

# A New Architecture for Differentiated Quality of Services in Web Applications

Audran Le Baron<sup>1</sup> and Charles Elkan<sup>2</sup>

<sup>1</sup> École Polytechnique - France

audran.le-baron@polytechnique.org

<sup>2</sup> University of California, San Diego

Department of Computer Science and Engineering

AP&M Building, Room 4856

La Jolla, California 92093-0114

(858) 534-8897—voice

(858) 534-7029—fax

elkan@cs.ucsd.edu

**Abstract.** In this paper, we explore the possibilities of using the new *staged-event driven architecture*—SEDA, designed by Matt Welsh—to introduce differentiated Quality of Service (QoS) in end host applications such as web servers. This architecture, in which applications consist of a network of event-driven *stages* connected by explicit *queues*, was originally designed for highly concurrent Internet services, to introduce *absolute* QoS in end host applications. Here, we intend to introduce *relative* QoS by implementing request classification, differentiated admission control and scheduling policies in SEDA queues. We evaluate the use of this design through a web server.

## 1 Introduction

The objective of this research is to explore the possibilities of a new architecture to implement differentiated Quality of Service (QoS) in web applications. QoS is basically the ability of a network element to guarantee a certain number of service requirements such as bandwidth, low delay or packet loss. Differentiated QoS is then the ability to provide different levels of QoS according to the user identity or the type of request demanded. QoS can take place at every stage of a network: here, our goal is to provide differentiated QoS in end hosts (i.e. in server applications). This includes *absolute* QoS, or performance assurance: we guarantee that all requests are answered—possibly with an error message warning the user that the server is overloaded—within a certain time; and *relative* QoS: this is what *differentiated* QoS is about, we guarantee that *premium* requests are privileged over *basic* requests, i.e. handled in priority.

The need for differentiated QoS in the World Wide Web is becoming more and more evident as stressed in [7]. Web site contents are becoming more and more dynamic, especially in e-commerce applications, where pages tend to be exclusively the results of dynamically computed data. In this context, end hosts

performances have become significant in the global network's performance (as already stressed in [1]), so implementing differentiated QoS in end hosts can reveal very efficient. Several ways to achieve this goal have been studied in different works such as in [2, 3, 5, 6].

Most companies now rely on the Internet to gather vital informations, and are willing to invest a substantial amount of money to have privileged access to crucial data, for example on a particular web site providing highly important financial data. Differentiated QoS is an attractive way to finance web site content by making premium users pay for their privileged access to public data. Actually, the binary solution consisting in authorizing the access to content only to paying users and forbidding it to others, has proved to be inadequate, since this is only viable until another web server provides the same information to everyone for free. Another great field of application is web hosting: individuals, institutions or companies have a provider store and host their web site. The customers, who may pay different amounts of money for this service (usually free for individuals), expect that their potential clients are serviced with a priority level reflecting the amount of money they pay for the host service. Economic models have already been studied in [4] to maximize Service-Level-Agreement (SLA) profits.

## 2 Overview of the Staged Event-Driven Architecture (SEDA)

The Staged Event-Driven Architecture (SEDA) is a design, proposed by Matt Welsh in [8], for highly concurrent Internet services. It is designed to support massive concurrency demands. In SEDA, applications consist of a network of event-driven *stages* connected by explicit *event queues*. The core logic for each stage is provided by the event handler, the input to which is a batch of multiple events. Event handlers may enqueue events onto another stage by invoking an enqueue operation on that stage's incoming event queue. Event handlers do not have direct control over queue operations or threads pools, for this is managed externally by the runtime environment: basically, the only operations available concerning queues at the application level are essentially the enqueue and dequeue operations. The complexity of queue management and dynamic resource controllers are thus hidden from the application programmer's point of view.

Matt Welsh has also implemented a SEDA-based Internet services platform, called *Sandstorm*<sup>3</sup>. Sandstorm is implemented entirely in Java. In Sandstorm—and according to the philosophy of the SEDA architecture—, each application module (or *stage*) implements a simple event handler interface, with a simple method called `handleEvents()` which processes a batch of events pulled from the stage's incoming event queue.

