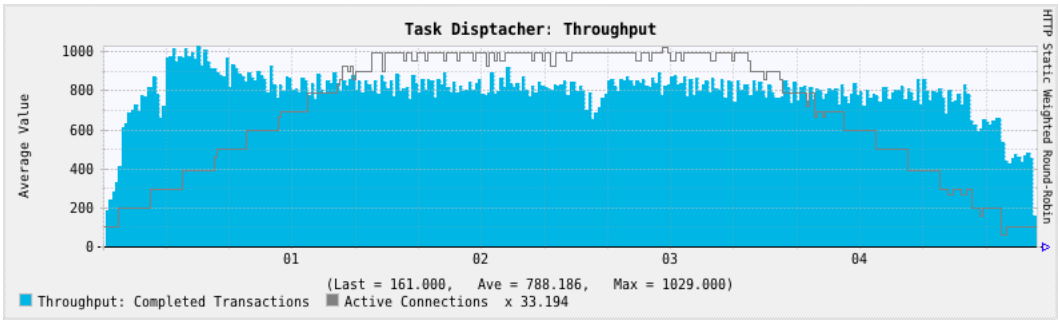


Figure 47 - Throughput graph for the HTTP Static WRR policy.

Baseline experiment VCN configurations				
AppServer Name	Heap Space	HTTP connection queue threads	HTTP connection queue size	JDBC connector threads
VCN1	512MB	12	12	10
VCN2	512MB	12	12	10
VCN3	512MB	12	12	10

Heap Space experiment VCN configurations				
AppServer Name	Heap Space	HTTP connection queue threads	HTTP connection queue size	JDBC connector threads
VCN1	512MB	12	12	10
VCN2	512MB	12	12	10
VCN3	*256MB*	12	12	10

HTTP queue worker threads experiment VCN configurations				
AppServer Name	Heap Space	HTTP connection queue threads	HTTP connection queue size	JDBC connector threads
VCN1	512MB	12	12	10
VCN2	512MB	12	12	10
VCN3	512MB	*6*	12	10

JDBC connection threads experiment VCN configurations				
AppServer Name	Heap Space	HTTP connection queue threads	HTTP connection queue size	JDBC connector threads
VCN1	512MB	12	12	10
VCN2	512MB	12	12	10
VCN3	512MB	12	12	*5*

Combined Scenarios experiment VCN configurations				
AppServer Name	Heap Space	HTTP connection queue threads	HTTP connection queue size	JDBC connector threads
VCN1	512MB	12	12	10
VCN2	512MB	12	12	10
VCN3	*256MB*	*6*	12	*5*

Table 9 - Application Server configurations for each experiment.

6.3 Results and evaluation of experiments

6.3.1 The baseline reference policy and failed-requests tolerance

The HTTP Static Weighted Round-Robin policy with a weight ratio of 2:2:1 was used to establish the baseline reference (control) for all experiments.

The control policy, at times, produced a maximum of two failed requests in preliminary tests. All policies that therefore produced a maximum of 2

failed requests were adjusted to 0 failures, as they often completed without failures in the preliminary tests.

6.3.2 TCP and HTTP Policies

In the experiment results a reference is made to TCP and HTTP policies:

TCP policies represent content and server state-blind policies, and HTTP policies represent (HTTP) content and server state-aware policies that have access to the server state metrics and server health information that is maintained by the Node Monitoring Service for each node.

TCP policies do not have access to the Node Monitoring Service, and no server state, health information, or NMA time series graphs (other than the task dispatcher graphs) are available for their analysis.

6.3.3 Tabulated experiment results

Baseline Experiment Results												
Task Assignment Policy	Weight Ratio	Samples	Active Conns	Throughput	Failed	Failed %	Latency	WT1	WT2	PT	Slow-down	Node Sels
TCP Random	1:1:1	250350	21.19	848.64	0.00	0.00	24.06	0.00	0.00	24.06	1.00	27.41
HTTP Random	1:1:1	233862	21.35	790.07	0.00	0.00	29.49	0.00	0.01	29.48	1.00	25.58
TCP Round-Robin	1:1:1	251632	21.16	852.99	0.00	0.00	24.85	0.00	0.03	24.83	1.02	27.54
HTTP Round-Robin	1:1:1	232013	21.09	783.83	0.00	0.00	26.14	0.00	0.05	26.09	1.04	25.42
TCP Static Weighted Round-Robin	2:2:1	251829	21.19	853.66	0.00	0.00	24.61	0.00	0.07	24.55	1.06	27.56
HTTP Static Weighted Round-Robin	2:2:1	233303	21.12	788.19	0.00	0.00	26.94	0.00	0.05	26.88	1.05	25.52
HTTP F5 Dynamic Ratio	1:1:1	231642	21.39	782.57	0.00	0.00	27.25	0.00	0.01	27.24	1.00	25.39
HTTP jQoStdV1Policy	1:1:1	229494	21.37	783.26	0.00	0.00	26.82	0.00	0.05	26.77	1.05	25.40
HTTP Fastest Response Time	1:1:1	239469	20.46	772.48	1693.00	0.71	34.69	0.00	0.00	34.69	1.00	29.83
HTTP Least Loaded	1:1:1	241398	20.81	791.47	30.00	0.01	31.21	0.00	0.00	31.21	1.00	25.68
HTTP Least Connections	1:1:1	238383	21.20	805.35	0.00	0.00	24.70	0.00	0.06	24.64	1.05	26.07
HTTP Weighted Least Connections	2:2:1	231679	21.15	782.70	0.00	0.00	28.60	0.00	0.01	28.59	1.00	25.36
Min		*229494*	*20.46*	*772.48*	*0.00*	*0.00*	*24.06*	*0.00*	*0.00*	*24.06*	*1.00*	*25.36*
Max		*251829*	*21.39*	*853.66*	*1693.00*	*0.71*	*34.69*	*0.00*	*0.07*	*34.69*	*1.06*	*29.83*
Average		*238755*	*21.12*	*802.93*	*143.58*	*0.06*	*27.45*	*0.00*	*0.03*	*27.42*	*1.02*	*26.40*
Standard Deviation		*8336*	*0.26*	*30.44*	*488.02*	*0.20*	*3.14*	*0.00*	*0.03*	*3.15*	*0.02*	*1.40*

Table 10 - The results of the Baseline experiment.

JVM Heap Space Experiment Results												
Task Assignment Policy	Weight Ratio	Samples	Active Conns	Throughput	Failed	Failed %	Latency	WT1	WT2	PT	Slow-down	Node Sels
TCP Random	1:1:1	181503	20.30	579.88	108.00	0.06	154.33	0.00	0.01	154.32	1.00	19.02
HTTP Random	1:1:1	183838	21.46	614.84	0.00	0.00	69.98	0.00	0.01	69.98	1.00	19.90
TCP Round-Robin	1:1:1	183269	21.37	617.07	62.00	0.03	70.60	0.00	0.00	70.59	1.00	20.10
HTTP Round-Robin	1:1:1	177230	21.16	590.77	23.00	0.01	69.29	0.00	0.01	69.28	1.00	19.20
TCP Static Weighted Round-Robin	2:2:1	252091	21.22	854.55	0.00	0.00	25.59	0.00	0.01	25.58	1.00	27.58
HTTP Static Weighted Round-Robin	2:2:1	232627	21.29	785.90	0.00	0.00	32.43	0.00	0.02	32.42	1.00	25.44
HTTP F5 Dynamic Ratio	1:1:1	180411	21.24	609.50	58.00	0.03	57.49	0.00	0.01	57.49	1.00	19.95
HTTP jQoStdV1Policy	1:1:1	234651	21.42	792.74	0.00	0.00	27.54	0.00	0.05	27.49	1.05	25.67
HTTP Fastest Response Time	1:1:1	239415	20.62	764.90	1457.00	0.61	226.21	0.00	0.00	226.20	1.00	28.73
HTTP Least Loaded	1:1:1	210546	21.59	708.91	13.00	0.01	64.23	0.00	0.01	64.23	1.00	23.00
HTTP Least Connections	1:1:1	211775	21.47	715.46	0.00	0.00	95.59	0.00	0.01	95.58	1.00	23.20
HTTP Weighted Least Connections	2:2:1	197858	21.29	668.44	0.00	0.00	39.18	0.00	0.05	39.13	1.04	21.63
Min		*177230*	*20.30*	*579.88*	*0.00*	*0.00*	*25.59*	*0.00*	*0.00*	*25.58*	*1.00*	*19.02*
Max		*252091*	*21.59*	*854.55*	*1457.00*	*0.61*	*226.21*	*0.00*	*0.05*	*226.20*	*1.05*	*28.73*
Average		*207101*	*21.20*	*691.91*	*143.58*	*0.06*	*77.70*	*0.00*	*0.01*	*77.69*	*1.01*	*22.78*
Standard Deviation		*26912*	*0.37*	*92.09*	*415.06*	*0.17*	*58.61*	*0.00*	*0.02*	*58.61*	*0.02*	*3.38*

Table 11 - The results of the JVM Heap Space experiment.

Application Server HTTP connection queue worker threads Experiment Results												
Task Assignment Policy	Weight Ratio	Samples	Active Conns	Throughput	Failed	Failed %	Latency	WT1	WT2	PT	Slow-down	Node Sels
TCP Random	1:1:1	250174	21.28	848.05	504.00	0.20	24.58	0.00	0.00	24.58	1.00	28.79
HTTP Random	1:1:1	230822	21.21	779.80	360.00	0.16	26.97	0.00	0.00	26.97	1.00	26.20
TCP Round-Robin	1:1:1	243357	21.13	824.94	517.00	0.21	25.68	0.00	0.00	25.68	1.00	28.01
HTTP Round-Robin	1:1:1	225674	21.13	762.41	284.00	0.13	27.25	0.00	0.05	27.20	1.04	25.39
TCP Static Weighted Round-Robin	2:2:1	252575	21.06	856.19	0.00	0.00	24.59	0.00	0.00	24.59	1.00	27.63
HTTP Static Weighted Round-Robin	2:2:1	236001	21.11	797.30	0.00	0.00	27.18	0.00	0.01	27.17	1.00	25.84
HTTP F5 Dynamic Ratio	1:1:1	228551	21.16	772.13	195.00	0.09	27.88	0.00	0.01	27.87	1.00	25.45
HTTP jQoStdV1Policy	1:1:1	231670	21.12	782.67	0.00	0.00	30.41	0.00	0.01	30.40	1.00	25.32
HTTP Fastest Response Time	1:1:1	234728	20.99	774.68	1912.00	0.81	35.00	0.00	0.00	35.00	1.00	30.62
HTTP Least Loaded	1:1:1	222632	19.95	704.53	1169.00	0.53	35.44	0.00	0.01	35.44	1.00	25.97
HTTP Least Connections	1:1:1	239588	21.50	809.42	0.00	0.00	25.75	0.00	0.06	25.69	1.05	26.18
HTTP Weighted Least Connections	2:2:1	235018	21.15	793.98	0.00	0.00	26.70	0.00	0.01	26.69	1.00	25.74
Min		*222632*	*19.95*	*704.53*	*0.00*	*0.00*	*24.58*	*0.00*	*0.00*	*24.58*	*1.00*	*25.32*
Max		*252575*	*21.50*	*856.19*	*1912.00*	*0.81*	*35.44*	*0.00*	*0.06*	*35.44*	*1.05*	*30.62*
Average		*235899*	*21.07*	*792.18*	*411.83*	*0.18*	*28.12*	*0.00*	*0.01*	*28.11*	*1.01*	*26.76*
Standard Deviation		*9202*	*0.37*	*40.61*	*584.39*	*0.25*	*3.67*	*0.00*	*0.02*	*3.67*	*0.02*	*1.66*

Table 12 - The results of the HTTP connection queue worker threads experiment.

Application Server JDBC connection pool threads Experiment Results												
Task Assignment Policy	Weight Ratio	Samples	Active Conns	Throughput	Failed	Failed %	Latency	WT1	WT2	PT	Slow-down	Node Sels
TCP Random	1:1:1	248692	20.80	787.00	93.00	0.04	53.79	0.00	0.01	53.78	1.00	25.64
HTTP Random	1:1:1	226521	21.43	742.69	120.00	0.05	61.58	0.00	0.00	61.57	1.00	24.32
TCP Round-Robin	1:1:1	248072	21.66	832.46	0.00	0.00	49.10	0.00	0.01	49.09	1.00	26.88
HTTP Round-Robin	1:1:1	224829	21.69	751.94	8.00	0.00	33.86	0.00	0.06	33.80	1.04	24.31
TCP Static Weighted Round-Robin	2:2:1	253921	21.31	860.75	0.00	0.00	24.57	0.00	0.00	24.56	1.00	27.80
HTTP Static Weighted Round-Robin	2:2:1	233498	21.14	788.85	0.00	0.00	27.65	0.00	0.01	27.64	1.00	25.50
HTTP F5 Dynamic Ratio	1:1:1	225881	21.26	738.17	3.00	0.00	69.31	0.00	0.00	69.30	1.00	23.85
HTTP jQoStdV1Policy	1:1:1	227004	21.08	746.72	0.00	0.00	25.88	0.00	0.00	25.88	1.00	24.09
HTTP Fastest Response Time	1:1:1	235983	19.46	723.87	1427.00	0.60	42.31	0.00	0.01	42.30	1.00	27.21
HTTP Least Loaded	1:1:1	232366	21.50	769.42	476.00	0.20	39.76	0.00	0.00	39.76	1.00	26.23
HTTP Least Connections	1:1:1	236874	21.98	800.25	0.00	0.00	28.22	0.00	0.01	28.22	1.00	25.88
HTTP Weighted Least Connections	2:2:1	233487	21.14	788.81	0.00	0.00	27.17	0.00	0.01	27.17	1.00	25.48
Min		*224829*	*19.46*	*723.87*	*0.00*	*0.00*	*24.57*	*0.00*	*0.00*	*24.56*	*1.00*	*23.85*
Max		*253921*	*21.98*	*860.75*	*1427.00*	*0.60*	*69.31*	*0.00*	*0.06*	*69.30*	*1.04*	*27.80*
Average		*235594*	*21.20*	*777.58*	*177.25*	*0.08*	*40.27*	*0.00*	*0.01*	*40.25*	*1.00*	*25.60*
Standard Deviation		*9772*	*0.63*	*40.58*	*416.59*	*0.18*	*15.17*	*0.00*	*0.01*	*15.18*	*0.01*	*1.29*

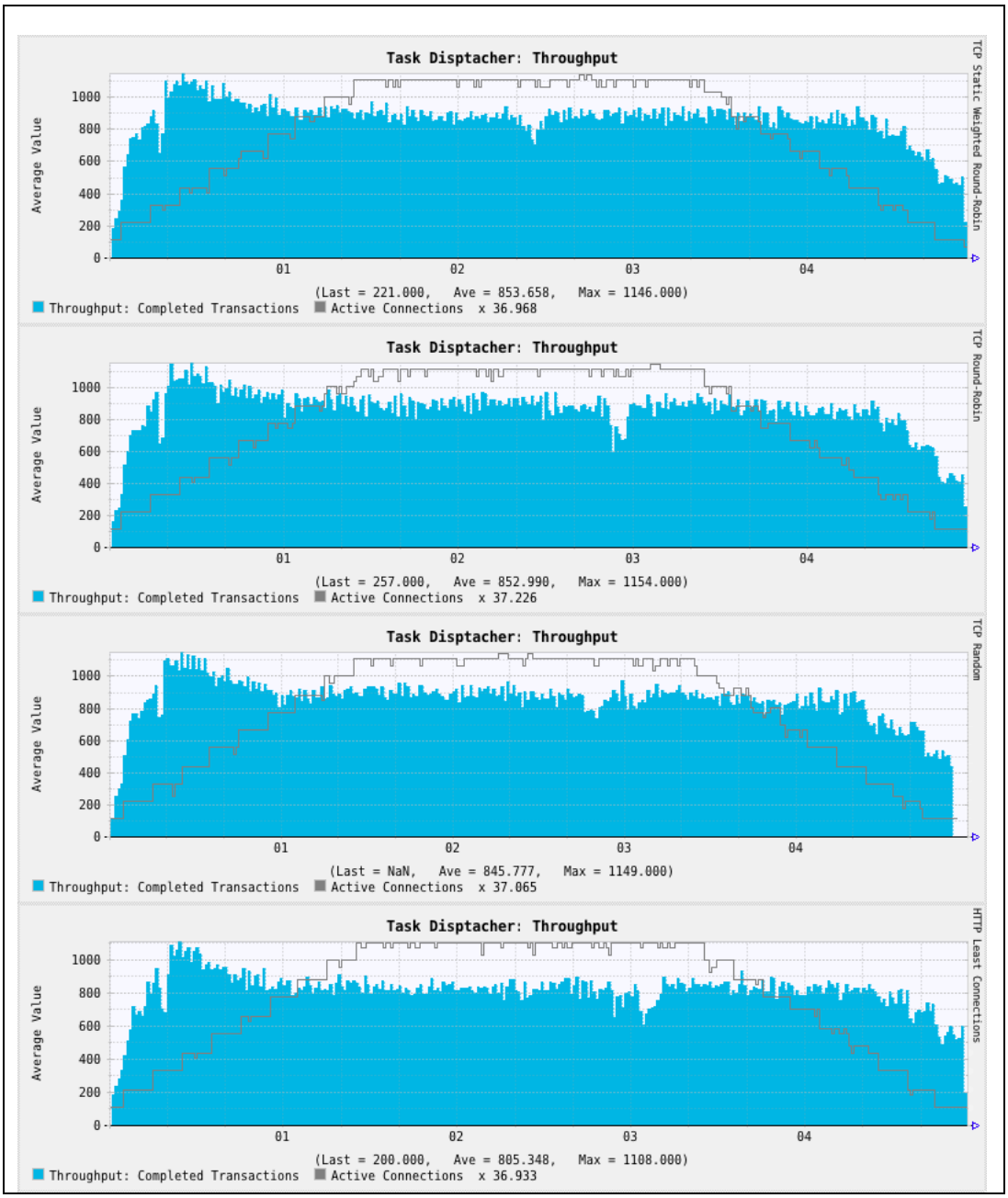
Table 13 - The results of the JDBC database connection threads experiment.

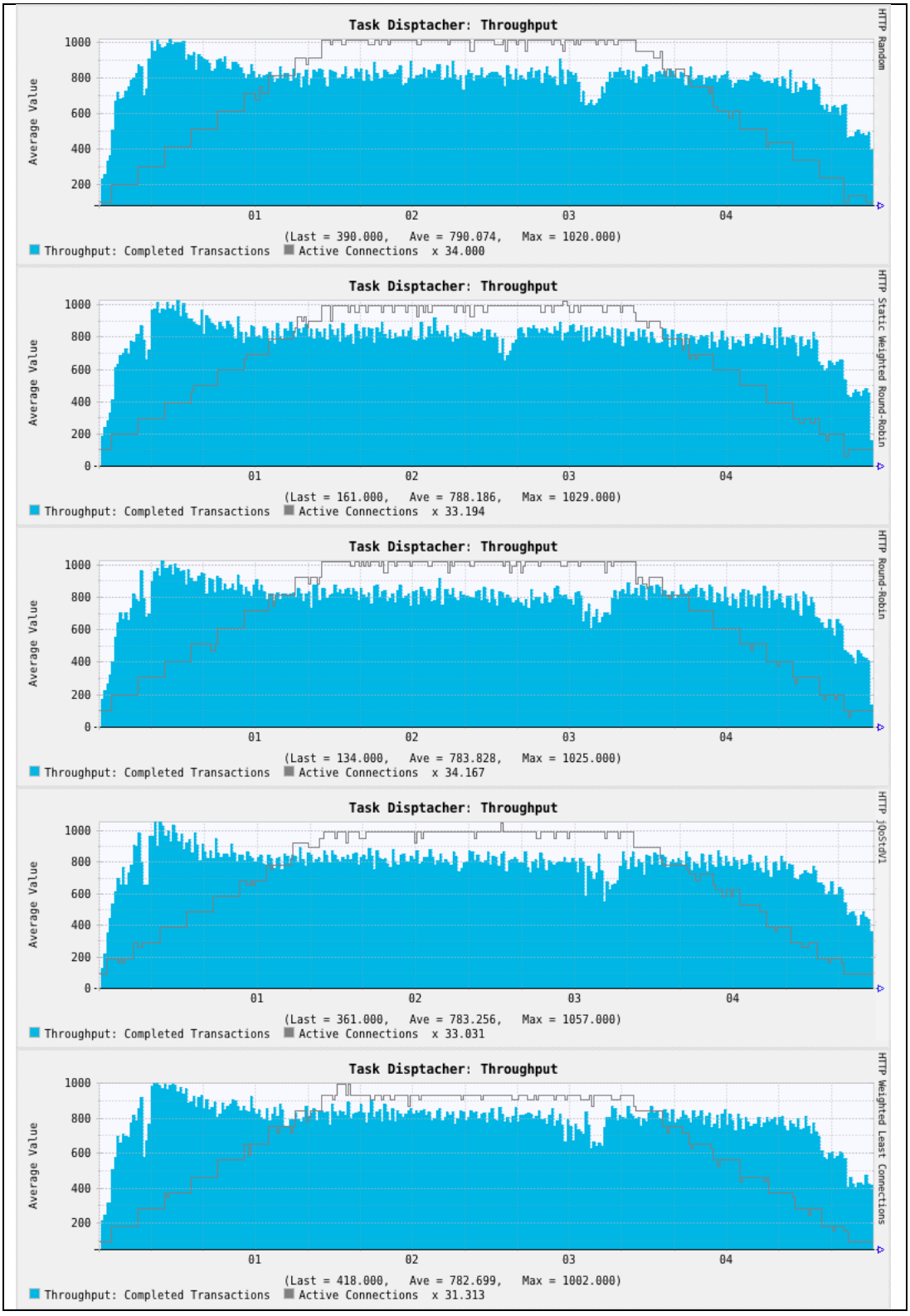
All Scenarios Combined Experiment Results												
Task Assignment Policy	Weight Ratio	Samples	Active Conns	Throughput	Failed	Failed %	Latency	WT1	WT2	PT	Slow-down	Node Sels
TCP Random	1:1:1	214241	21.49	726.24	742.00	0.35	37.53	0.00	0.00	37.52	1.00	25.61
HTTP Random	1:1:1	200539	21.36	677.50	582.00	0.29	38.83	0.00	0.01	38.82	1.00	23.51
TCP Round-Robin	1:1:1	211982	21.29	716.16	725.00	0.34	36.13	0.00	0.01	36.12	1.00	25.14
HTTP Round-Robin	1:1:1	200639	21.21	675.55	538.00	0.27	40.33	0.00	0.00	40.33	1.00	23.33
TCP Static Weighted Round-Robin	2:2:1	253252	21.09	858.48	65.00	0.03	24.11	0.00	0.05	24.06	1.04	27.86
HTTP Static Weighted Round-Robin	2:2:1	236626	21.16	799.41	0.00	0.00	27.03	0.00	0.01	27.02	1.00	25.86
HTTP F5 Dynamic Ratio	1:1:1	199125	21.34	670.46	557.00	0.28	41.25	0.03	0.05	41.17	1.06	23.13
HTTP jQoStdV1Policy	1:1:1	233013	21.41	787.21	0.00	0.00	27.78	0.00	0.00	27.78	1.00	25.44
HTTP Fastest Response Time	1:1:1	233029	21.22	779.36	1708.00	0.73	36.64	0.00	0.06	36.59	1.05	30.21
HTTP Least Loaded	1:1:1	201294	19.89	627.08	2232.00	1.11	60.39	0.00	0.00	60.39	1.00	26.53
HTTP Least Connections	1:1:1	238366	21.29	805.29	0.00	0.00	27.14	0.00	0.01	27.14	1.00	26.03
HTTP Weighted Least Connections	2:2:1	232312	21.27	784.84	0.00	0.00	26.54	0.00	0.00	26.54	1.00	25.38
Min		*199125*	*19.89*	*627.08*	*0.00*	*0.00*	*24.11*	*0.00*	*0.00*	*24.06*	*1.00*	*23.13*
Max		*253252*	*21.49*	*858.48*	*2232.00*	*1.11*	*60.39*	*0.03*	*0.06*	*60.39*	*1.06*	*30.21*
Average		*221202*	*21.17*	*742.30*	*596.17*	*0.28*	*35.31*	*0.00*	*0.02*	*35.29*	*1.01*	*25.67*
Standard Deviation		*18658*	*0.42*	*70.04*	*716.87*	*0.34*	*10.02*	*0.01*	*0.02*	*10.02*	*0.02*	*1.99*

Table 14 - The results of the All Scenarios Combined experiment.

6.3.4 Evaluation of the Baseline experiment

The first experiment result (Table 10) establishes the baseline for all of the task assignment policies, where each VCN was configured identically.




Task Disptacher: Throughput
Average Value
1000
800
600
400
200
01
02
03
04
(Last = 390.000, Ave = 790.074, Max = 1020.000)
Throughput: Completed Transactions
Active Connections x 34.000
HTTP Random
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 161.000, Ave = 788.186, Max = 1029.000)
Throughput: Completed Transactions
Active Connections x 33.194
HTTP Static Weighted Round-Robin
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 134.000, Ave = 783.828, Max = 1025.000)
Throughput: Completed Transactions
Active Connections x 34.167
HTTP Round-Robin
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 361.000, Ave = 783.256, Max = 1057.000)
Throughput: Completed Transactions
Active Connections x 33.031
HTTP jdk5tdv1
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
01
02
03
04
(Last = 418.000, Ave = 782.699, Max = 1002.000)
Throughput: Completed Transactions
Active Connections x 31.313
HTTP Weighted Least Connections

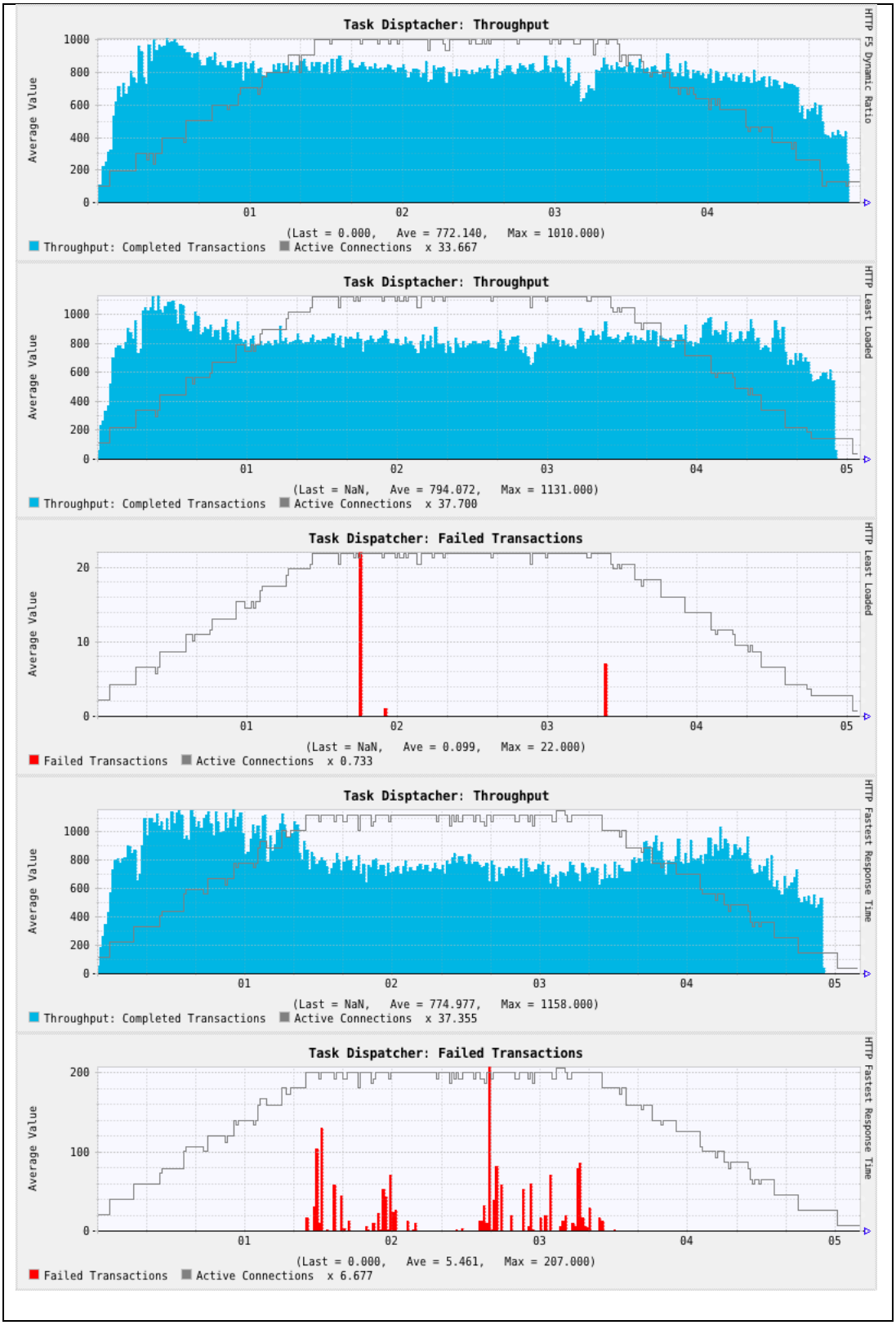


Figure 48 - Throughput (and failure, when applicable) comparison of the policies in the Baseline experiment.

All static WRR-based policies (configured with 2:2:1 ratios) performed really well, confirming the observations provided in Andreolini, Colajanni and Morselli (2002).

The TCP and HTTP Random policies support the findings presented in Casalicchio and Tucci (2001), as their performance was indeed comparable to the static Weighted Round-Robin policies.

Server health-aware (HTTP) implementations of the Random, Round-Robin, and Static Weighted Round-Robin policies achieved a reduced throughput of ~65 fewer transactions per second than their server state-blind (TCP) implementations, due to the additional (waiting time) overheads involved in the consideration of server state health information.

The Least Connections policy performed the best from the set of dynamic server state-aware policies that considered 1 or more server state metrics.

The jQoStdV1 policy resulted in an average latency of ~26.8 milliseconds, comparable to that of the HTTP Static Weighted Round-Robin policy.

The Weighted Least Connections policy produced an average latency of 28.6 milliseconds per request, ~2 milliseconds slower than the jQoStdV1 policy.

The F5 Dynamic Ratio policies' performance was comparable with the Weighted Least Connections policy.

The results for the Least Loaded policy support the findings in Andreolini, Colajanni and Morselli (2002), where the authors suggested that transient server load may result in poor task assignment decisions.

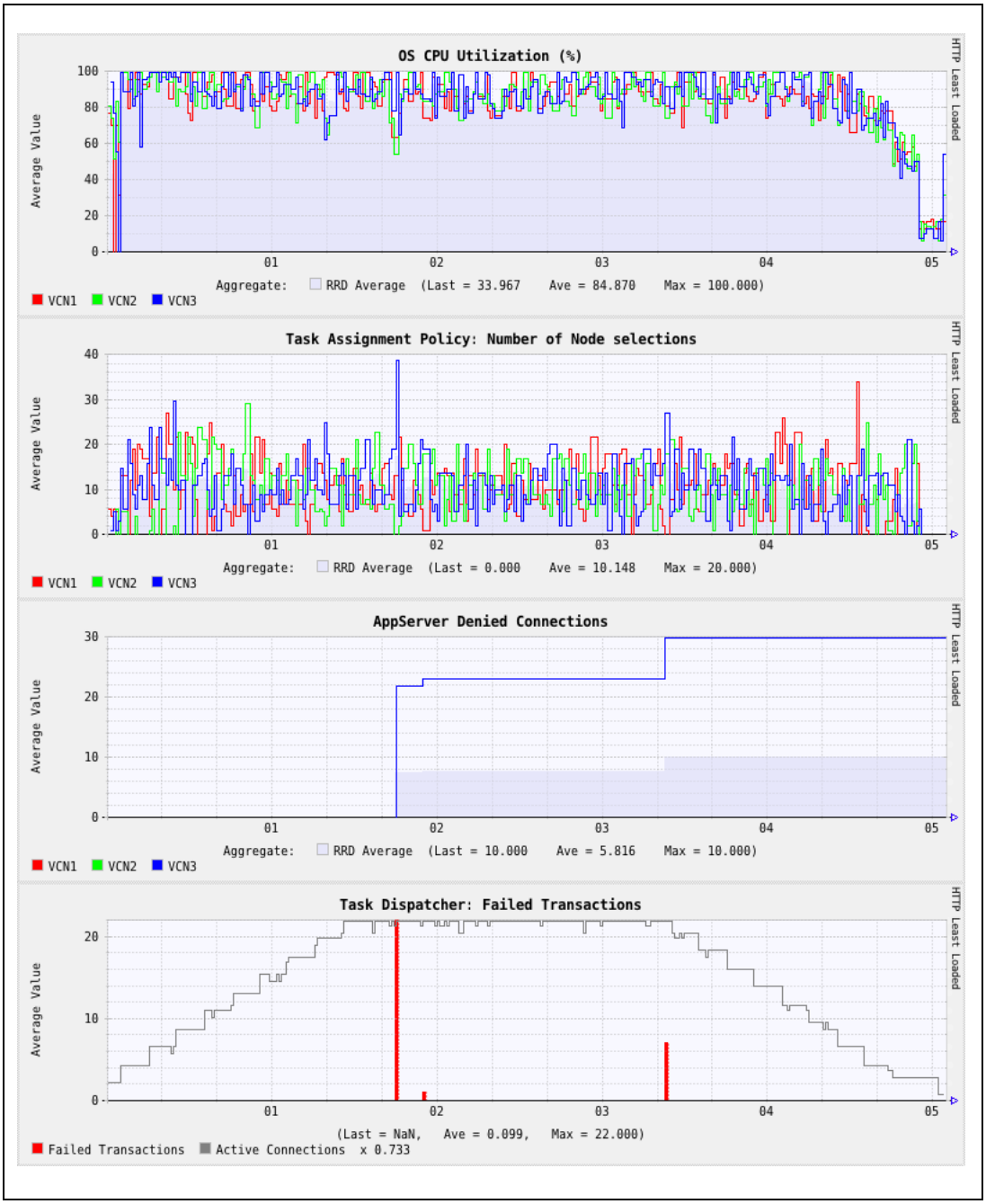


Figure 49 – Failure trace of the Least Loaded policy in the Baseline experiment.

The Fastest Response Time policy produced the poorest task assignment decisions in the experiment, resulting in ~56 times as many failed transactions as the Least Loaded policy.

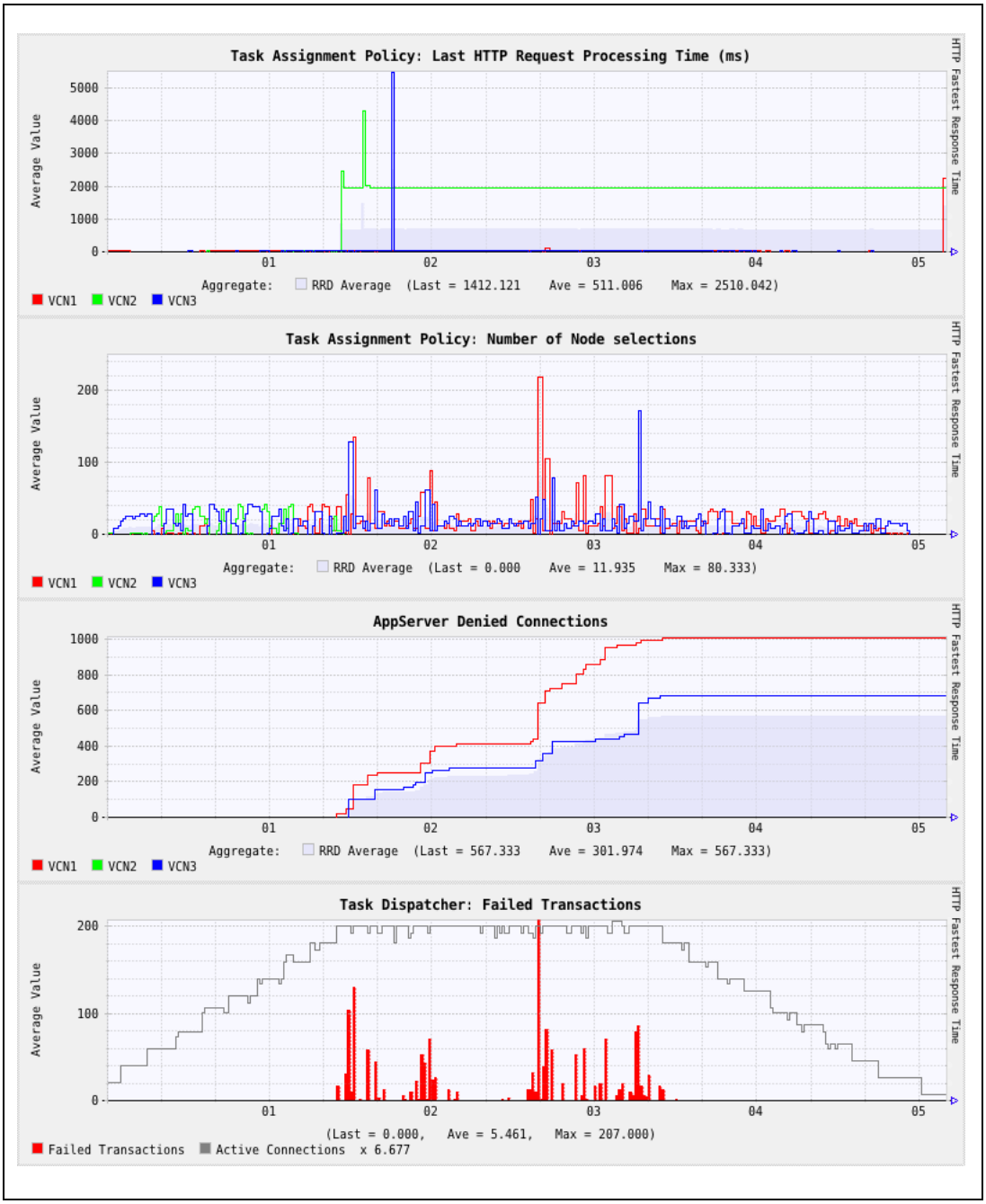
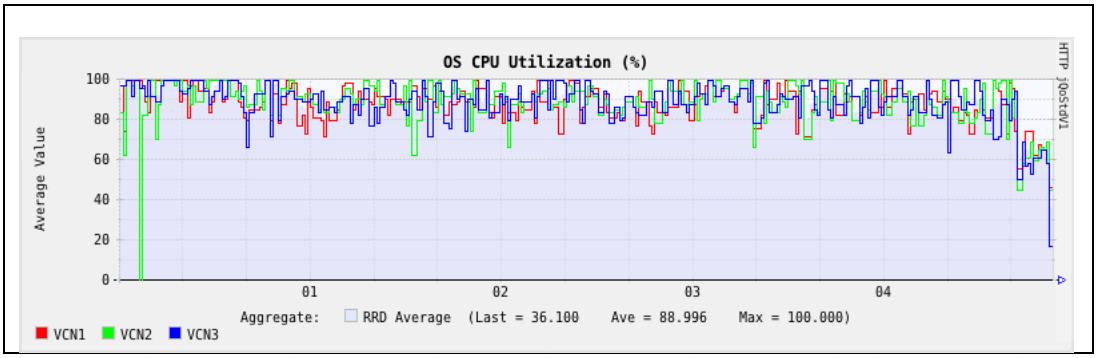
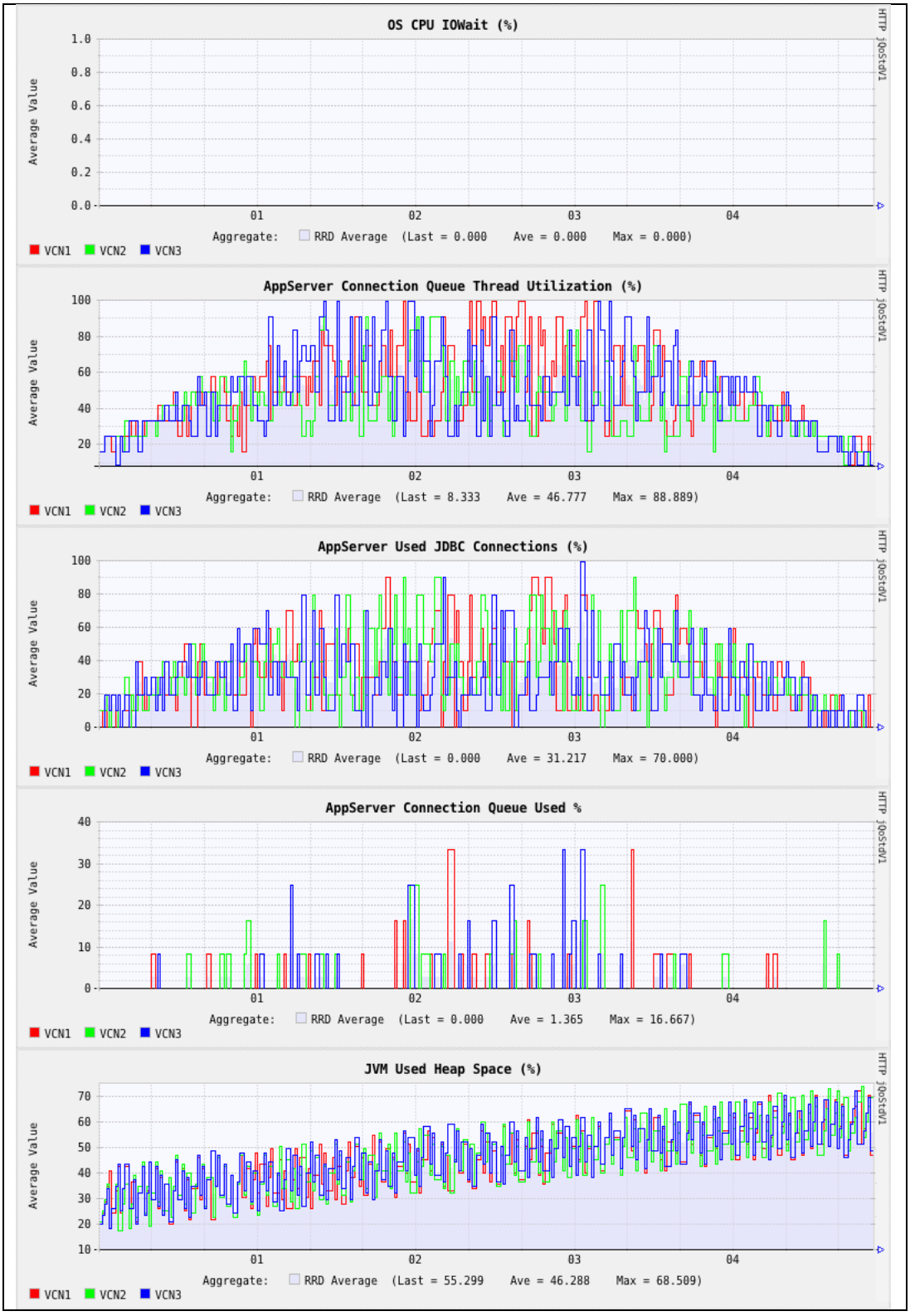


Figure 50 - Failure trace of the Fastest Response Time policy in the Baseline experiment.




OS CPU IOWait (%)
Average Value
1.0
0.8
0.6
0.4
0.2
0.0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 0.000 Max = 0.000)
VCN1 VCN2 VCN3
HTTP jOosStdV1
AppServer Connection Queue Thread Utilization (%)
Average Value
100
80
60
40
20
01
02
03
04
Aggregate: RRD Average (Last = 8.333 Ave = 46.777 Max = 88.889)
VCN1 VCN2 VCN3
HTTP jOosStdV1
AppServer Used JDBC Connections (%)
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 31.217 Max = 70.000)
VCN1 VCN2 VCN3
HTTP jOosStdV1
AppServer Connection Queue Used %
Average Value
40
30
20
10
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 1.365 Max = 16.667)
VCN1 VCN2 VCN3
HTTP jOosStdV1
JVM Used Heap Space (%)
Average Value
70
60
50
40
30
20
10
01
02
03
04
Aggregate: RRD Average (Last = 55.299 Ave = 46.288 Max = 68.509)
VCN1 VCN2 VCN3
HTTP jOosStdV1

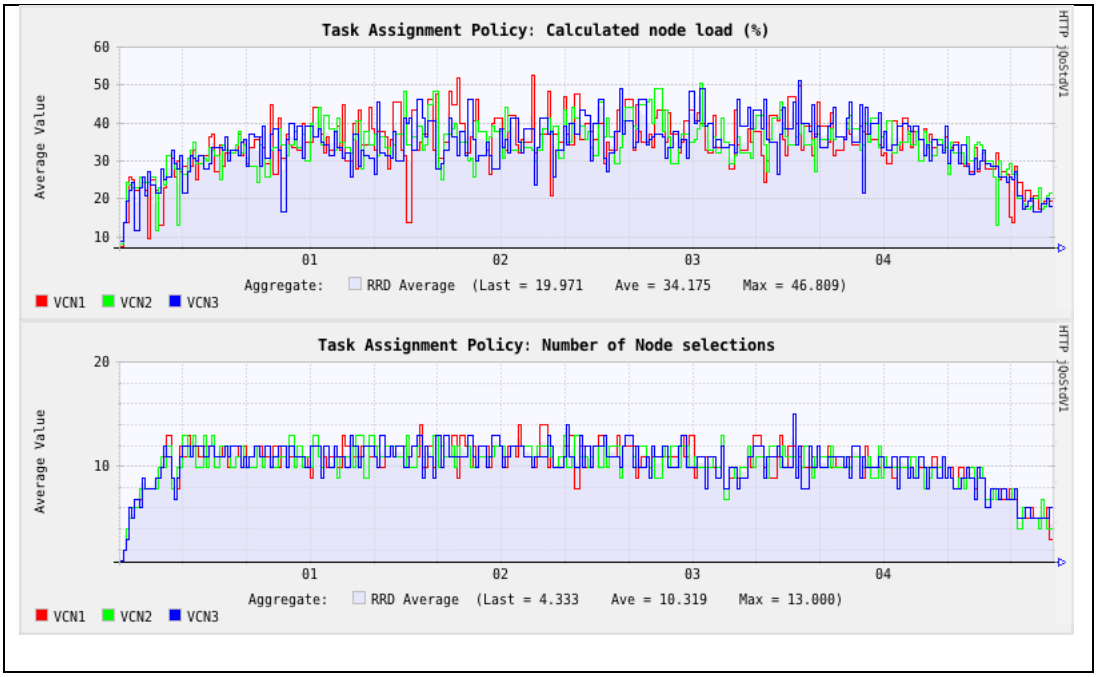
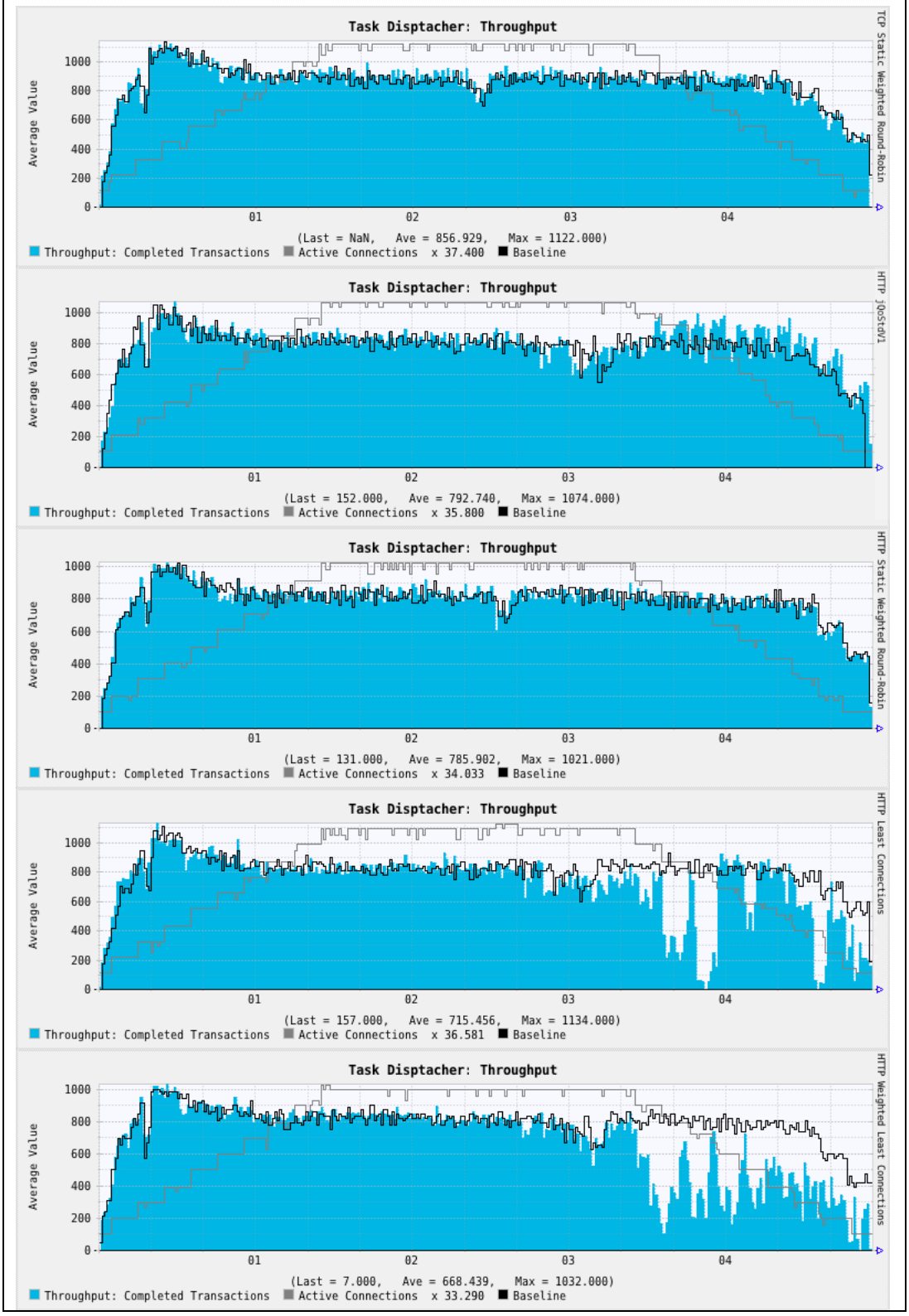


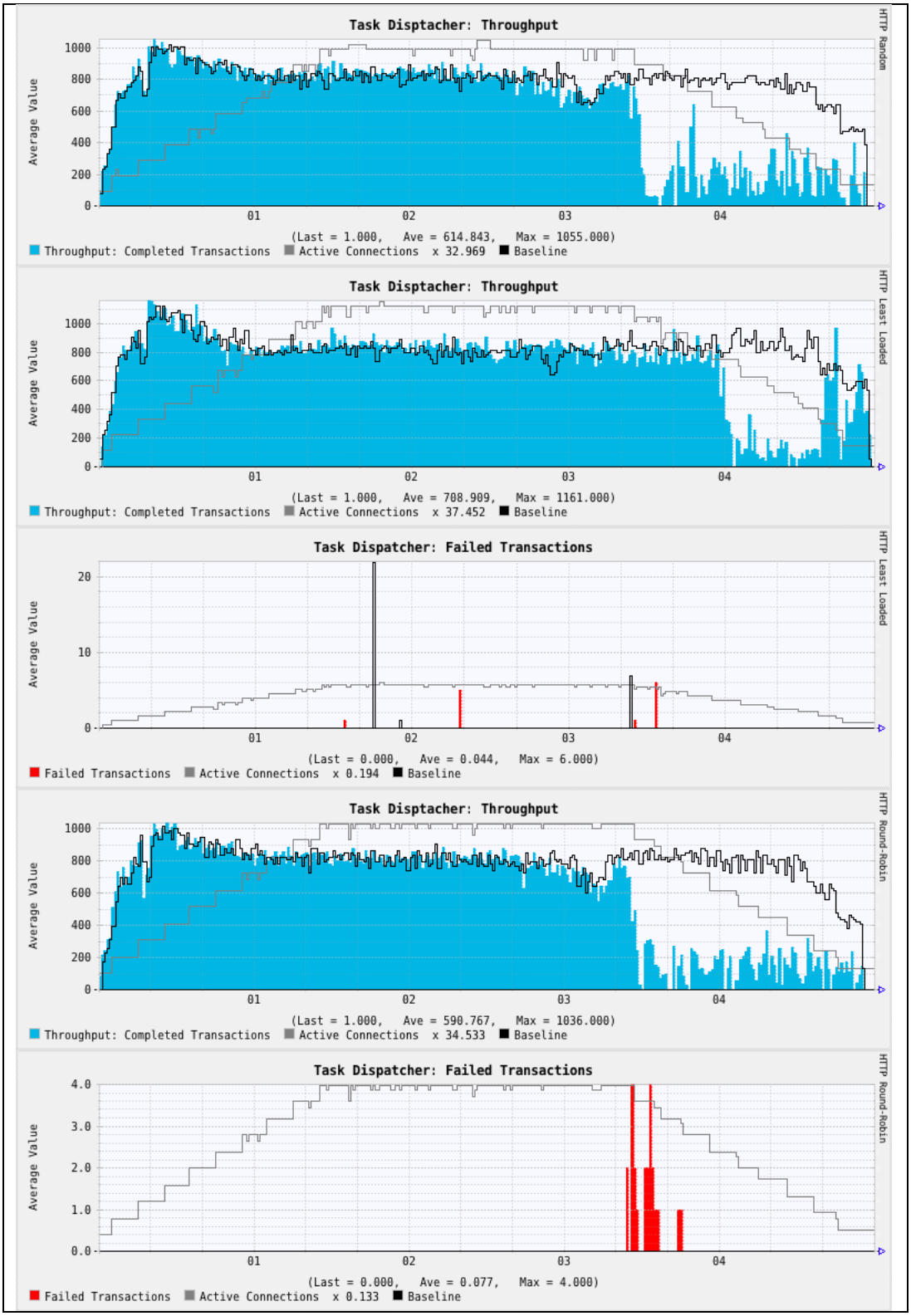
Figure 51 - Trace of the jQoStdV1 server load calculations and resulting server selections in the Baseline experiment, providing efficient dynamic load equalization.

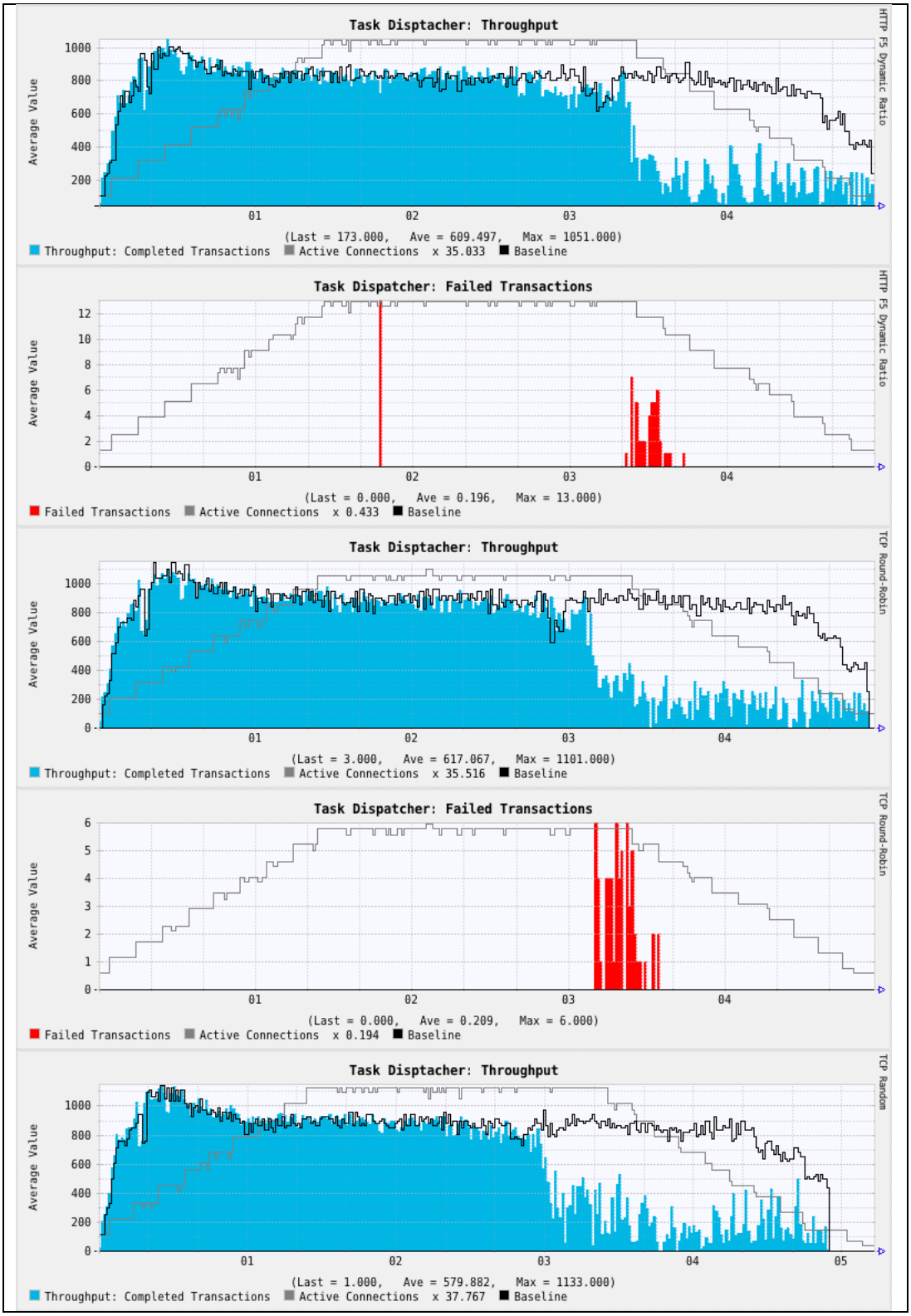
6.3.5 Evaluation of the Heap Space Experiment

Table 11 presents the results of the task assignment policies the Heap Space experiment configuration, where VCN1 and 2 were configured as per the baseline configuration (512MB heap space), and VCN3 configured with half the amount of heap space (256MB).

The heap space scenario was designed to test the efficacy of task assignment policies in a cluster-state where the 3rd node (VCN3) would run out of heap space about halfway into the experiment should it receive the same amount of requests as the other nodes.


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
TCP Static Weighted Round-Robin
(Last = NaN, Ave = 856.929, Max = 1122.000)
Throughput: Completed Transactions
Active Connections x 37.400
Baseline
Task Disptacher: Throughput
HTTP jQsStdV1
(Last = 152.000, Ave = 792.740, Max = 1074.000)
Throughput: Completed Transactions
Active Connections x 35.800
Baseline
Task Disptacher: Throughput
HTTP Static Weighted Round-Robin
(Last = 131.000, Ave = 785.902, Max = 1021.000)
Throughput: Completed Transactions
Active Connections x 34.033
Baseline
Task Disptacher: Throughput
HTTP Least Connections
(Last = 157.000, Ave = 715.456, Max = 1134.000)
Throughput: Completed Transactions
Active Connections x 36.581
Baseline
Task Disptacher: Throughput
HTTP Weighted Least Connections
(Last = 7.000, Ave = 668.439, Max = 1032.000)
Throughput: Completed Transactions
Active Connections x 33.290
Baseline


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
HTTP Random
(Last = 1.000, Ave = 614.843, Max = 1055.000)
Throughput: Completed Transactions
Active Connections x 32.969
Baseline
Task Disptacher: Throughput
HTTP Least Loaded
(Last = 1.000, Ave = 708.909, Max = 1161.000)
Throughput: Completed Transactions
Active Connections x 37.452
Baseline
Task Dispatcher: Failed Transactions
20
10
0
HTTP Least Loaded
(Last = 0.000, Ave = 0.044, Max = 6.000)
Failed Transactions
Active Connections x 0.194
Baseline
Task Disptacher: Throughput
HTTP Round-Robin
(Last = 1.000, Ave = 590.767, Max = 1036.000)
Throughput: Completed Transactions
Active Connections x 34.533
Baseline
Task Dispatcher: Failed Transactions
4.0
3.0
2.0
1.0
0.0
HTTP Round-Robin
(Last = 0.000, Ave = 0.077, Max = 4.000)
Failed Transactions
Active Connections x 0.133
Baseline


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
01
02
03
04
HTTP F5 Dynamic Ratio
(Last = 173.000, Ave = 609.497, Max = 1051.000)
Throughput: Completed Transactions
Active Connections x 35.033
Baseline
Task Dispatcher: Failed Transactions
Average Value
12
10
8
6
4
2
0
01
02
03
04
HTTP F5 Dynamic Ratio
(Last = 0.000, Ave = 0.196, Max = 13.000)
Failed Transactions
Active Connections x 0.433
Baseline
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
TCP Round-Robin
(Last = 3.000, Ave = 617.067, Max = 1101.000)
Throughput: Completed Transactions
Active Connections x 35.516
Baseline
Task Dispatcher: Failed Transactions
Average Value
6
5
4
3
2
1
0
01
02
03
04
TCP Round-Robin
(Last = 0.000, Ave = 0.209, Max = 6.000)
Failed Transactions
Active Connections x 0.194
Baseline
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
05
TCP Random
(Last = 1.000, Ave = 579.882, Max = 1133.000)
Throughput: Completed Transactions
Active Connections x 37.767
Baseline

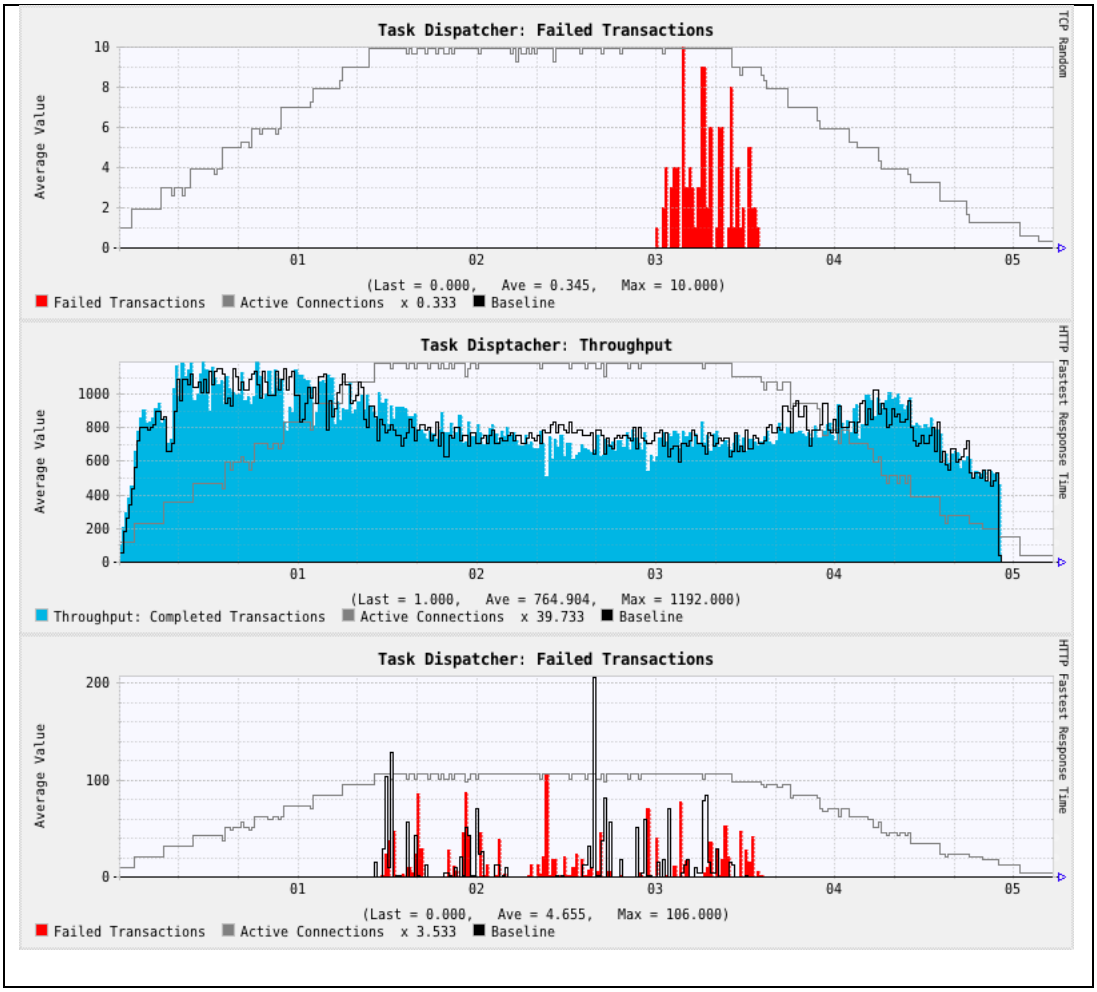
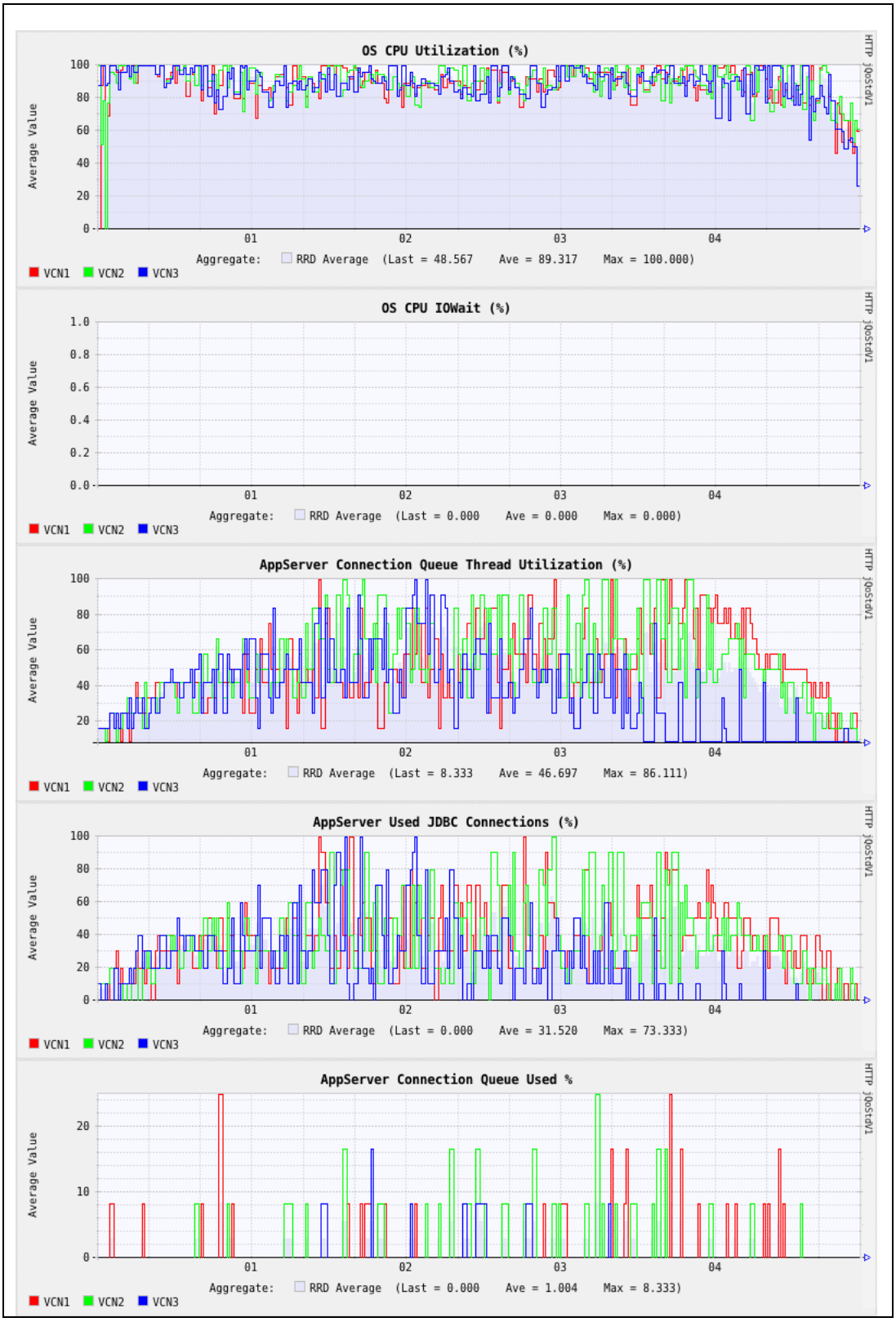


Figure 52 - Throughput (with Baseline and failure, where applicable) comparison of the policies in the Heap Space experiment.

All of the HTTP policies benefitted from the use of the NMA server health indicator; effectively preventing assignments to nodes for which NMA samples could not be received from the URL-based NMA sampler.

The jQoStdV1 policy produced excellent results, and performed the best from the set of dynamic policies, with an average throughput of 793 transactions per second. The consideration of heap space and the limiting of connections based upon a heap space utilization threshold of 86%, proved to be sensitive enough to prevent over-utilization while not impacting throughput.


OS CPU Utilization (%)
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 48.567 Ave = 89.317 Max = 100.000)
VCN1 VCN2 VCN3
HTTP jQoSTdV1
OS CPU IOWait (%)
Average Value
1.0
0.8
0.6
0.4
0.2
0.0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 0.000 Max = 0.000)
VCN1 VCN2 VCN3
HTTP jQoSTdV1
AppServer Connection Queue Thread Utilization (%)
Average Value
100
80
60
40
20
01
02
03
04
Aggregate: RRD Average (Last = 8.333 Ave = 46.697 Max = 86.111)
VCN1 VCN2 VCN3
HTTP jQoSTdV1
AppServer Used JDBC Connections (%)
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 31.520 Max = 73.333)
VCN1 VCN2 VCN3
HTTP jQoSTdV1
AppServer Connection Queue Used %
Average Value
20
10
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 1.004 Max = 8.333)
VCN1 VCN2 VCN3
HTTP jQoSTdV1

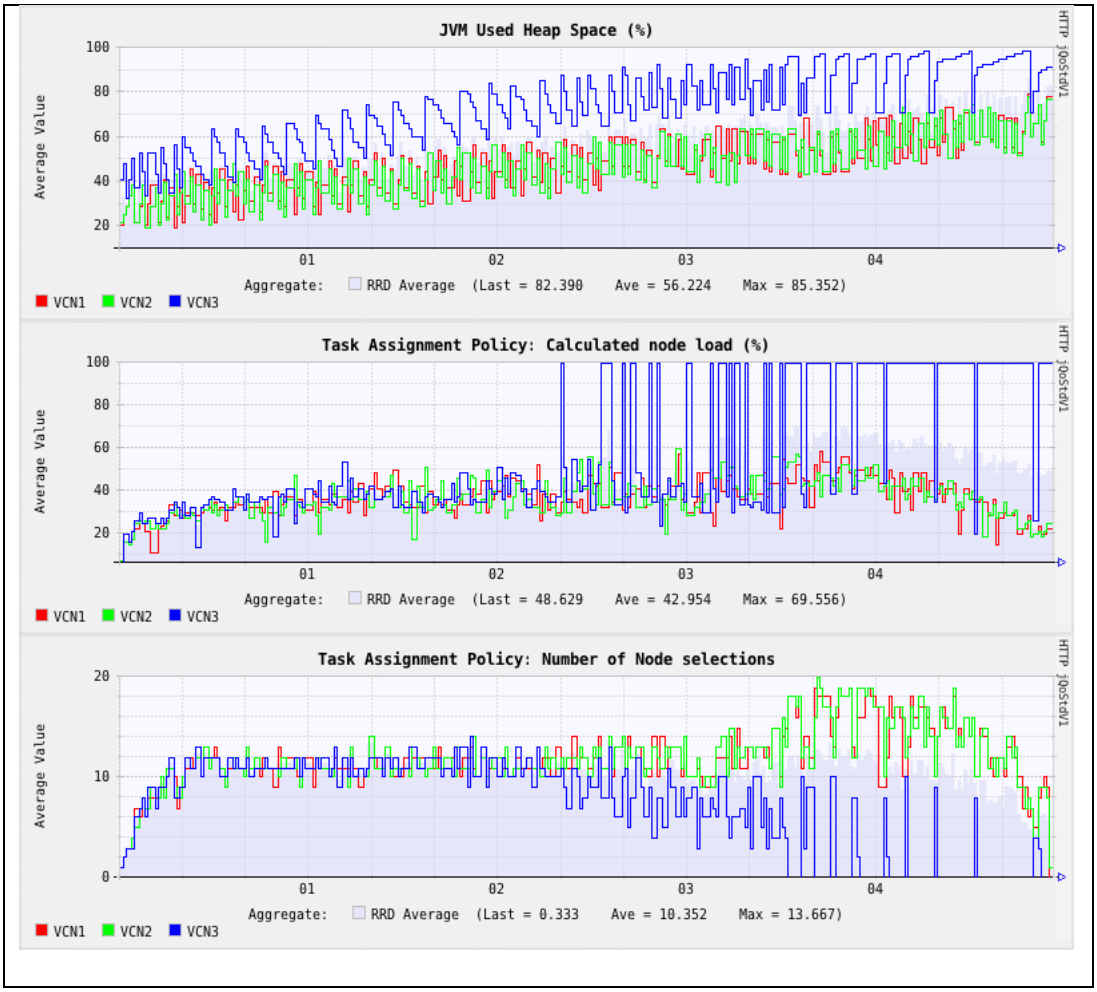


Figure 53 - Trace of the jQoStdV1 server load calculation and resulting server selections in the Heap Space Test, highlighting jQoStdV1's success in preventing assignments to over-utilized nodes.

The remaining policies in the test failed because they were unable to consider the over-utilization of heap space on VCN3, leading to an increase in connection queuing, and finally, failures in the form of denied connections and request processing timeouts.

6.3.6 Evaluation of the HTTP connection queue worker threads experiment

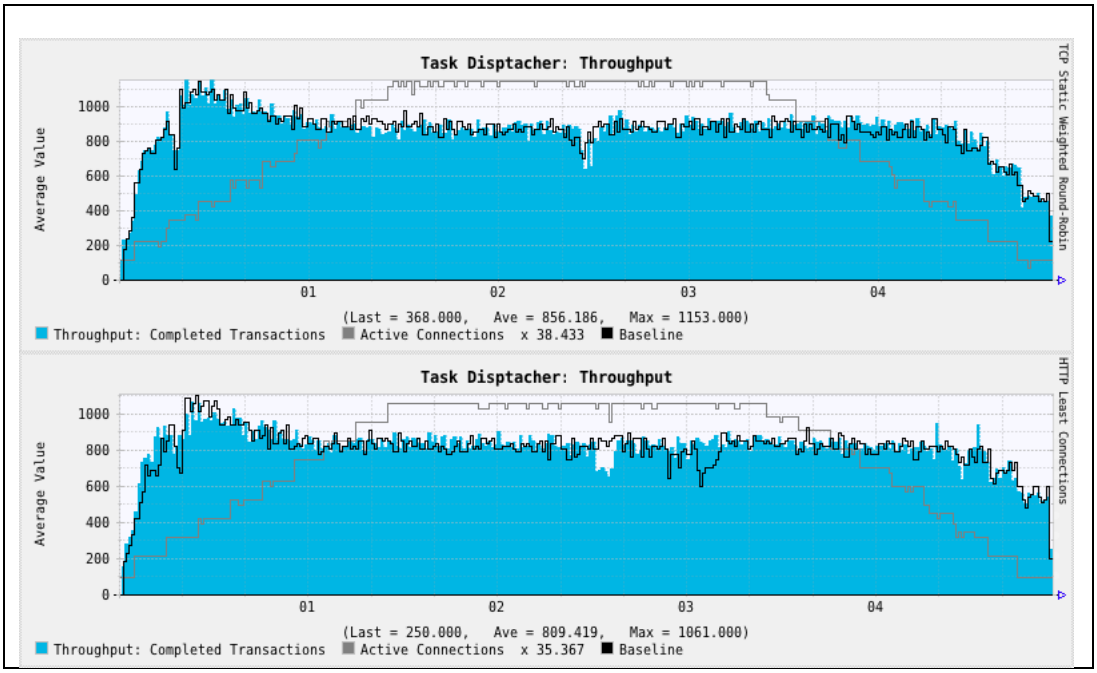
Table 12 presents the results of the task assignment policies in the HTTP connection queue worker threads experiment, where VCN1 and 2 were configured as per the baseline configuration (12 HTTP connection queue

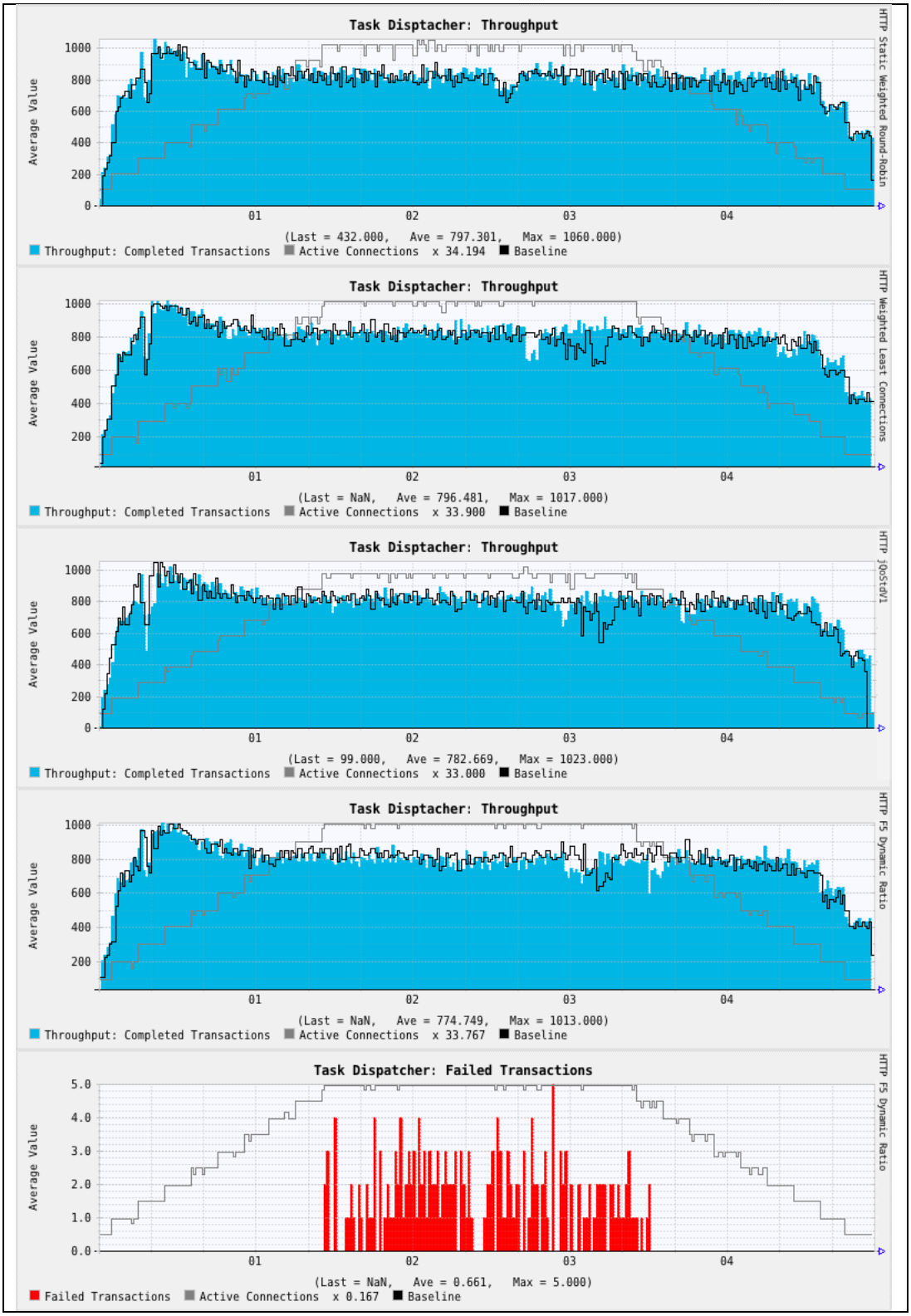
worker threads, with a maximum queue size of 12), and VCN3 configured with half the amount of worker threads (6), and also a maximum queue size of 12.

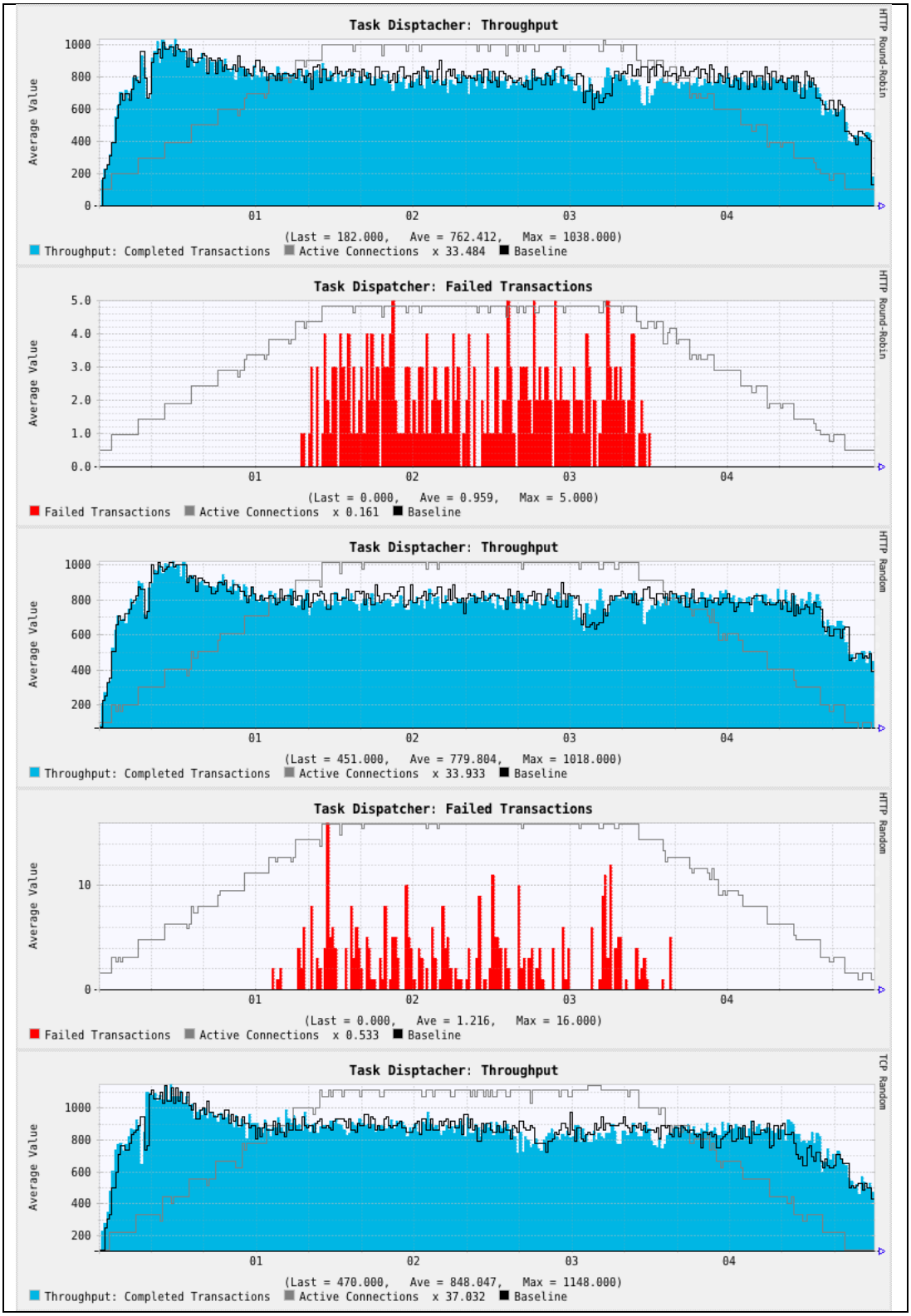
This experiment was designed to test the efficacy of task assignment policies in a cluster-state where the 3rd node (VCN3) would run out of HTTP connection queue worker threads about halfway into the experiment, should it receive the same amount of connections as the other nodes.

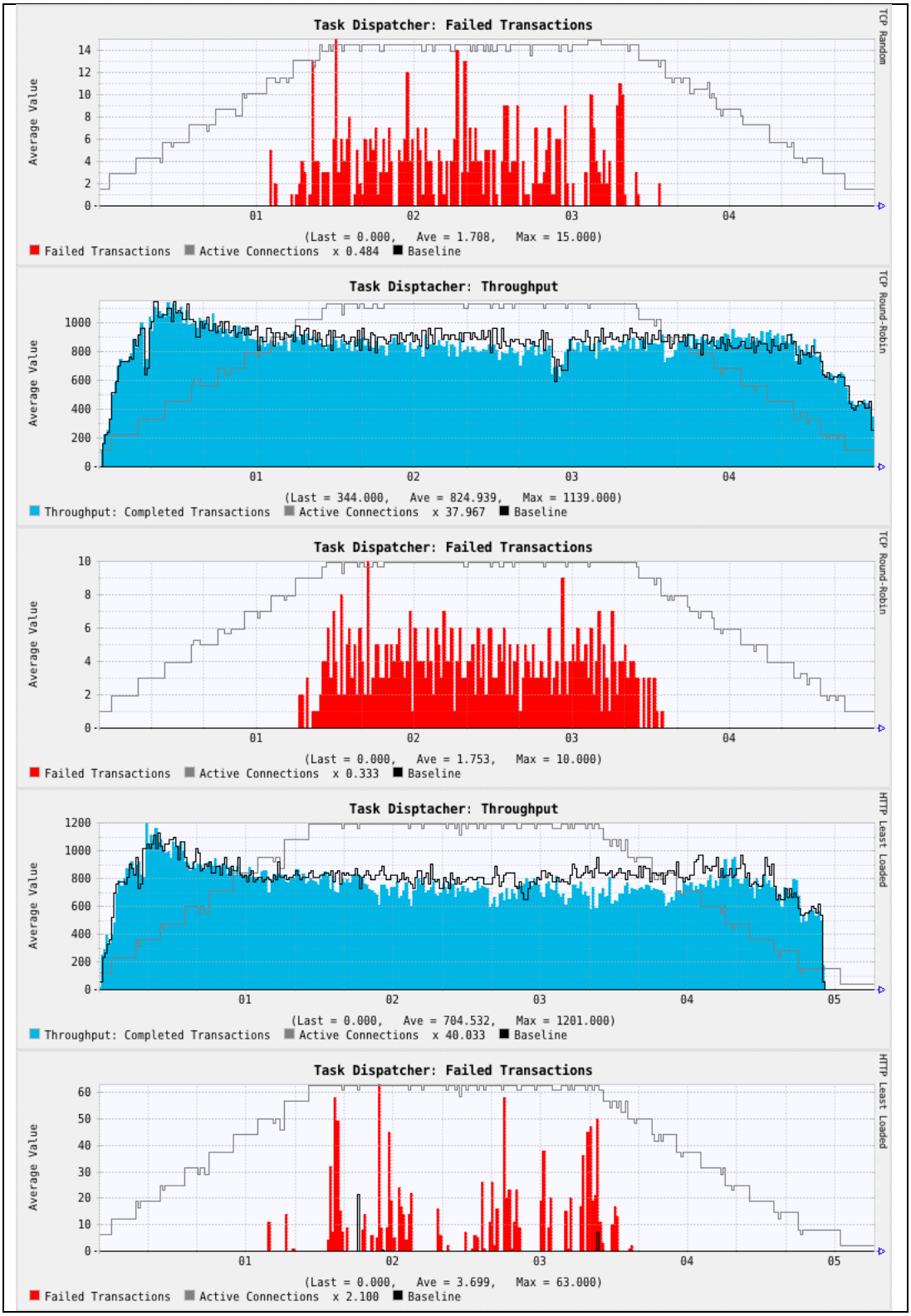
The over-utilisation of HTTP connection queue worker threads would lead to increased utilization of the HTTP connection queue where new connections will be rejected whenever the maximum queue size has been reached.

The TCP and HTTP Static Weighted Round-Robin, Least Connections, Weighted Least Connections and jQoStdV1 policies produced no failed requests.




Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 432.000, Ave = 797.301, Max = 1060.000)
Throughput: Completed Transactions
Active Connections x 34.194
Baseline
HTTP Static Weighted Round-Robin
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
01
02
03
04
(Last = NaN, Ave = 796.481, Max = 1017.000)
Throughput: Completed Transactions
Active Connections x 33.900
Baseline
HTTP Weighted Least Connections
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 99.000, Ave = 782.669, Max = 1023.000)
Throughput: Completed Transactions
Active Connections x 33.000
Baseline
HTTP jQuStd/1
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
01
02
03
04
(Last = NaN, Ave = 774.749, Max = 1013.000)
Throughput: Completed Transactions
Active Connections x 33.767
Baseline
HTTP F5 Dynamic Ratio
Task Dispatcher: Failed Transactions
Average Value
5.0
4.0
3.0
2.0
1.0
0.0
01
02
03
04
(Last = NaN, Ave = 0.661, Max = 5.000)
Failed Transactions
Active Connections x 0.167
Baseline
HTTP F5 Dynamic Ratio


Task Disptacher: Throughput
HTTP Round-Robin
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 182.000, Ave = 762.412, Max = 1038.000)
Throughput: Completed Transactions
Active Connections x 33.484
Baseline
Task Dispatcher: Failed Transactions
HTTP Round-Robin
Average Value
5.0
4.0
3.0
2.0
1.0
0.0
01
02
03
04
(Last = 0.000, Ave = 0.959, Max = 5.000)
Failed Transactions
Active Connections x 0.161
Baseline
Task Disptacher: Throughput
HTTP Random
Average Value
1000
800
600
400
200
01
02
03
04
(Last = 451.000, Ave = 779.804, Max = 1018.000)
Throughput: Completed Transactions
Active Connections x 33.933
Baseline
Task Dispatcher: Failed Transactions
HTTP Random
Average Value
10
0
01
02
03
04
(Last = 0.000, Ave = 1.216, Max = 16.000)
Failed Transactions
Active Connections x 0.533
Baseline
Task Disptacher: Throughput
TCP Random
Average Value
1000
800
600
400
200
01
02
03
04
(Last = 470.000, Ave = 848.047, Max = 1148.000)
Throughput: Completed Transactions
Active Connections x 37.032
Baseline


Task Dispatcher: Failed Transactions
TCP Random
Average Value
(Last = 0.000, Ave = 1.708, Max = 15.000)
Failed Transactions
Active Connections x 0.484
Baseline
Task Disptacher: Throughput
TCP Round-Robin
Average Value
(Last = 344.000, Ave = 824.939, Max = 1139.000)
Throughput: Completed Transactions
Active Connections x 37.967
Baseline
Task Dispatcher: Failed Transactions
TCP Round-Robin
Average Value
(Last = 0.000, Ave = 1.753, Max = 10.000)
Failed Transactions
Active Connections x 0.333
Baseline
Task Disptacher: Throughput
HTTP Least Loaded
Average Value
(Last = 0.000, Ave = 704.532, Max = 1201.000)
Throughput: Completed Transactions
Active Connections x 40.033
Baseline
Task Dispatcher: Failed Transactions
HTTP Least Loaded
Average Value
(Last = 0.000, Ave = 3.699, Max = 63.000)
Failed Transactions
Active Connections x 2.100
Baseline

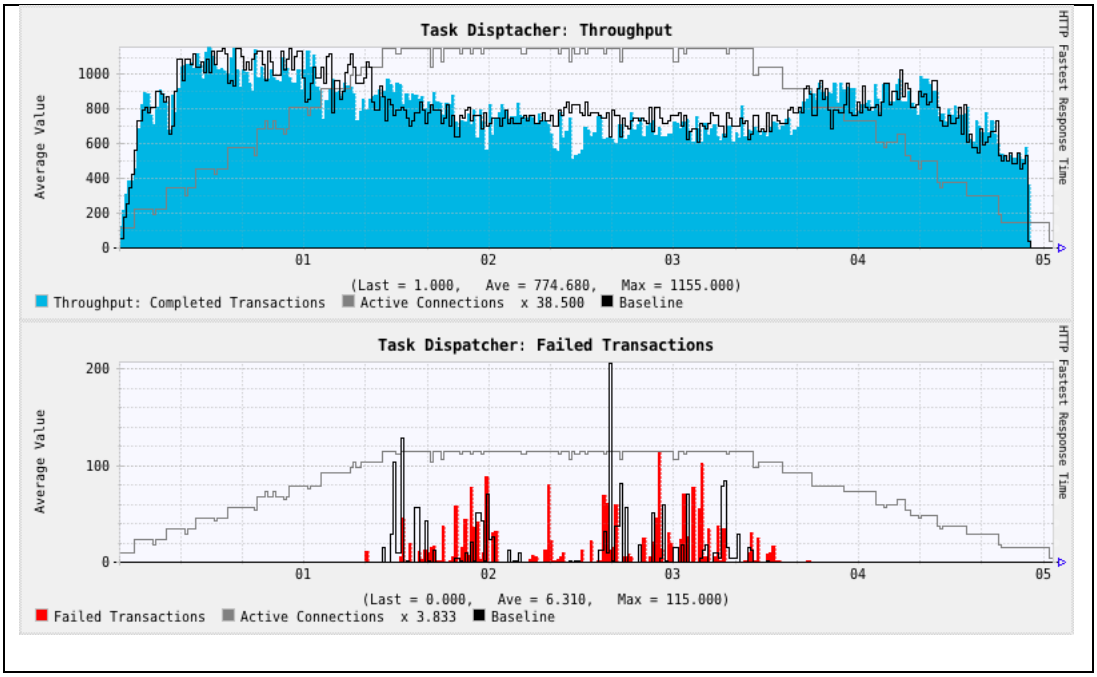
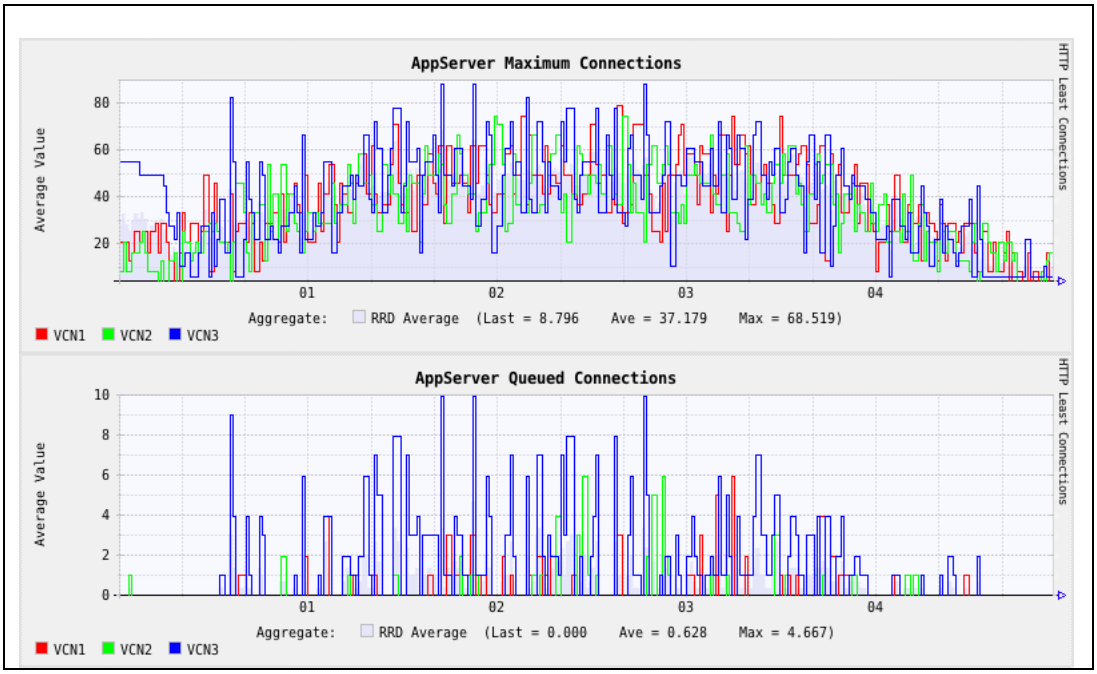


Figure 54 - Throughput (with Baseline and failure, where applicable) comparison of the policies in the HTTP request queue worker threads experiment.

The Least Connections policy produced excellent results in this experiment, as it considers the percentage of total connections that each server has, where the impact of queuing on VCN3 would result in more connections to VCN 1 and 2, avoiding failures on VCN3.



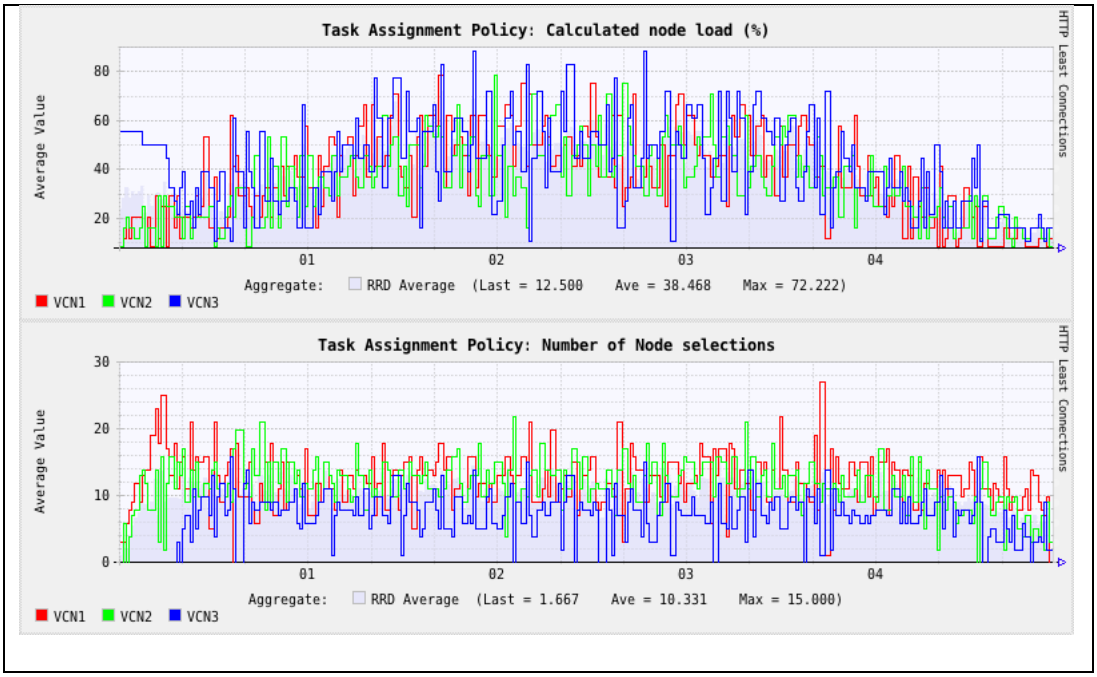
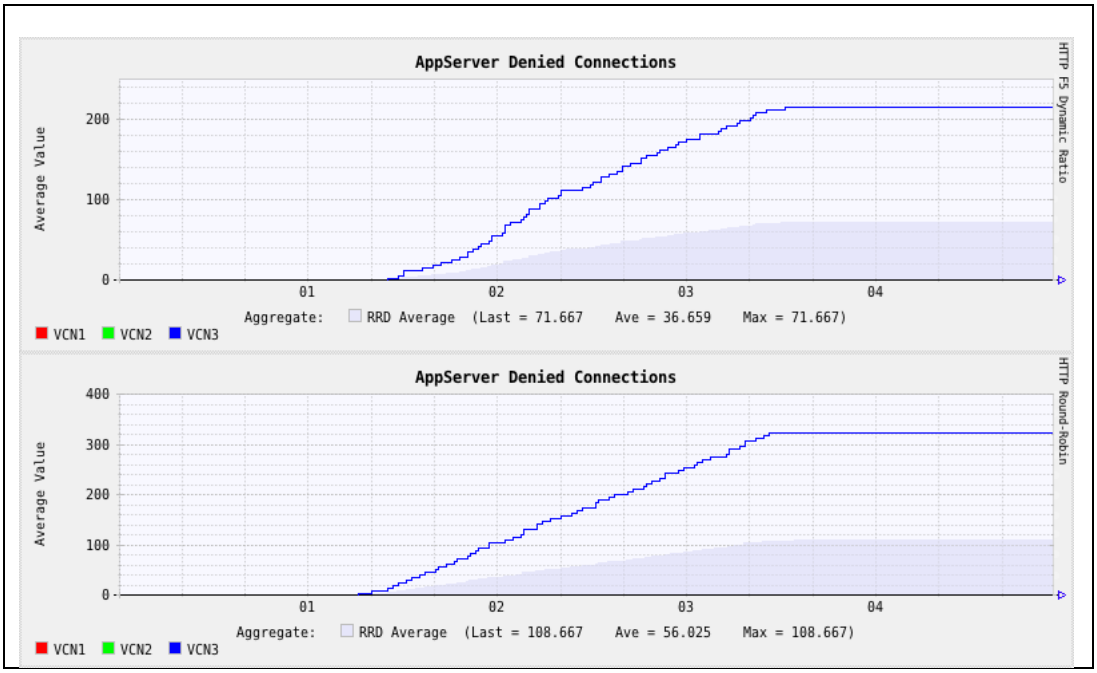


Figure 55 - The load calculation of the Least Connections policy in the HTTP connection queue threads experiment.

The remaining policies all exceeded the maximum connection queue size, and resulted in denied connections on VCN3, with exception of the Fastest Response Time policy that produced denied connections on all servers.



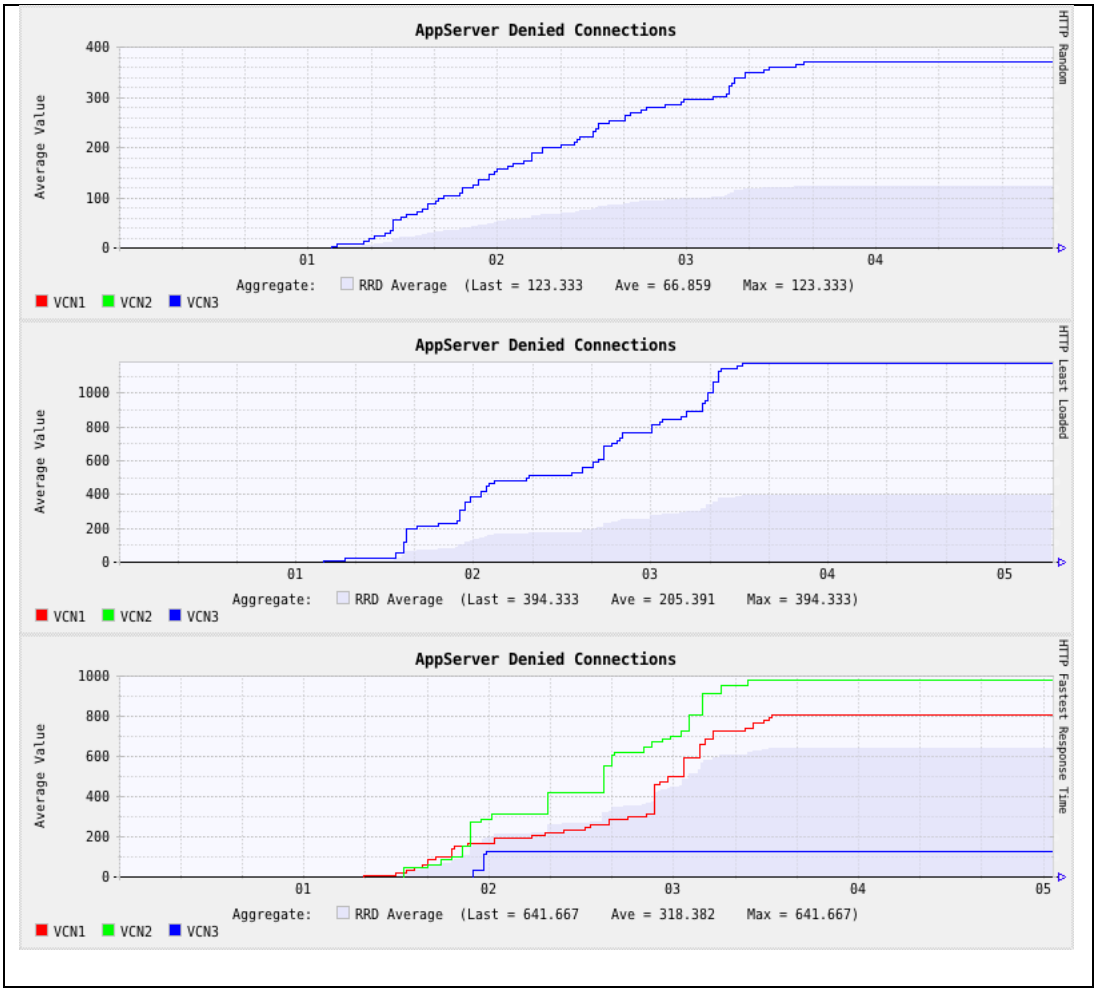
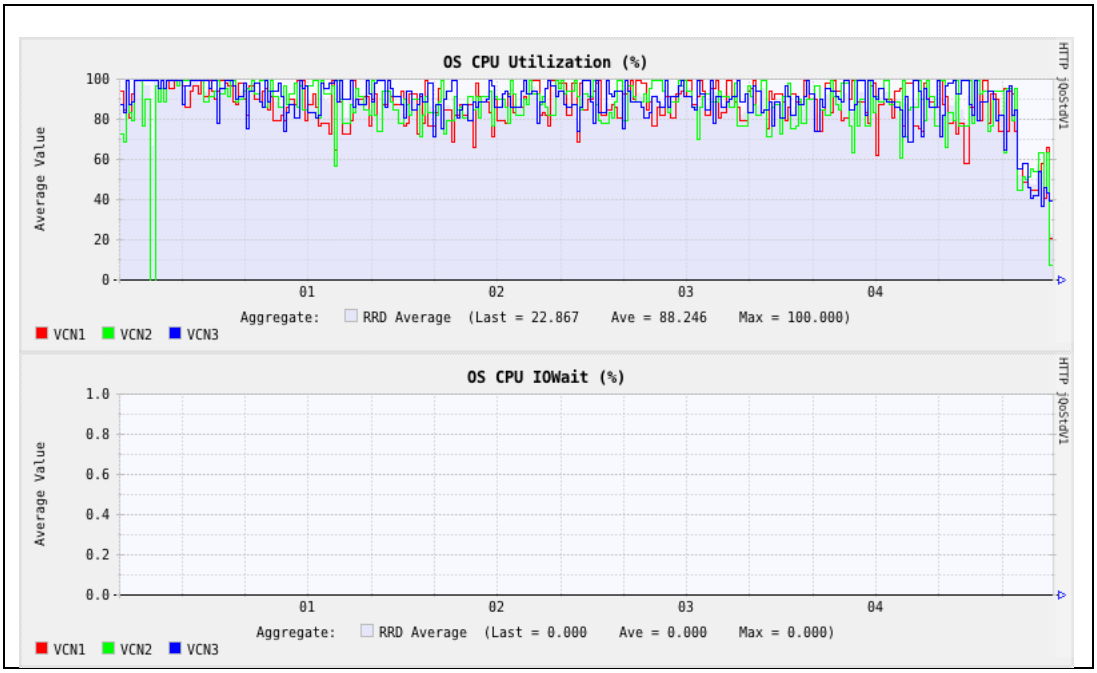
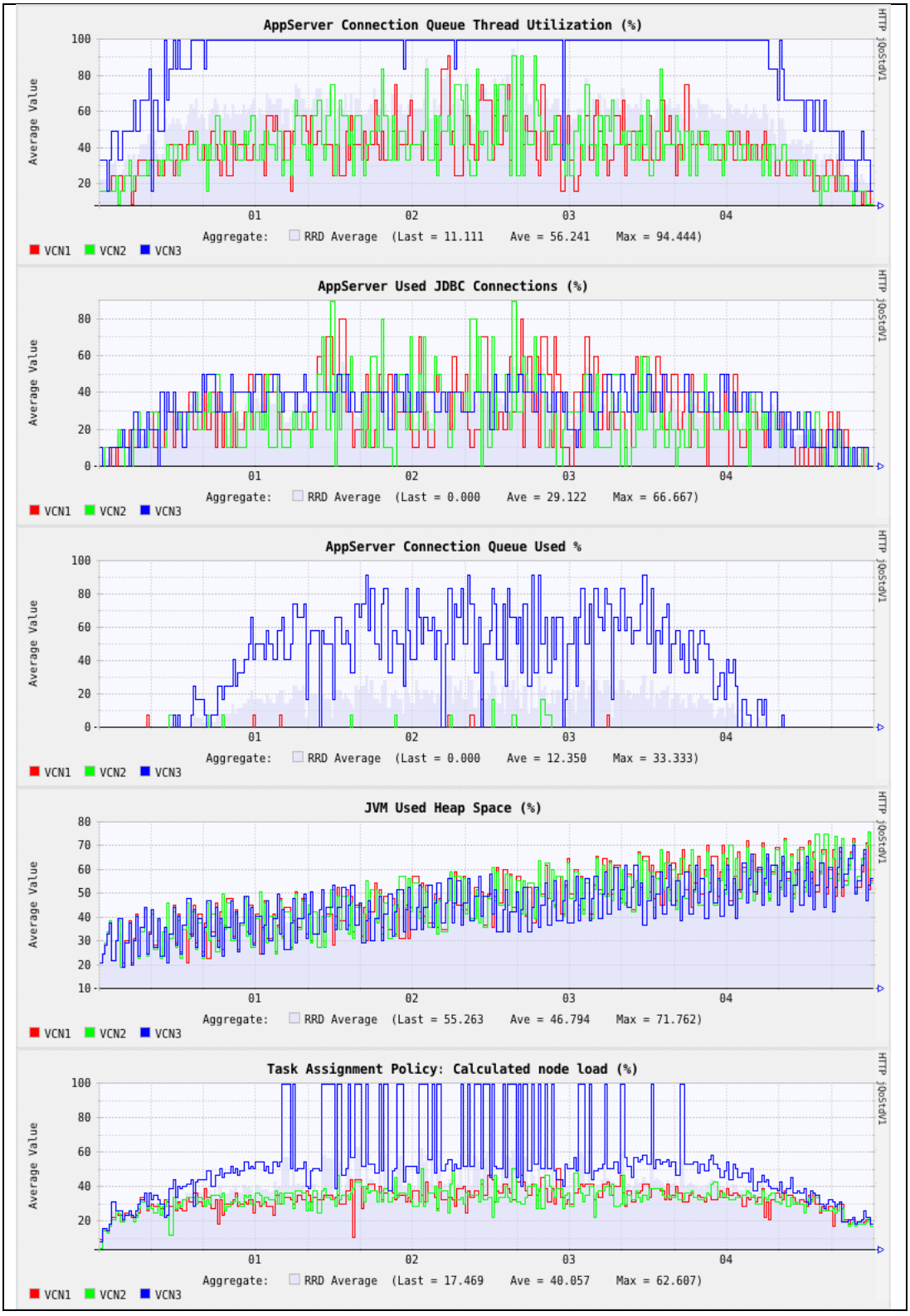


Figure 56 - Failure trace of the HTTP policies in the HTTP connection queue worker threads experiment. (No traces are available for the TCP policies.)




AppServer Connection Queue Thread Utilization (%)
Average Value
100
80
60
40
20
01
02
03
04
Aggregate: RRD Average (Last = 11.111 Ave = 56.241 Max = 94.444)
VCN1 VCN2 VCN3
HTTP jQosStdV1
AppServer Used JDBC Connections (%)
Average Value
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 29.122 Max = 66.667)
VCN1 VCN2 VCN3
HTTP jQosStdV1
AppServer Connection Queue Used %
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 12.350 Max = 33.333)
VCN1 VCN2 VCN3
HTTP jQosStdV1
JVM Used Heap Space (%)
Average Value
80
70
60
50
40
30
20
10
01
02
03
04
Aggregate: RRD Average (Last = 55.263 Ave = 46.794 Max = 71.762)
VCN1 VCN2 VCN3
HTTP jQosStdV1
Task Assignment Policy: Calculated node load (%)
Average Value
100
80
60
40
20
01
02
03
04
Aggregate: RRD Average (Last = 17.469 Ave = 40.057 Max = 62.607)
VCN1 VCN2 VCN3
HTTP jQosStdV1

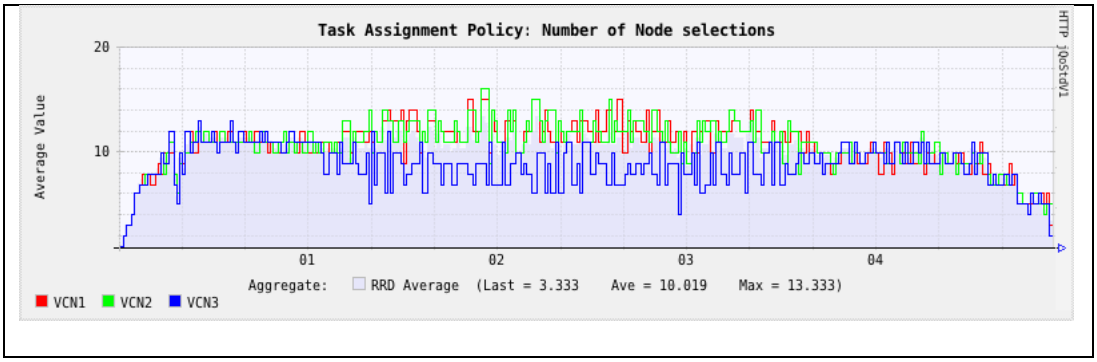


Figure 57 - Trace of the jQoStdV1 server load calculation and resulting server selections in the HTTP connection queue worker threads experiment.

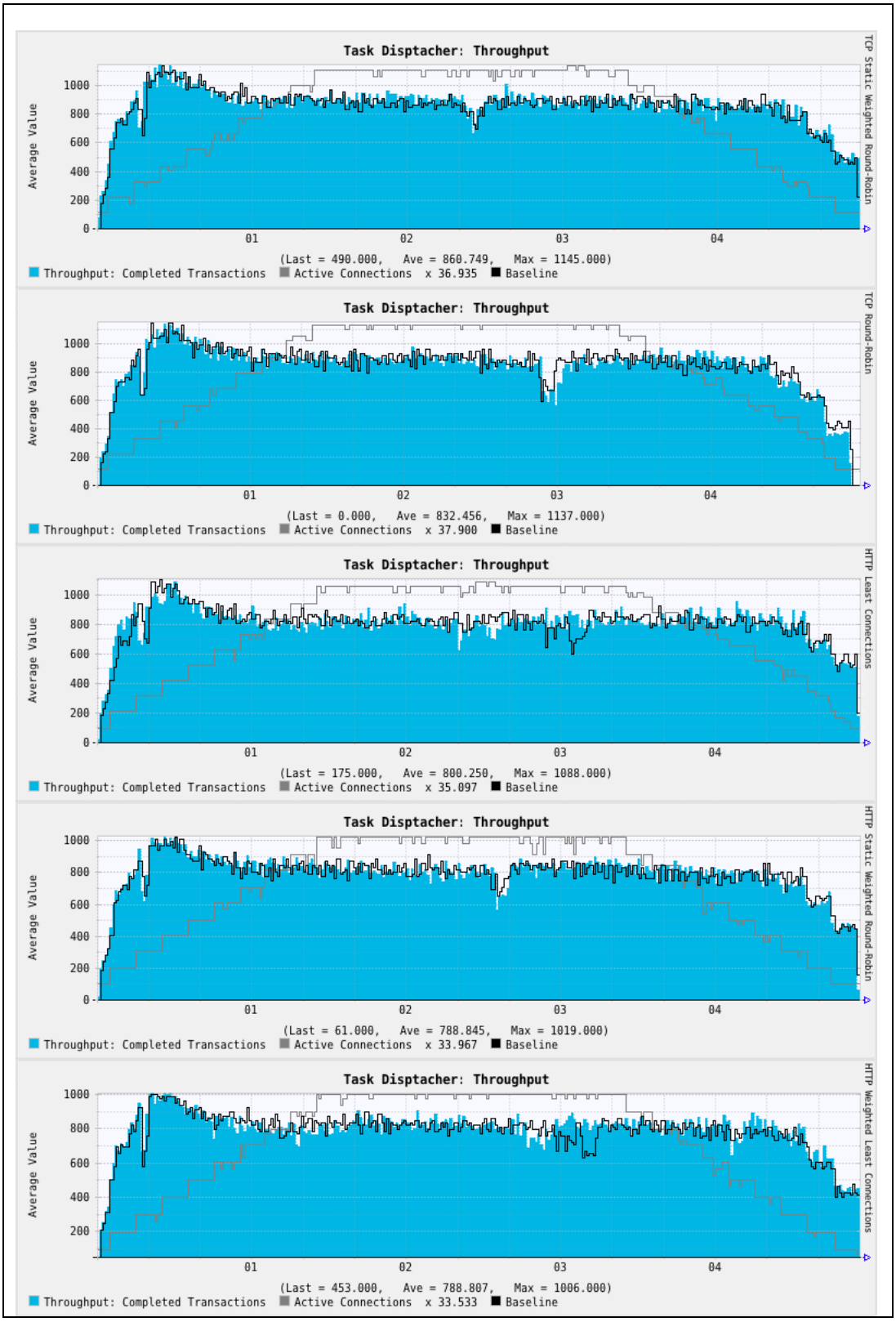
6.3.7 Evaluation of the JDBC connection threads experiment

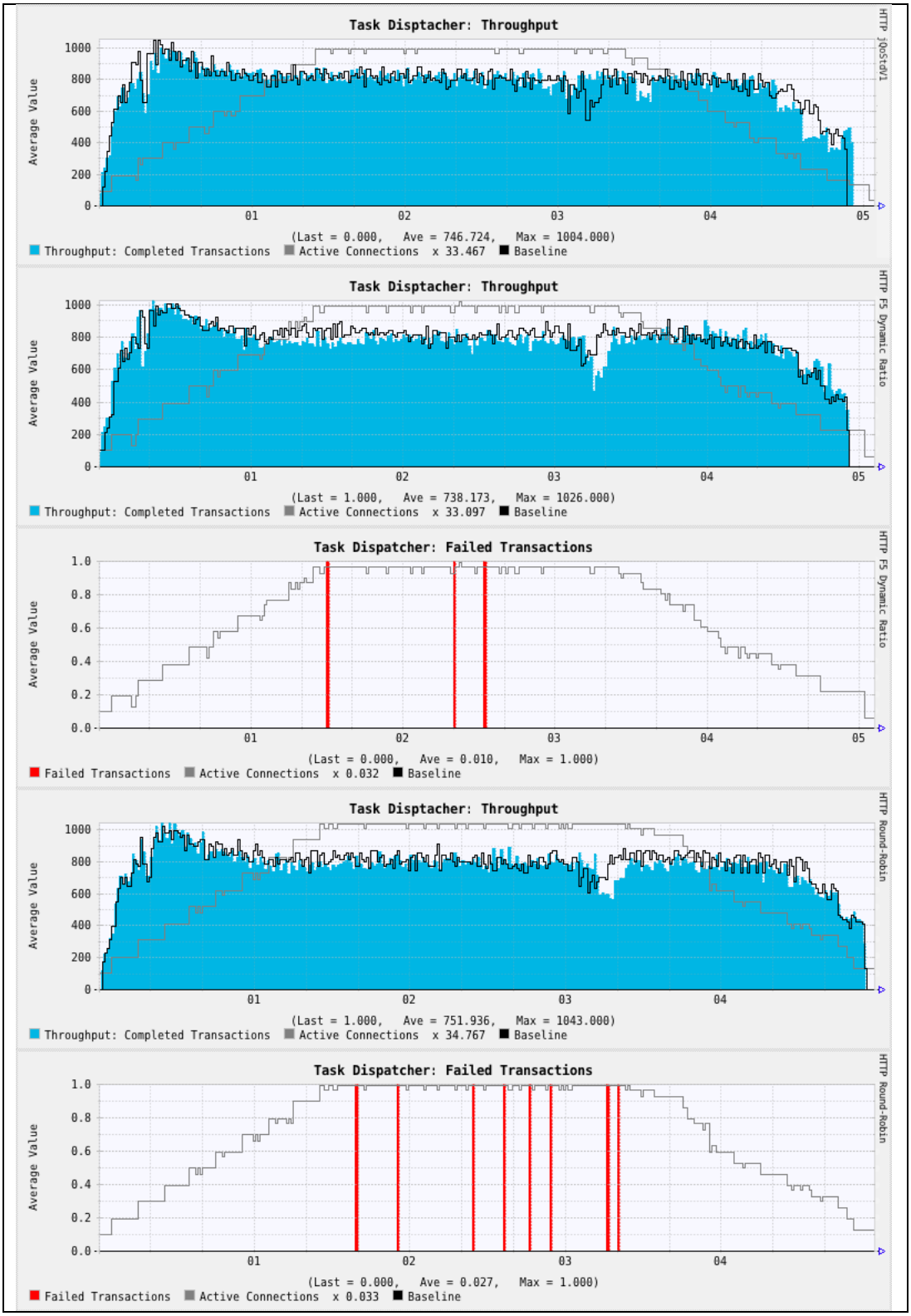
Table 13 presents the results of the task assignment policies in the JDBC connection threads experiment, where VCN1 and 2 were configured as per the baseline configuration (10 database connections), and VCN3 configured with half the amount connections (5).

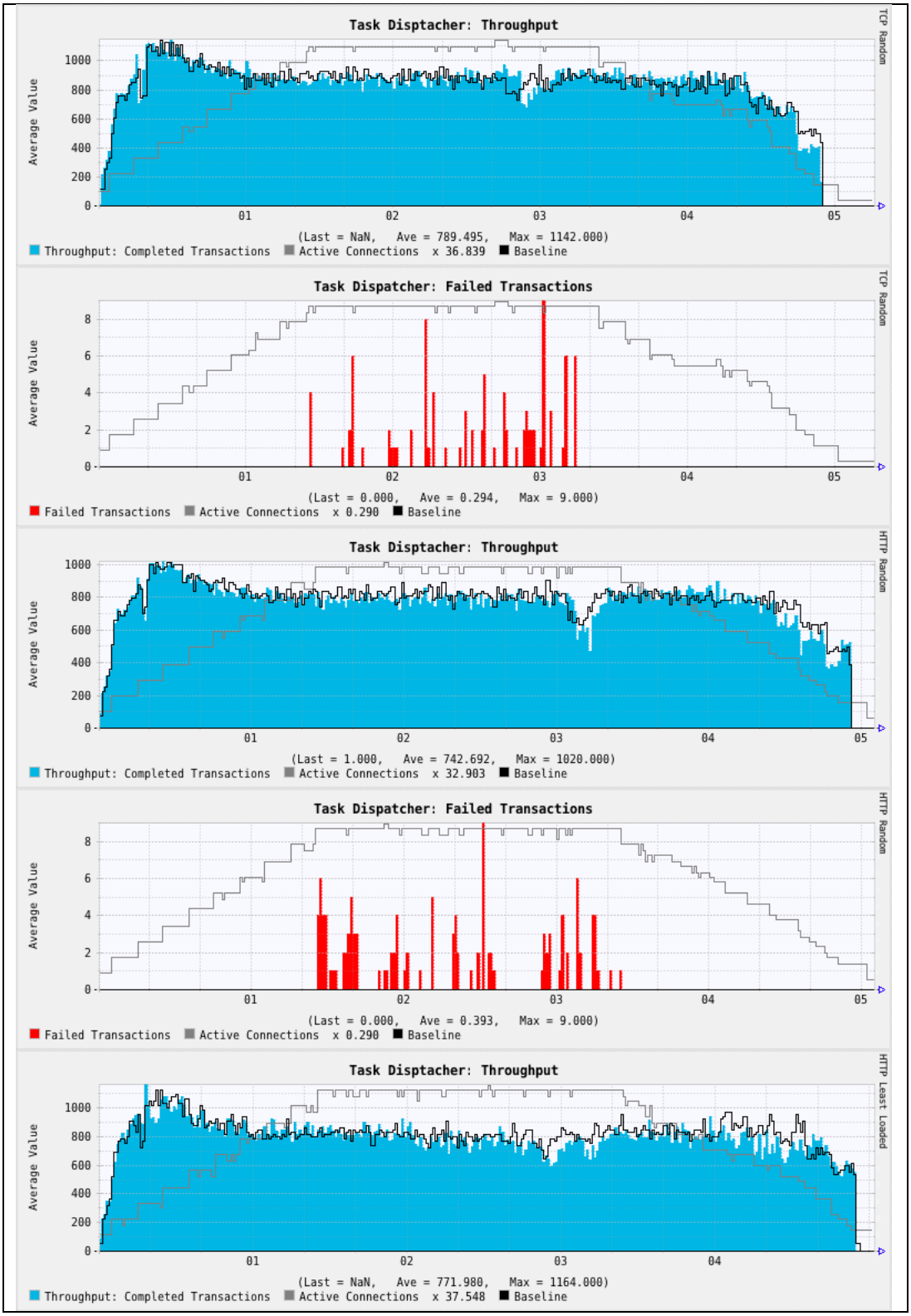
The JDBC connection threads experiment was designed to test the efficacy of task assignment policies in a cluster-state where the 3rd node (VCN3) would run out of database connections about halfway into the test, should it receive the same amount of requests as the other nodes.

Reaching the maximum number of JDBC database connections would result in the extended queuing of HTTP connections, because HTTP requests are waiting longer for database connections, resulting in denied connections when the HTTP connection queue size is exceeded.

The Least Connections policy, by evaluating the amount of connections to each server, was able to avoid denied connections as there is a strong relationship between the amount of HTTP connections that each server receives and the number of database connections that are used.


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
TCP Static Weighted Round-Robin
(Last = 490.000, Ave = 860.749, Max = 1145.000)
Throughput: Completed Transactions
Active Connections x 36.935
Baseline
Task Disptacher: Throughput
TCP Round-Robin
(Last = 0.000, Ave = 832.456, Max = 1137.000)
Throughput: Completed Transactions
Active Connections x 37.900
Baseline
Task Disptacher: Throughput
HTTP Least Connections
(Last = 175.000, Ave = 800.250, Max = 1088.000)
Throughput: Completed Transactions
Active Connections x 35.097
Baseline
Task Disptacher: Throughput
HTTP Static Weighted Round-Robin
(Last = 61.000, Ave = 788.845, Max = 1019.000)
Throughput: Completed Transactions
Active Connections x 33.967
Baseline
Task Disptacher: Throughput
HTTP Weighted Least Connections
(Last = 453.000, Ave = 788.807, Max = 1006.000)
Throughput: Completed Transactions
Active Connections x 33.533
Baseline


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
05
HTTP JdoStdV1
(Last = 0.000, Ave = 746.724, Max = 1004.000)
Throughput: Completed Transactions
Active Connections x 33.467
Baseline
Task Disptacher: Throughput
Average Value
HTTP F5 Dynamic Ratio
(Last = 1.000, Ave = 738.173, Max = 1026.000)
Throughput: Completed Transactions
Active Connections x 33.097
Baseline
Task Dispatcher: Failed Transactions
Average Value
1.0
0.8
0.6
0.4
0.2
0.0
HTTP F5 Dynamic Ratio
(Last = 0.000, Ave = 0.010, Max = 1.000)
Failed Transactions
Active Connections x 0.032
Baseline
Task Disptacher: Throughput
Average Value
HTTP Round-Robin
(Last = 1.000, Ave = 751.936, Max = 1043.000)
Throughput: Completed Transactions
Active Connections x 34.767
Baseline
Task Dispatcher: Failed Transactions
Average Value
HTTP Round-Robin
(Last = 0.000, Ave = 0.027, Max = 1.000)
Failed Transactions
Active Connections x 0.033
Baseline


Task Disptacher: Throughput
TCP Random
Average Value
1000
800
600
400
200
0
01
02
03
04
05
(Last = NaN, Ave = 789.495, Max = 1142.000)
Throughput: Completed Transactions
Active Connections x 36.839
Baseline
Task Dispatcher: Failed Transactions
TCP Random
Average Value
8
6
4
2
0
01
02
03
04
05
(Last = 0.000, Ave = 0.294, Max = 9.000)
Failed Transactions
Active Connections x 0.290
Baseline
Task Disptacher: Throughput
HTTP Random
Average Value
1000
800
600
400
200
0
01
02
03
04
05
(Last = 1.000, Ave = 742.692, Max = 1020.000)
Throughput: Completed Transactions
Active Connections x 32.903
Baseline
Task Dispatcher: Failed Transactions
HTTP Random
Average Value
8
6
4
2
0
01
02
03
04
05
(Last = 0.000, Ave = 0.393, Max = 9.000)
Failed Transactions
Active Connections x 0.290
Baseline
Task Disptacher: Throughput
HTTP Least Loaded
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = NaN, Ave = 771.980, Max = 1164.000)
Throughput: Completed Transactions
Active Connections x 37.548
Baseline

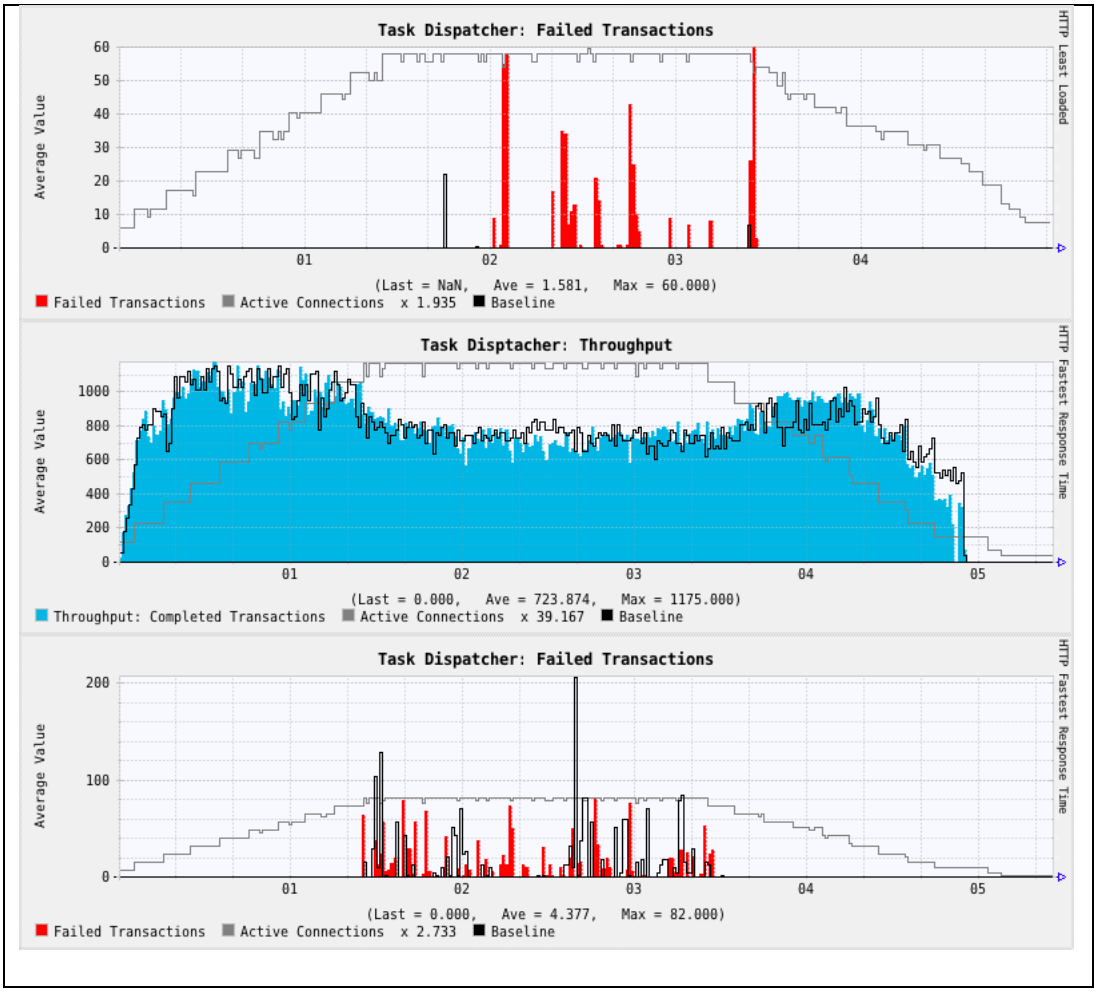
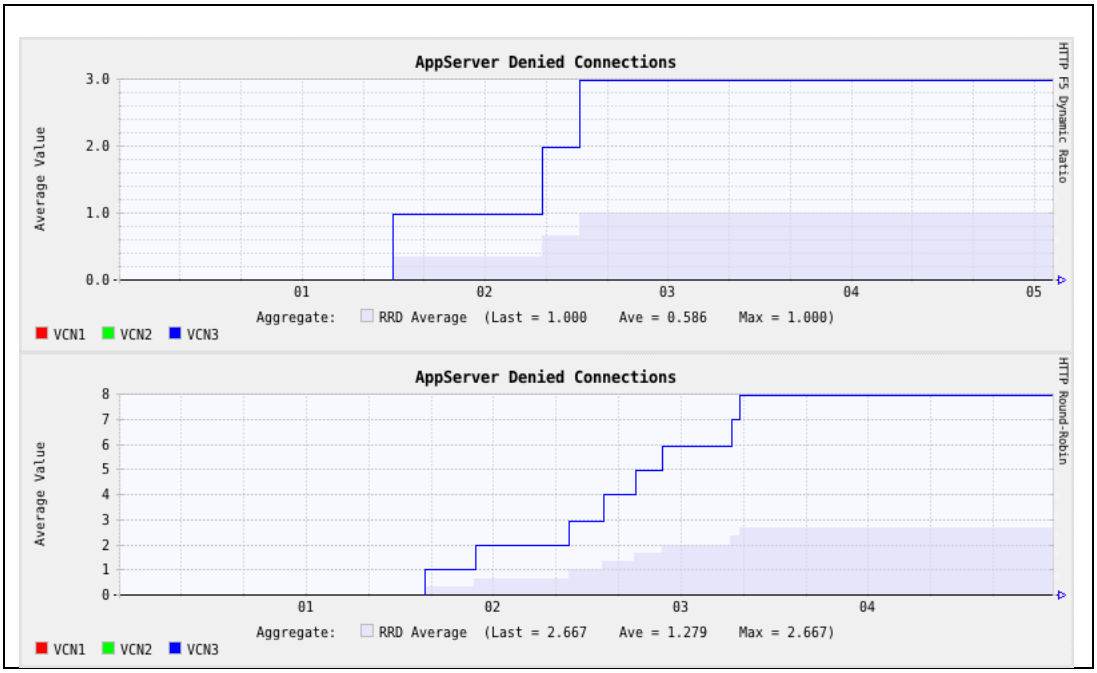


Figure 58 - Throughput (with Baseline and failure, where applicable) comparison of the policies in the JDBC connection threads experiment.



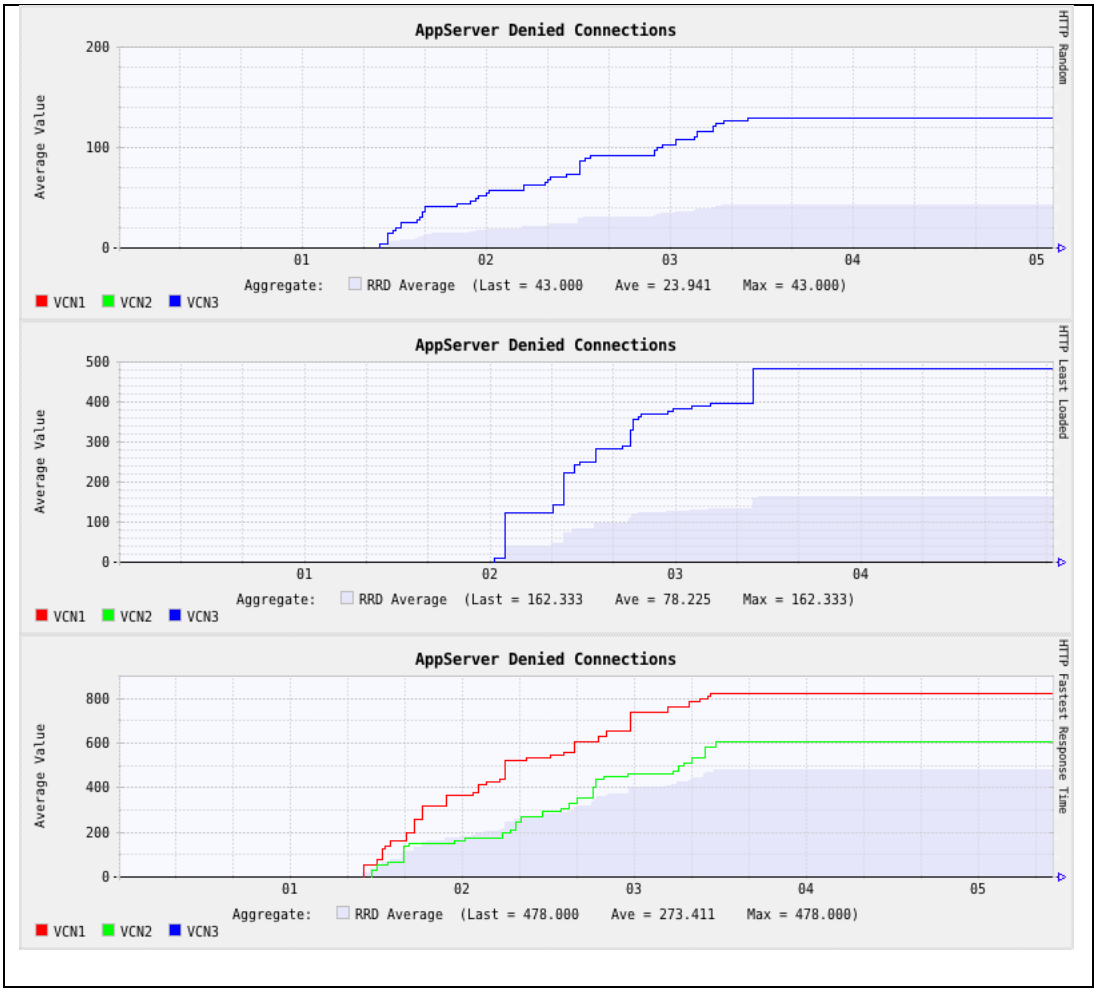
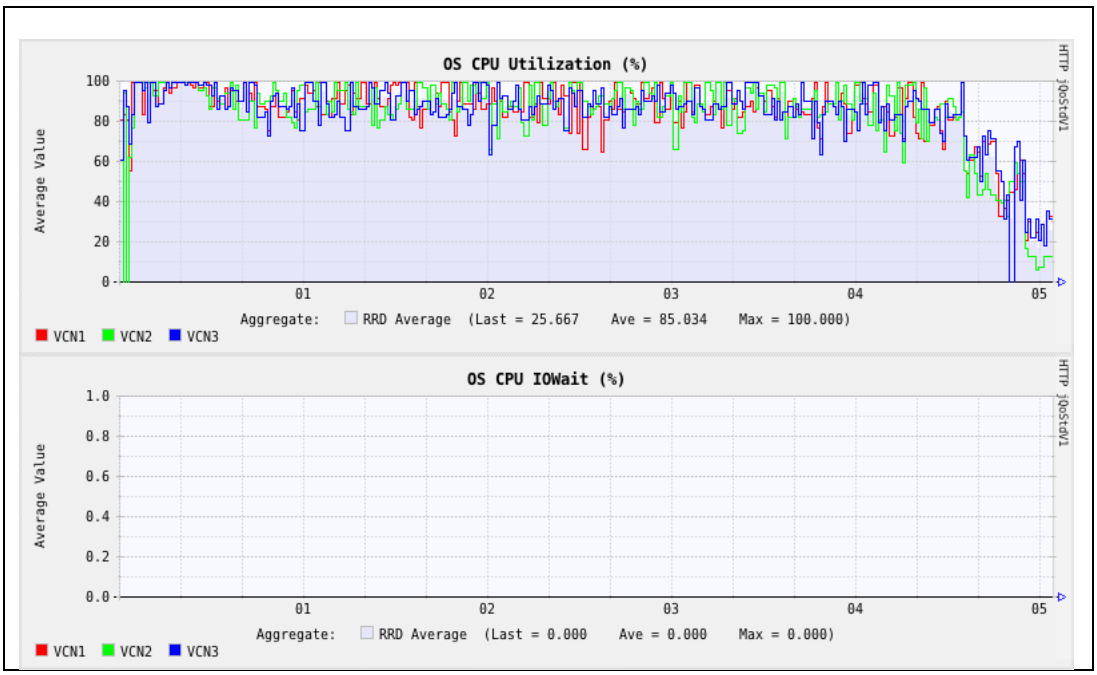
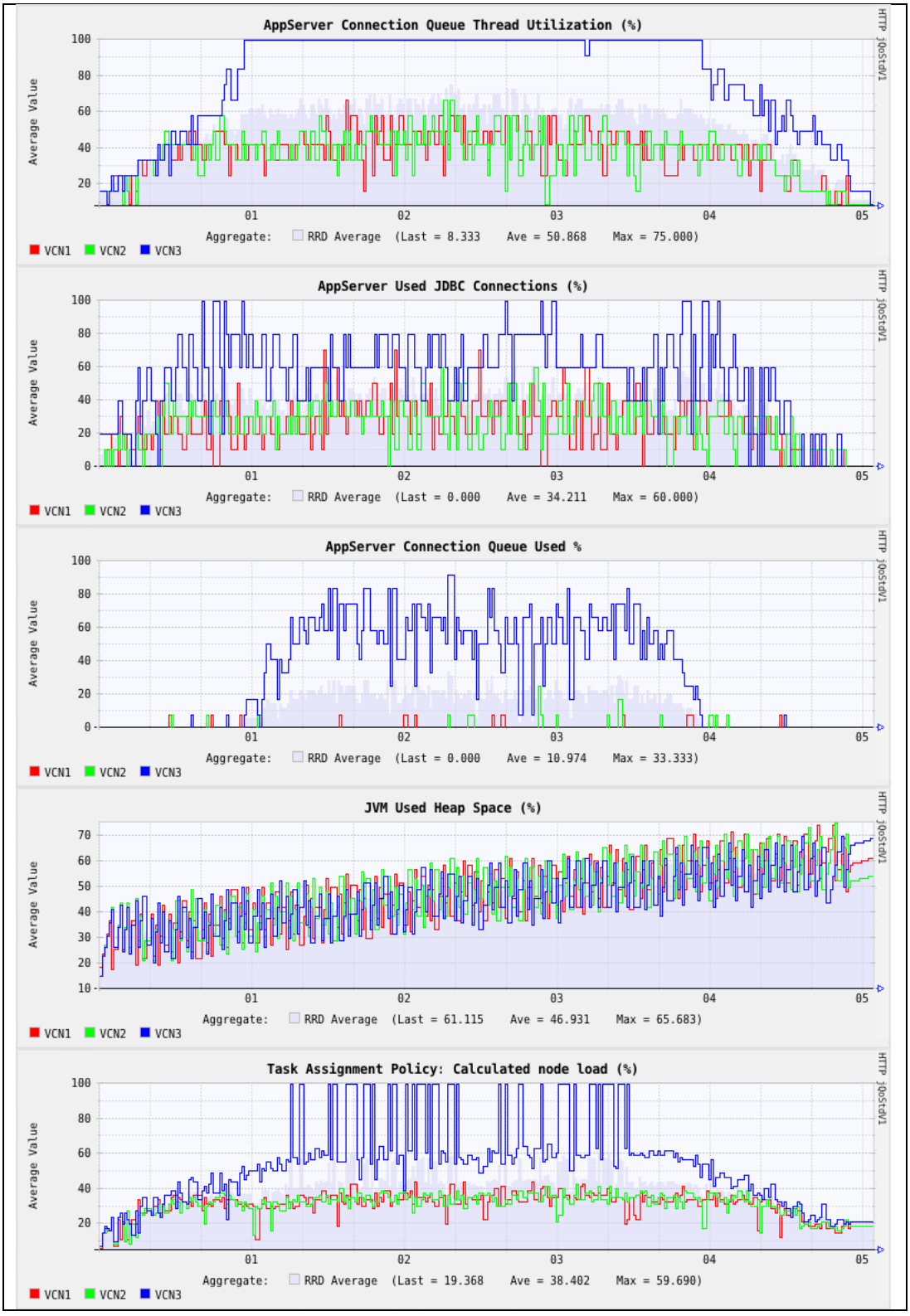


Figure 59 - Failure trace of the HTTP policies in the JDBC connection threads experiment.




AppServer Connection Queue Thread Utilization (%)
Average Value
100
80
60
40
20
01
02
03
04
05
Aggregate: RRD Average (Last = 8.333 Ave = 50.868 Max = 75.000)
VCN1 VCN2 VCN3
HTTP jOosStdV1
AppServer Used JDBC Connections (%)
Average Value
100
80
60
40
20
0
01
02
03
04
05
Aggregate: RRD Average (Last = 0.000 Ave = 34.211 Max = 60.000)
VCN1 VCN2 VCN3
HTTP jOosStdV1
AppServer Connection Queue Used %
Average Value
100
80
60
40
20
0
01
02
03
04
05
Aggregate: RRD Average (Last = 0.000 Ave = 10.974 Max = 33.333)
VCN1 VCN2 VCN3
HTTP jOosStdV1
JVM Used Heap Space (%)
Average Value
70
60
50
40
30
20
10
01
02
03
04
05
Aggregate: RRD Average (Last = 61.115 Ave = 46.931 Max = 65.683)
VCN1 VCN2 VCN3
HTTP jOosStdV1
Task Assignment Policy: Calculated node load (%)
Average Value
100
80
60
40
20
01
02
03
04
05
Aggregate: RRD Average (Last = 19.368 Ave = 38.402 Max = 59.690)
VCN1 VCN2 VCN3
HTTP jOosStdV1

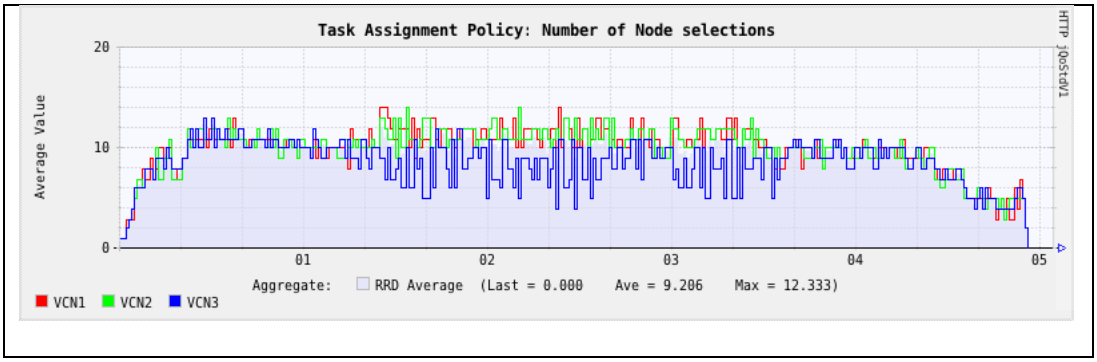


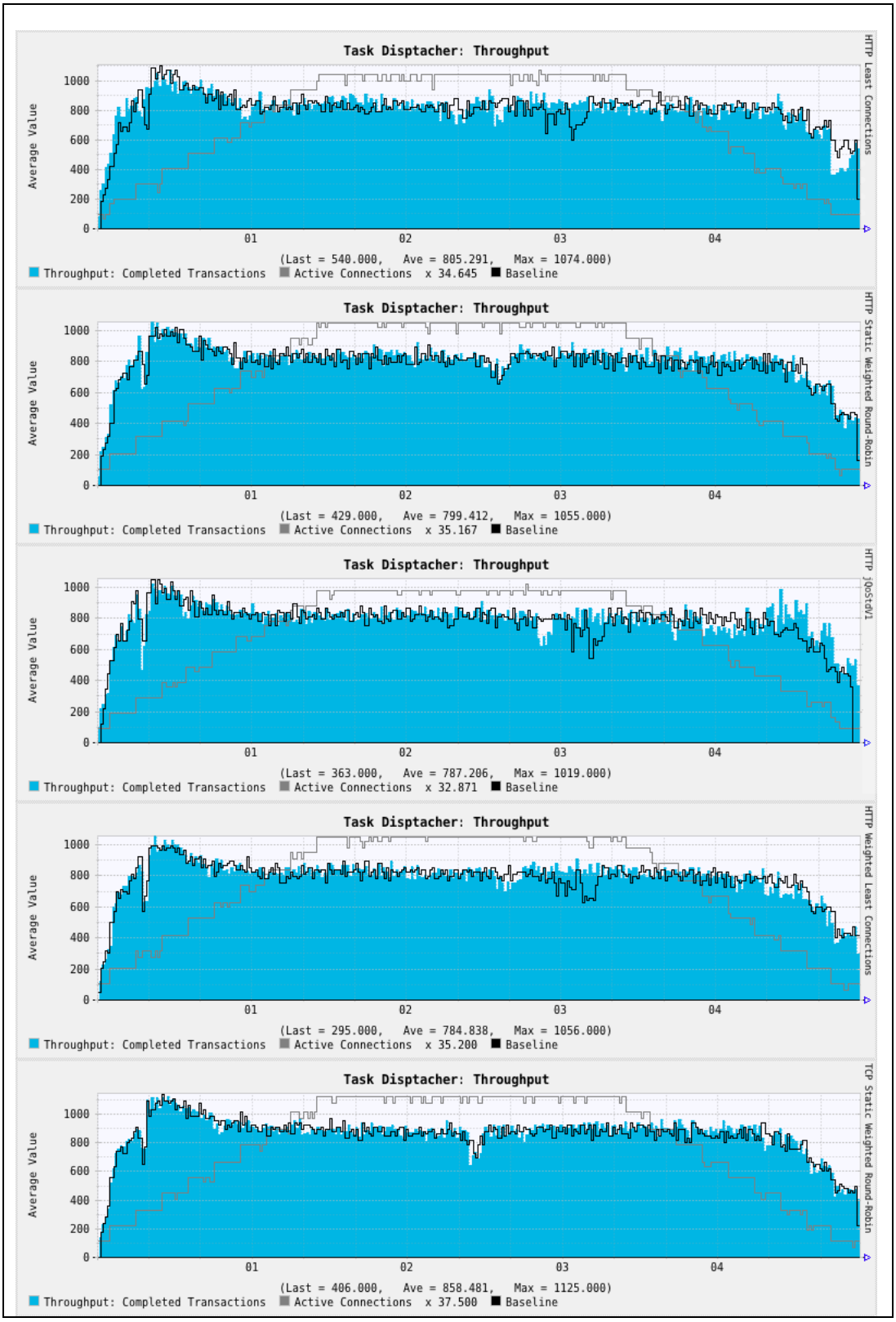
Figure 60 - Trace of the jQoStdV1 server load calculation and resulting server selections in the JDBC connection threads experiment.

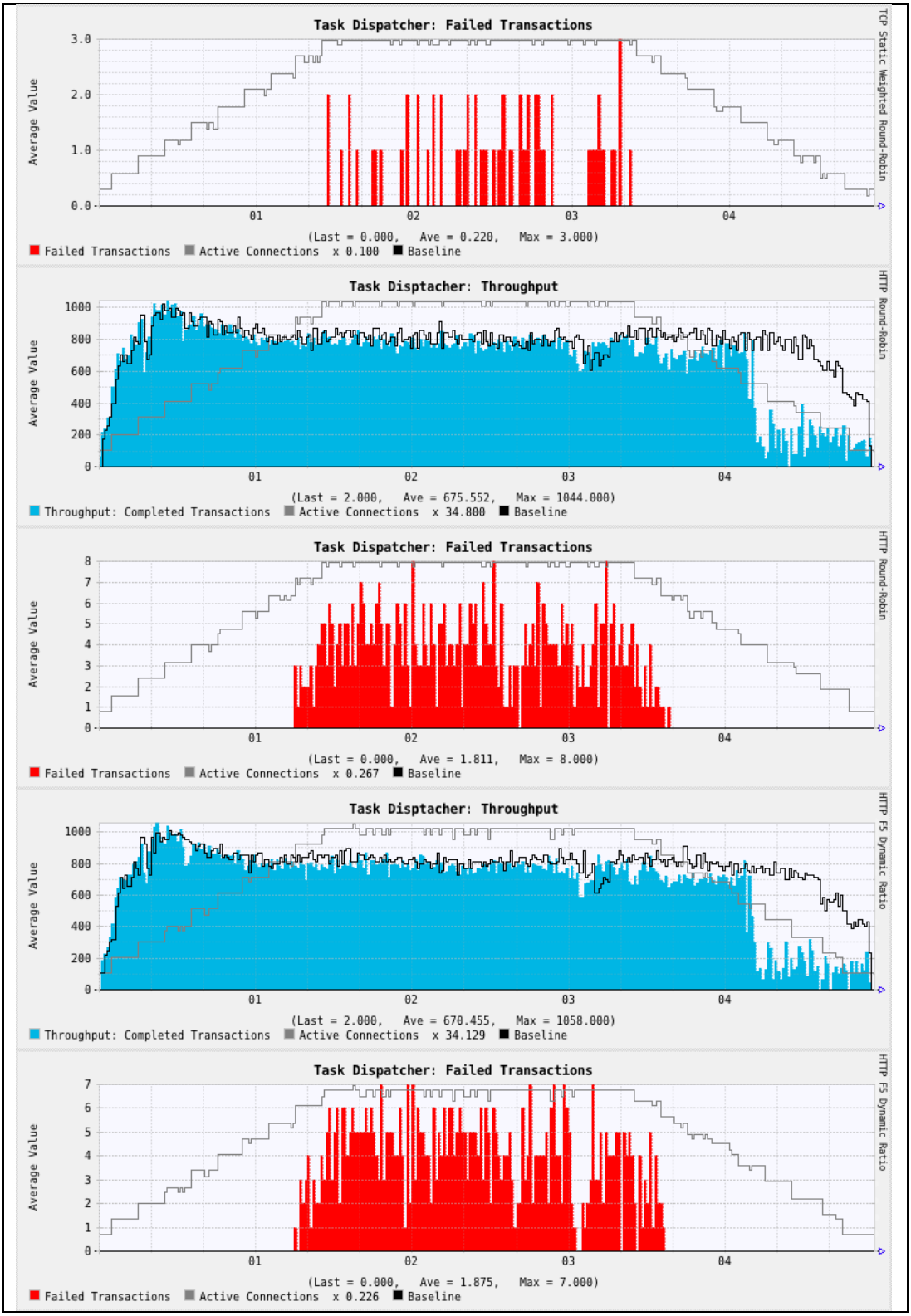
6.3.8 Evaluation of the All Scenarios Combined experiment

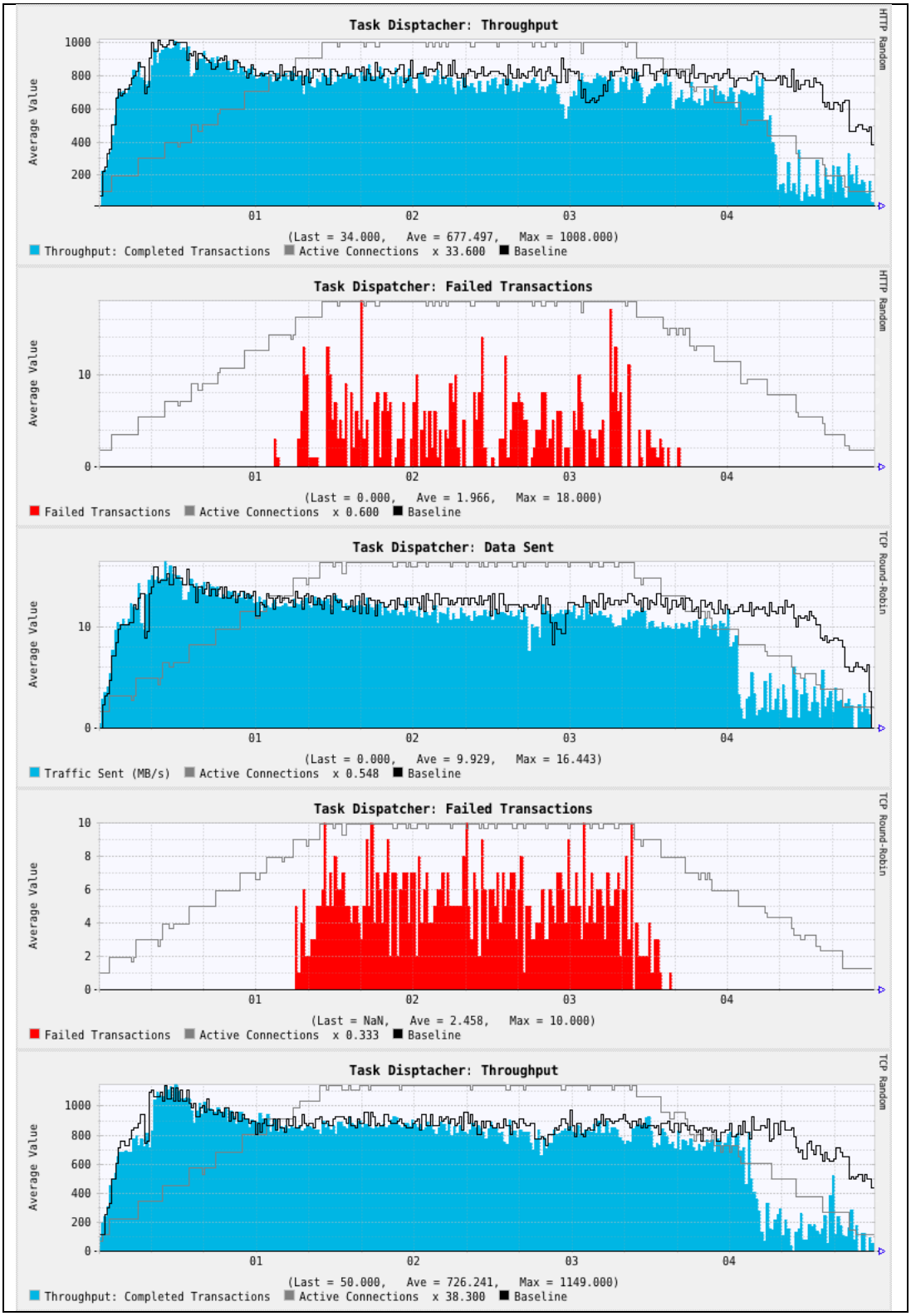
Table 14 presents the results of the task assignment policies in the All Scenarios Combined experiment, where VCN1 and 2 were configured as per the baseline configuration (512MB Heap Space, 12 HTTP connection queue worker threads and 10 database connections), and VCN3 configured with half the amount of capacity (256MB, 6, and 5).

The All Scenarios experiment was designed to test the efficacy of task assignment policies in a cluster-state where the 3rd node (VCN3) would run out of heap space, HTTP queue connection worker threads and database connections about halfway into the experiment, should it receive the same amount of requests as the other nodes.

This experiment resulted in request failures with all but the Least Connections, HTTP Static Weighted Round-Robin, jQoStdV1, and Weighted Least Connections policies.


Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 540.000, Ave = 805.291, Max = 1074.000)
Throughput: Completed Transactions
Active Connections x 34.645
Baseline
HTTP Least Connections
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 429.000, Ave = 799.412, Max = 1055.000)
Throughput: Completed Transactions
Active Connections x 35.167
Baseline
HTTP Static Weighted Round-Robin
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 363.000, Ave = 787.206, Max = 1019.000)
Throughput: Completed Transactions
Active Connections x 32.871
Baseline
HTTP jOoStdV1
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 295.000, Ave = 784.838, Max = 1056.000)
Throughput: Completed Transactions
Active Connections x 35.200
Baseline
HTTP Weighted Least Connections
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 406.000, Ave = 858.481, Max = 1125.000)
Throughput: Completed Transactions
Active Connections x 37.500
Baseline
TCP Static Weighted Round-Robin


Task Dispatcher: Failed Transactions
Average Value
3.0
2.0
1.0
0.0
01
02
03
04
TCP Static Weighted Round-Robin
(Last = 0.000, Ave = 0.220, Max = 3.000)
Failed Transactions
Active Connections x 0.100
Baseline
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
HTTP Round-Robin
(Last = 2.000, Ave = 675.552, Max = 1044.000)
Throughput: Completed Transactions
Active Connections x 34.800
Baseline
Task Dispatcher: Failed Transactions
Average Value
8
7
6
5
4
3
2
1
0
01
02
03
04
HTTP Round-Robin
(Last = 0.000, Ave = 1.811, Max = 8.000)
Failed Transactions
Active Connections x 0.267
Baseline
Task Disptacher: Throughput
Average Value
1000
800
600
400
200
0
01
02
03
04
HTTP F5 Dynamic Ratio
(Last = 2.000, Ave = 670.455, Max = 1058.000)
Throughput: Completed Transactions
Active Connections x 34.129
Baseline
Task Dispatcher: Failed Transactions
Average Value
7
6
5
4
3
2
1
0
01
02
03
04
HTTP F5 Dynamic Ratio
(Last = 0.000, Ave = 1.875, Max = 7.000)
Failed Transactions
Active Connections x 0.226
Baseline


Task Disptacher: Throughput
HTTP Random
Average Value
1000
800
600
400
200
01
02
03
04
(Last = 34.000, Ave = 677.497, Max = 1008.000)
Throughput: Completed Transactions
Active Connections x 33.600
Baseline
Task Dispatcher: Failed Transactions
HTTP Random
Average Value
10
0
01
02
03
04
(Last = 0.000, Ave = 1.966, Max = 18.000)
Failed Transactions
Active Connections x 0.600
Baseline
Task Dispatcher: Data Sent
TCP Round-Robin
Average Value
10
0
01
02
03
04
(Last = 0.000, Ave = 9.929, Max = 16.443)
Traffic Sent (MB/s)
Active Connections x 0.548
Baseline
Task Dispatcher: Failed Transactions
TCP Round-Robin
Average Value
10
8
6
4
2
0
01
02
03
04
(Last = NaN, Ave = 2.458, Max = 10.000)
Failed Transactions
Active Connections x 0.333
Baseline
Task Disptacher: Throughput
TCP Random
Average Value
1000
800
600
400
200
0
01
02
03
04
(Last = 50.000, Ave = 726.241, Max = 1149.000)
Throughput: Completed Transactions
Active Connections x 38.300
Baseline

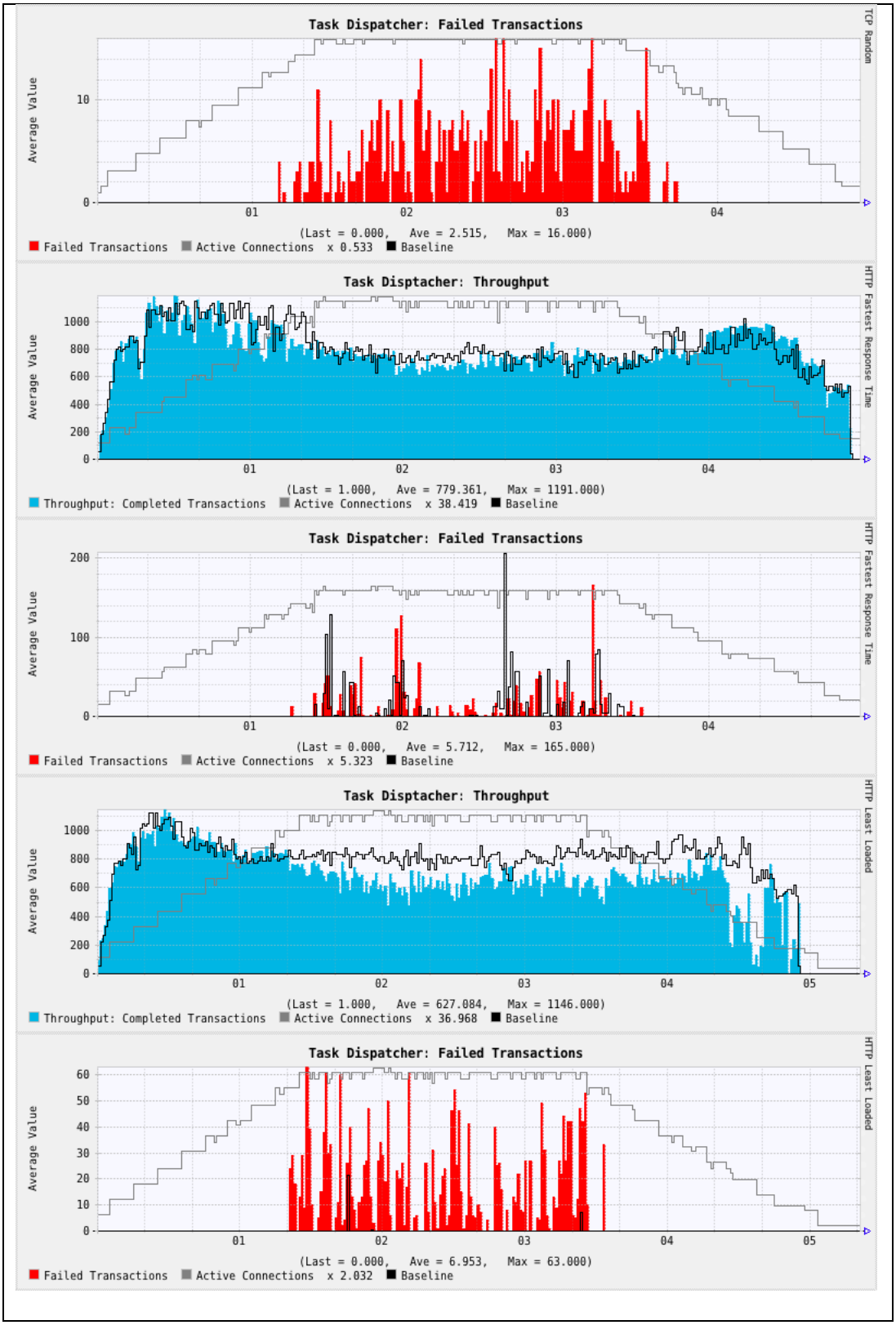
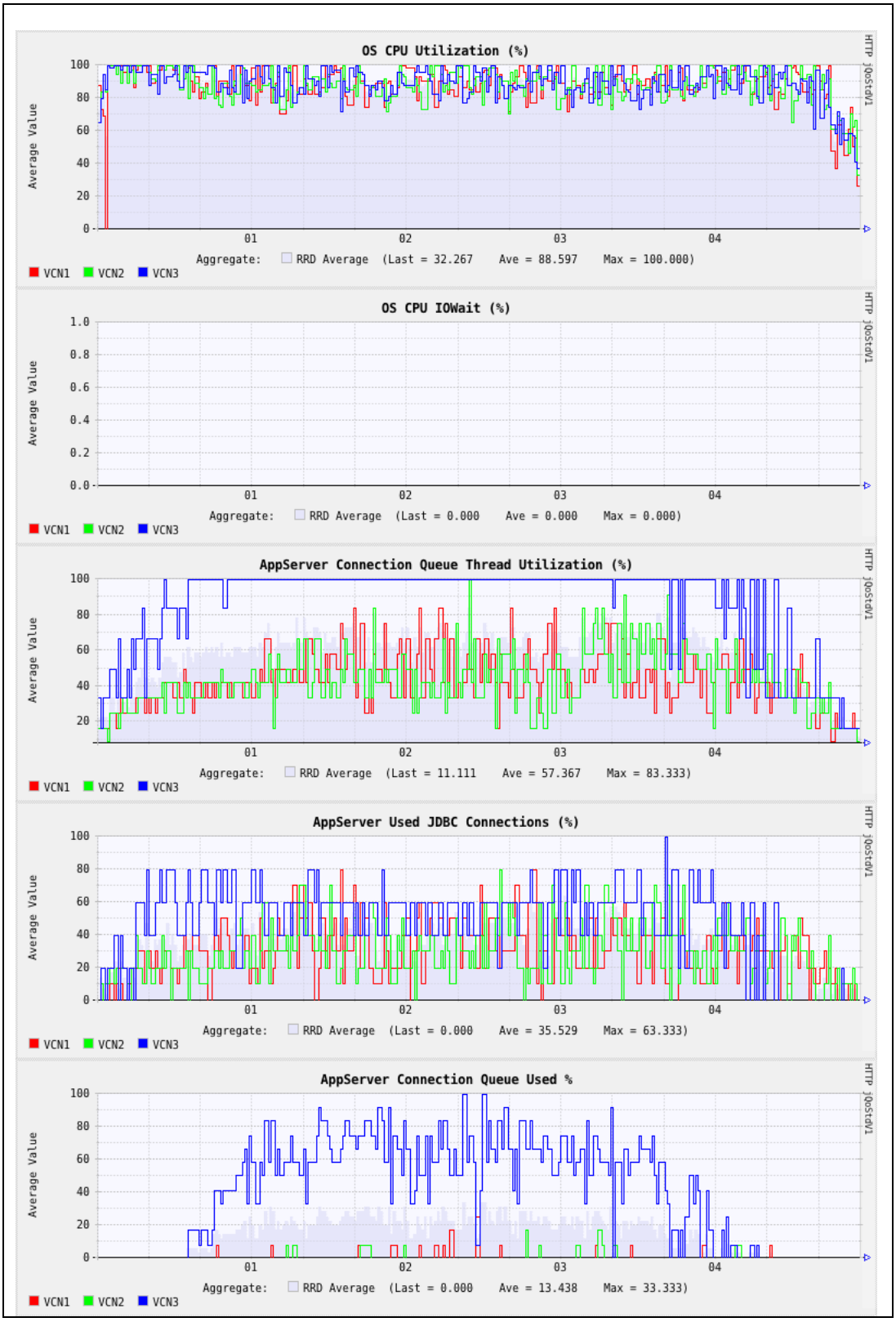


Figure 61 - Throughput (with Baseline and failure, where applicable) comparison of the policies in the All Scenarios Combined experiment.

Figure 62 - Failure trace of the HTTP policies in the All Scenarios Combined experiment.


OS CPU Utilization (%)
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 32.267 Ave = 88.597 Max = 100.000)
VCN1 VCN2 VCN3
HTTP jQoStdV1
OS CPU IOWait (%)
Average Value
1.0
0.8
0.6
0.4
0.2
0.0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 0.000 Max = 0.000)
VCN1 VCN2 VCN3
HTTP jQoStdV1
AppServer Connection Queue Thread Utilization (%)
Average Value
100
80
60
40
20
01
02
03
04
Aggregate: RRD Average (Last = 11.111 Ave = 57.367 Max = 83.333)
VCN1 VCN2 VCN3
HTTP jQoStdV1
AppServer Used JDBC Connections (%)
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 35.529 Max = 63.333)
VCN1 VCN2 VCN3
HTTP jQoStdV1
AppServer Connection Queue Used %
Average Value
100
80
60
40
20
0
01
02
03
04
Aggregate: RRD Average (Last = 0.000 Ave = 13.438 Max = 33.333)
VCN1 VCN2 VCN3
HTTP jQoStdV1

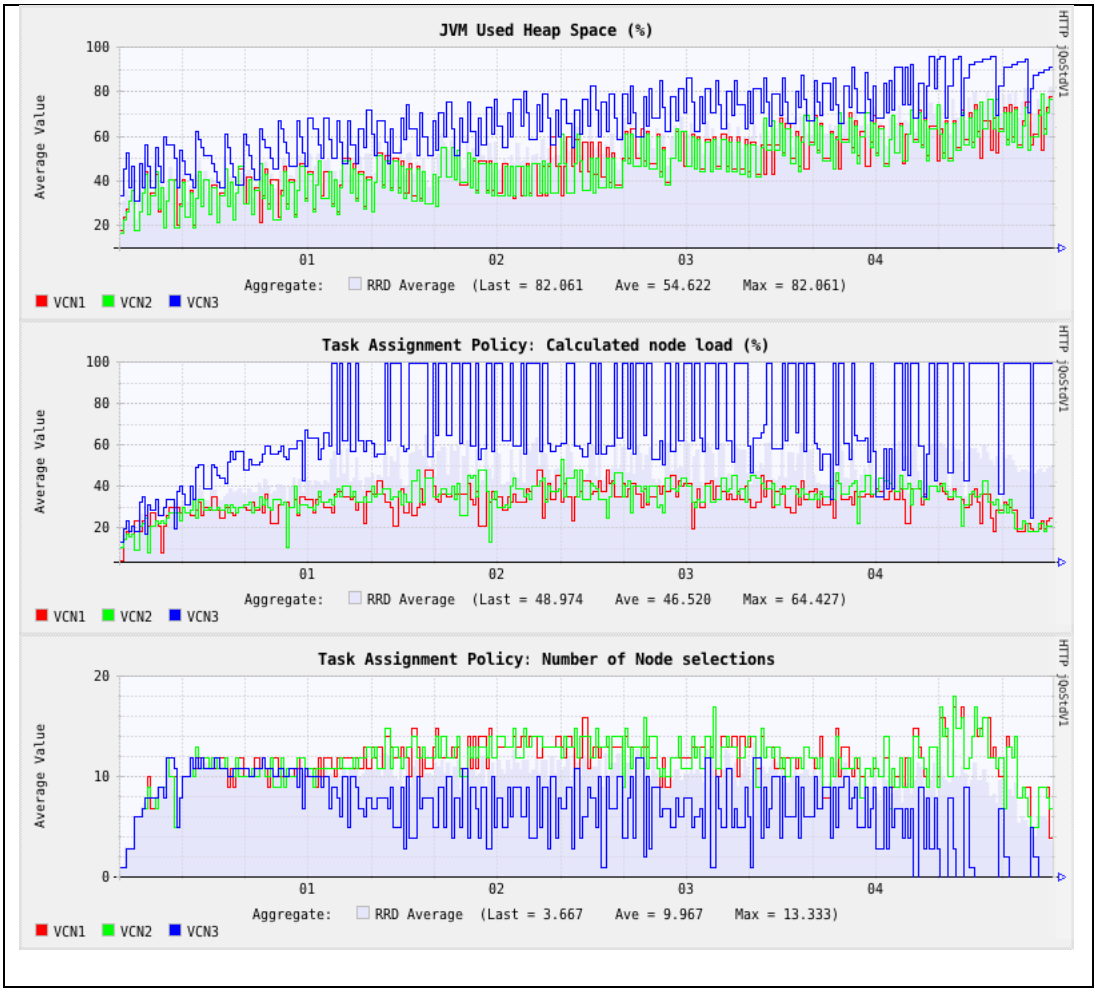


Figure 63 - Trace of the jQoStdV1 server load calculation and resulting server selections in the All Scenarios Combined experiment.

6.4 Summary

Overall policy rank and summary results					
Overall Rank	Task Assignment Policy	Weight Ratio	Samples	Throughput	Failed
Control	*HTTP Static Weighted Round-Robin*	*2:2:1*	*234411*	*791.93*	*0.00*
1	HTTP Least Connections	1:1:1	232997	787.15	0.00
2	HTTP jQoStdV1Policy	1:1:1	231166	778.52	0.00
3	HTTP Weighted Least Connections	2:2:1	226071	763.75	0.00
4	TCP Static Weighted Round-Robin	2:2:1	252734	856.72	13.00
5	HTTP F5 Dynamic Ratio	1:1:1	213122	714.57	162.60
6	HTTP Round-Robin	1:1:1	212077	712.90	170.60
7	HTTP Random	1:1:1	215116	720.98	212.40
8	TCP Round-Robin	1:1:1	227662	768.72	260.80
9	TCP Random	1:1:1	228992	757.96	289.40
10	HTTP Least Loaded	1:1:1	221647	720.28	784.00
11	HTTP Fastest Response Time	1:1:1	236525	763.06	1639.40
	Min		*212077*	*712.90*	*0.00*
	Max		*252734*	*856.72*	*1639.40*
	Average		*227710*	*761.38*	*294.35*
	Standard Deviation		*11464*	*41.47*	*478.18*

Table 15 - The overall ranking of the policies based upon the averages of their throughput and failed transactions in all experiments.

This study has shown that the Static Weighted Round-Robin, Least Connections and jQoStdV1 policies were able to provide stable results in all of the experiments, provided that a moderate amount of capacity planning was done.

Dynamic policies, such as the Least Connections and jQoStdV1 policies, are predicted to perform extremely well in varying (but sensible) capacity and utilization situations, with all of the other policies being very likely to compromise the reliability of Java EE deployments in situations where capacity planning is less accurate.

The proposed jQoStdV1 policy, in particular, has performed much better than the competing F5 Dynamic Ratio policy, and provides the jQoStdV1 task dispatching solution with an advantage over both the F5 LTM, and Apache mod_proxy solutions in situations where static WRR policies can't be used with a great degree of confidence.

The use of the jQoStdV1 policy would also provide the ability to automatically prevent the over-utilization of core Java EE web application resources, providing jQoStdV1 with a major advantage over both the static WRR, Least Connections, and Weighted Least Connections, as well as all of the other policies.

The task dispatcher implementation, and the experimental environment also provided very consistent and reliable results, and in consideration of all experiments, resulted in an average slowdown of 1.008, and average latency of 41.75 milliseconds per request.

Chapter 7. CONCLUSIONS

7.1 Lessons Learned

The evaluation of the task assignment policies proved to be the most challenging activity in this dissertation.

The initial evaluation of the policies was conducted without any automation, amounting to ~15 hours of testing per cycle, provided that nothing went wrong in any of the experiments. This, unfortunately, happened quite often due to the unreliability of application servers under such capacity constraints.

Performing capacity planning for one policy in an environment can be a challenge, but accomplishing this for 12 policies in 5 different environments that create situations where the policies may over-utilise the capacity of core resources (and then often crash application servers and JVMs), is really challenging.

Changes to the source code or environment configurations also required that all of the experiments are repeated, and with a testing cycle of 15 hours, this activity soon became very unproductive.

A decision was made to investigate the possibility of automating the experiments. This required a significant investment of time, in an already compressed schedule, as modifications had to be made to the task dispatcher in order to support the automated collection of statistics, time series

graphs and other data in a consistent and reliable form. This investment, however, proved to be vey successful and is a valuable lesson learnt.

Another interesting problem presented itself in the form of system heat. Initial testing was done with a set of powerful servers in a testing lab. As part of this dissertation's assessment process, however, it is beneficial to provide assessors with a fully functioning copy of the software and the evaluation environment. Hence, a decision was made to perform all experiments with the required software components installed on the same computer.

The computational demands of the experiments, however, soon highlighted the impact of system (CPU) heat in the outcome of experiments. The average system temperature, after a couple of experiments, reached 98°C, dangerously close to the 100 degrees threshold where Intel CPUs would send shutdown signals to prevent damage. Additional air-cooling, in the form of external fans, stabilised the system temperature to an average 85°C for all experiments, thereby providing consistent results.

7.2 Future Activity

One of the goals of the jQoStd project is provide an Open Source load balancing solution to the Java EE community that can compete with commercial solutions, such as the F5 LTM.

The prototype version of jQoStd currently supports GlassFish Server with some preliminary support for jBoss versions 5 and 7. A lot of work still

needs to be done to provide a solution that is compatible with a greater variety of application servers, where we hope that this goal can be realized with the participation of the Java EE community.

Providing support for the collection of application server metrics in all major application servers is thus the next logical step in the jQoStd project.

7.3 Prospects for Further Work

The success of the jQoStdV1 policy in this study supports further research into enhancements and similar platform-dependant policies for use with other environments as well.

The dynamic detection and prevention of over-utilization states in core system resources, in particular, has proven to be very useful and may motivate further research as well.

JQoStd has been designed to support the development of dispatching policies in a variety of contexts, and can easily be extended to include dispatching support for many other environments and situations, thereby providing a stable and modular platform for the research and development of new policies.

jQoStd's Application Performance Monitoring features also provide a competitive and free alternative to commercial products that perform similar functions. Further research and development of these features, such as automatic thresholds-based reporting, alerts, and fault-tracing features,

could be a valuable addition to jQoStd that would challenge other established APM solutions.

Another interesting academic research prospect may be research and development of automated capacity planning, and validation software for Java EE environments. The jQoStd project already provides many of the core components that would be required in such research, and may become a very promising commercial venture.

REFRENCES CITED

Aberdeen Group (2008) *Customers are won or lost in one second* [Online]. Available from: http://www.gomez.com/wp-content/downloads/Aberdeen_WebApps.pdf (Accessed: 28 February 2012).

Andreolini, M., Casalicchio, E., Colajanni, M. & Mambelli, M. (2001) 'A Cluster-Based Web System Providing Differentiated and Guaranteed Services', *Cluster Computing*, 7 (1), January, pp.7-19.

Andreolini, M., Casolari, S. & Colajanni, M. (2006) 'A Distributed Architecture for Gracefully Degradable Web-Based Services', *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on* , pp.235-238.

Andreolini, M., Colajanni, M. & Morselli, R. (2002) 'Performance study of dispatching algorithms in multi-tier web architectures', *ACM SIGMETRICS Performance Evaluation Review*, 30 (2), September, pp.10-20.

Andreolini, M., Colajanni, M. & Nuccio, M. (2003) *Scalability of content-aware server switches for cluster-based Web information systems* [Online]. Available from: http://weblab.ing.unimo.it/papers/www2003.pdf (Accessed: 15 January 2012).

Apache (2012a) *Apache HTTP Server Project* [Online]. Available from: http://httpd.apache.org/ (Accessed: 15 January 2012).

Apache (2012b) *Apache: Module mod_proxy* [Online]. Available from: http://httpd.apache.org/docs/2.0/mod/mod_proxy.html (Accessed: 15 January 2012).

Apache (2012c) *Apache: Module mod_rewrite* [Online]. Available from: http://httpd.apache.org/docs/current/mod/mod_rewrite.html (Accessed: 15 January 2012).

Apache Geronimo (2010) *DayTrader*, 12 March [Online]. Available from: https://cwiki.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html.

Apache jMeter (2012) *Apache jMeter*, 15 March [Online]. Available from: http://jmeter.apache.org/.

Aron, M., Druschel, P. & Zwaenepoel, W. (2000) 'Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers', *Measurement and modeling of computer systems; ACM SIGMETRICS '2000*, Santa Clara, pp.90-101.

Bourke, P. (1998) *AutoRegression Analysis (AP)* [Online]. Available from: http://paulbourke.net/miscellaneous/ar/ (Accessed: 16 January 2012).

Cardellini, V., Casalicchio, E., Colajanni, M. & Yu, P.S. (2002) 'The state of the art in locally distributed web-server systems', *ACM Computing Surveys (CSUR)*, 34 (2), June, pp.263-311.

Casalicchio, E. & Colajanni, M. (2001) 'A client-aware dispatching algorithm for Web clusters providing multiple services', *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, pp.535-544.

Casalicchio, E. & Tucci, S. (2001) 'Static and dynamic scheduling algorithms for scalable Web server farm', *Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on* , pp.369-376.

Casalicchio, E., Cardellini, V. & Colajanni, M. (2002) 'Content-Aware Dispatching Algorithms for Cluster-Based Web Servers', *Cluster Computing*, 5 (1), pp.65-74.

Challenger, J.R., Dantzig, P., Iyengar, A., Squillante, M.S., Zhang, L. & J., T. (2004) 'Efficiently serving dynamic data at highly accessed web sites', *Networking, IEEE/ACM Transactions on* , 12 (2), April, pp.233-246.

Cherkasova, L. & Phaal, P. (2002) 'Session-based admission control: a mechanism for peak load management of commercial Web sites', *Computers, IEEE Transactions on*, 51 (6), June, pp.669-685.

Dahlin, M. (2000) 'Interpreting stale load information', *Parallel and Distributed Systems, IEEE Transactions on*, 11 (10), October, pp.1033-1047.

Daily Mail (2012) *Twitter Outage: 16k Happy New Year tweets per second leads to meltdown.html* [Online]. Available from: http://www.dailymail.co.uk/sciencetech/article-2080814/Twitter-outage-16k-Happy-New-Year-tweets-SECOND-lead-meltdown.html (Accessed: 5 January 2012).

Denning, P.J. & Buzen, J.P. (1978) 'The Operational Analysis of Queueing Network Models', *ACM Computing Surveys (CSUR)*, 10 (3), p.225–261.

Dinda, P.A. & O'Hallaron, D.R. (2000) 'Host load prediction using linear models', *Cluster Computing*, 3 (4), pp.265-280.

F5 Networks (2010) *Overview of Dynamic Ratio load balancing* [Online]. Available from: http://support.f5.com/kb/en-us/solutions/public/9000/100/sol9125.html (Accessed: 26 March 2012).

F5 Networks (2011) *BIG-IP Local Traffic Manager: F5 Datasheet* [Online]. Available from: www.f5.com/pdf/products/big-ip-local-traffic-manager-ds.pdf (Accessed: 22 February 2012).

F5 Networks (2012) *BIG-IP Product Family* [Online]. Available from: http://www.f5.com/products/big-ip/ (Accessed: 6 February 2012).

Gartner (2009) *Load Balancers Are Dead: Time to Focus on Application Delivery* [Online]. Available from: http://swathidharshananaidu.posterous.com/load-balancers-are-dead-time-to-focus-on-appl-0 (Accessed: 21 February 2012).

Gartner (2010) *Magic Quadrant for Application Delivery Controllers* [Online]. Available from: http://www.gartner.com/technology/media-products/reprints/citrix/article13/article13.html (Accessed: 29 February 2012).

GlassFish (2012) *GlassFish Server* [Online]. Available from: http://glassfish.java.net/ (Accessed: 13 February 2012).

Grundy, J. & Liu, A. (2000) *Directions in Engineering Non-Functional Requirement Compliant Middleware Applications* [Online]. Available from: http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.16.4129&rep=rep1&type=pdf (Accessed: 1 February 2012).

Harchol-Balter, M., Crovella, M.E. & Murta, C.D. (1999) 'On Choosing a Task Assignment Policy for a Distributed Server System', *Journal of Parallel and Distributed Computing*, 59 (2), pp.204-228.

Harcol-Balter, M. (2002) 'Task Assignment with Unknown Duration', *Journal of the ACM (JACM)*, 49 (2), March, pp.260-288.

httperf (2011) *The httperf HTTP load generator* [Online]. Available from: http://code.google.com/p/httperf/ (Accessed: 17 January 2012).

Hunt, G.D.H., Goldszmidt, G.S., King, R.P. & Mukherjee, R. (1998) 'Network Dispatcher: A connection router for scalable Internet services. ', *Computer Networks*, 30 (1-7), pp.347-357.

Hyperic (2012) *Sigar API* [Online]. Available from: http://www.hyperic.com/products/sigar (Accessed: 01 February 2012).

Java Community Process (2009) *JSR-000316 Java Platform, Enterprise Edition 6 Specification 6.0 Final Release* [Online]. Available from: http://download.oracle.com/otndocs/jcp/javaee-6.0-fr-eval-oth-JSpec/ (Accessed: 12 January 2012).

Linden, G. (2006) *Make Data Useful* [Online]. Available from: https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt?attredirects=0 (Accessed: 28 February 2012).

Netty (2012) *Netty API* [Online]. Available from: http://netty.io (Accessed: 01 March 2012).

Oracle (2012) *Java Management Extensions JMX Technology* [Online]. Available from: http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html (Accessed: 01 February 2012).

OW2 Consortium (2009) *RUBiS* [Online]. Available from: http://rubis.ow2.org/ (Accessed: 16 January 2012).

Quest Software (2012) *Foglight* [Online]. Available from: http://www.quest.com/foglight/ (Accessed: 28 July 2012).

Reddy, A. (2000) *Java Coding Style Guidelines* [Online]. Available from: http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf (Accessed: 21 March 2012).

RRD4j (2012) *RRD4j* [Online]. Available from: http://code.google.com/p/rrd4j/ (Accessed: 01 March 2012).

Schroeder, B. & Harchol-Balter, M. (2006) 'Web servers under overload: How scheduling can help', *ACM Transactions on Internet Technology (TOIT)*, 6 (1), February, pp.20-52.

Schroeder, T., Goddard, S. & Ramamurthy, B. (2000) 'Scalable Web server clustering technologies', *Network, IEEE*, 14 (3), May\July, pp.38-45, Available: 10.1109/65.844499.

Shan, Z., Lin, C., Marinescu, D.C. & Yang, Y. (2002) 'Modeling and performance analysis of QoS-aware load balancing of Web-server clusters', *Computer Networks*, 40 (2), October, pp.235-256.

Sharifian, S., Motamedi, S.A. & Akbari, M.K. (2009) 'An approximation-based load-balancing algorithm with admission control for cluster web servers with dynamic workloads', *THE JOURNAL OF SUPERCOMPUTING*, 53 (3), July, pp.440-463.

Shurman, E. & Brutlag, J. (2009) *The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search* [Online]. Available from: http://velocityconf.com/velocity2009/public/schedule/detail/8523 (Accessed: 11 January 2012).

Sigar (2012) *SIGAR - System Information Gatherer And Reporter* [Online]. Available from: http://support.hyperic.com/display/SIGAR/Home (Accessed: 27 February 2012).

Twitter (2012) *Twitter.com Home page* [Online]. Available from: http://www.twitter.com (Accessed: 01 February 2012).

Webopedia (2010) *The 7 Layers of the OSI Model* [Online]. Available from: http://www.webopedia.com/quick_ref/OSI_Layers.asp (Accessed: 15 January 2012).

Webstone (1998) *WebStone 2.x Benchmark Tool* [Online]. Available from: http://www.mindcraft.com/webstone/ws201-descr.html (Accessed: 15 January 2012).

Xiong, Z. & Yan, P. (2005) 'A solution for supporting QoS in Web server cluster', *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on* , 2 (23), September, pp.834-839.

Yand, C.S. & Lou, M.Y. (2000) 'A content placement and management system for distributed Web-server systems.', *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taipei, pp.691-698.

APPENDICES

Appendix A. CONTENT-AWARE POLICIES

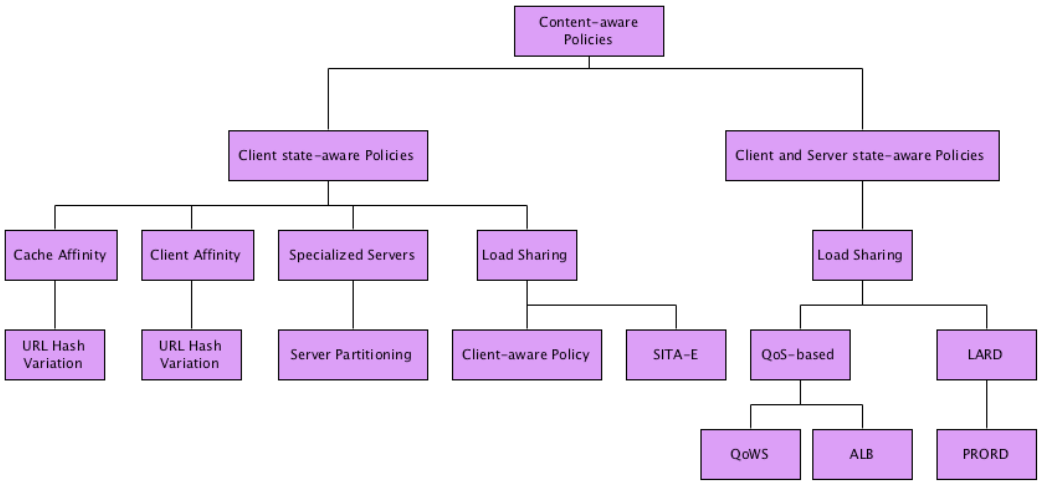


Figure 64 - Content-aware task assignment policies.

A.1 Client state-aware policies

A.1.1 Cache Affinity, Client Affinity and Server Partitioning policies

Cache Affinity policies are a variation of the Client Partitioning policy, where more detailed client state information, such as HTTP request headers, are evaluated in order to assign requests to static server partitions, with the goal of maximising the (local) cache hit rates of requests on those servers.

The **URL Hash** policy is similar in function to the **Client Affinity** policy, with the difference being that a hash of the request URL (or a substring of it) is used, instead of the client's IP address, in order to determine which server should receive the request.

Another variation of the URL Hash policy attempts to perform **Service Partitioning** where specialized servers are used for specific content request

types. A request for a video file would, for example, always be sent to a specialized video-streaming server.

Research by Yand and Lou (2000) and Cardellini et al. (2002) described the URL Hash policies as being effective for static content requests. Requests for dynamic content, however, may result in server saturation as these policies do not consider server load state information in their assignments, and would thus exhibit the same limitations that state-blind (static) policies have.

A.1.2 Load Sharing policies

The **Size Interval Task Assignment with Equal load** (SITA-E) policy dynamically partitions servers based on (static) file size distributions, and are primarily aimed at the provision of static content. As described in Harchol-Balter, Crovella and Murta (1999), each server in the cluster would have a size interval and would only receive tasks that fall within that size interval, thereby attempting to share load within the cluster.

The results described in Harchol-Balter, Crovella and Murta (1999) introduced some concerns about the performance of SITA-E. Static size-interval distributions may result in the saturation of some server(s) that fall into the “small” intervals, in that static content, such as an HTML page, is often composed from several smaller files, such as CSS, JavaScript’s and image files. In the absence of an effective caching mechanism this may have a pronounced impact on the overall performance of the cluster when size variability is introduced, typical of heavy-tailed distributions.

The **Client-Aware Policy** (CAP), proposed by Casalicchio and Colajanni (2001), attempts to address some of the SITA-E problems by classifying requests into four service time classes, on “the basis of their [estimated] impact on server resources”. Requests (URLs) are classified (manually) as static and light dynamic Web services, CPU-bound services, disk-bound services, and CPU-and-disk-bound services. The policy maintains a circular routing table of class to server mappings, where classified requests are routed according to the queue lengths of active class assignments, such that dynamic load sharing could be realised.

In Casalicchio and Colajanni (2001), it was demonstrated that the CAP policy achieves predictable performance in heterogeneous (server capacity) clusters, provided that the server utilization estimates for the respective URLs are accurate. Further research by Casalicchio, Cardellini and Colajanni (2002), showed that the CAP policy outperformed the Service Partitioning and Locality-Aware Request Distribution (LARD) policies with dynamic content. The LARD policy (discussed in the next section), however, performed the best when static content was considered (due to local server caching).

The CAP policy, in its basic form, does not consider any server state information, and may, as a result, cause server saturation and would be unable to detect server faults efficiently. This policy is the firm favourite in hardware task dispatchers, where variations of the policy have been implemented to include the consideration of basic server state information (F5 Networks, 2012), such as ICMP (ping) server health checks.

A.2 Client and Server state-aware policies

A.2.1 Load Sharing policies

The **Locality-Aware Request Distribution** (LARD) policy, as discussed in Aron, Druschel and Zwaenepoel (2000), considers both cache locality and load sharing in the assignment of requests with the following rules:

- New requests (URLs) are sent to a server, forming part of dynamic a pool, with the least amount of load.

- All subsequent requests for the same URL are then sent to the least loaded server that is selected from a pool of servers that have recently served such requests.

- The selection of a server is also based on a capacity threshold. Should the threshold be exceeded, the policy will assign the request to a “new” server, and add the server to the pool that serves those requests.

The primary goal of LARD is to exploit the caching (locality) in servers, by sending subsequent requests for the same URLs to the same servers. The benefits of caching, however, would only apply to the use of static content, and as such, the policy performs poorly when dynamic content is being served, as discussed in Cardellini et al. (2002).

A.3 Performance Evaluations

A.3.1 Content Aware Policy and Locality-Aware Request Distribution

Research by (Andreolini, Colajanni and Nuccio, 2003) considered the Content Aware Policy (CAP) and the Locality-Aware Request Distribution (LARD) policy within a Web cluster architecture, as illustrated in Schroeder, Goddard and Ramamurthy (2000).

An OSI Model Layer-7 (Webopedia, 2010) dispatcher (named ClubWeb-1w) was introduced with implementation of the policies. The load of each server, defined as the number of TCP connections, was used in order to select a target server that was able to provide processing for classified requests.

The experiment was implement as a one-way dispatcher, where servers responded directly to clients through the use of a custom TCP Handoff mechanism that was able to transfer the client's TCP\IP connection from the task dispatcher to a server node.

The TCP Handoff mechanism effectively bypassed the task dispatcher when responses were returned (from the server), as apposed to the (default) two-way architecture where the task dispatcher sends back the response that it had received from the server node, as shown in Cardellini et al. (2002).

The performance of the policies was evaluated with the httperf benchmarking-tool (httperf, 2011). It was designed to apply Inverse Gaussian distributions for the number of requests per session, and Pareto distributions

for user think time. The CAP policy provided the best throughput results for both CPU and disk-bound tasks.

A.3.2 Content Aware Policy, Service Partitioning and Locality-Aware Request Distribution

Research by Casalicchio, Cardellini and Colajanni (2002) evaluated the Content Aware Policy (CAP), Service Partitioning (Hash) and Locality-Aware Request Distribution policies (LARD), where a simulation and testing model was implemented that was functionally similar to the model that was previously discussed in Casalicchio and Tucci (2001).

The results of the study concluded that the Content Aware Policy (CAP) outperformed both the Service Partitioning and Locality-Aware Request Distribution (LARD) policies with regards to throughput for dynamic content, and noted that LARD provided good throughput with static content requests.