Eventually, Matt Welsh implemented a web server application, called *Haboob*, which relies on the Sandstorm APIs, and which we also used as a base for our implementation and experiments.

---

<sup>3</sup> A Java documentation of Sandstorm can be found at:  
<http://www.cs.berkeley.edu/mdw/proj/seda/javadoc/>.

## 3 Introducing Differentiated QoS in a SEDA-based Application

### 3.1 Request Classification

The first step leading toward differentiated QoS is certainly the request classification: each request received from a client has to be assigned a priority tag (the signification of which will be discussed later in the paper, in section 4.1), regarding a particular criteria. The two main questions in this first step are *how* and *when* to classify requests into different priority classes?

First, *how*? There are basically two main ways to accomplish the classification process:

- A *server-based* method: the nature of the request itself characterizes the priority class it should fall into. For instance, in a web server, the requested URL defines the priority level of the request. This allows for example to prioritize HTML pages over images (by matching the extension of the file in the URL), or a certain part of a web site over another one (by matching the directory), or even a web site over another one (useful when the web server hosts the content of different web sites).
- A *client-based* method: the identity of the client characterizes the priority level of the request. This identification can be made at various levels: the lowest level would be the IP source of the incoming request (this method allows to differentiate client host machines, not users; this can be the right method in some cases, though). Note that the IP address can be masked by a firewall or a proxy, but this would be a way to allow a whole intranet behind a particular firewall to have a privileged access to the data hosted by the server. A higher level identification would rely—for example—on the use of cookies.

Then, *when*? The answer to this question mostly depends on the answer to the first one. The lower-level the classification is, the earlier it can be processed. If the IP identification method is chosen, we can classify the request as soon as the server receives the IP packet from the network. Otherwise, we have to parse the request (the HTTP packet in the case of a web server), which is done in a later stage in a SEDA-based application.

SEDA-based applications are highly modular, and thus permit to change very easily the request classification method one wants to employ. Actually, every stage of the application can easily set the priority level of the elements it is given to handle. This way, the application can decide to set the priority level of its queue elements at any desired stage, regarding any pertinent criteria available at this particular stage. The implementation details—such as how queue elements are attached a priority tag—are explained further in section 4.1.

Once our queue elements are priority-tagged, we need to take this new information into account in queues. This has to be done at two different levels:

- *The admission level*: at each stage, high-priority elements must have a privileged access to queues.

- *The scheduling level:* each stage—or at least each bottleneck stage—has to reorder its queue so that high-priority elements are treated first. This can be done either internally by Sandstorm, or explicitly in the stage implementation.

One last important point about request classification is that it should be done *before* the bottleneck stage(s) of the application, in order to be able to prioritize premium requests in slow bottleneck stages. Actually, a priority scheduling will have a significant impact only in queues where events are susceptible to be pending for a long time, i.e. in queues corresponding to a bottleneck stage. On the contrary, if a high-priority event is pushed onto a queue corresponding to a very fast stage, then the time spent by this event pending on this particular queue will not change significantly, whether the queue implements a priority scheduling policy or not.

### 3.2 Admission Control in Queues

Most of the time, every event's permission to be enqueued should be granted. However, as the application (e.g. the web server) becomes overloaded, queue sizes might explode and make the application unstable on heavy loads, if no admission control is implemented.

This is why the Sandstorm APIs provide the programmer with the possibility to restrict the access to queues at the application level: during the initialization of stages, each queue is assigned a certain admission control policy. Again, originally, this functionality was implemented as an overload protection: it aims at preventing overloads from slowing down the server—or even making it crash—by refusing incoming events to a busy stage. Usually, a simple threshold is fixed for each queue: if a queue size reaches its threshold, enqueue attempts fail, until the queue size diminishes. This threshold constraint is one—simple—of many possibilities to regulate access to queues. Here, our goal is to use admission control not only as an overload protection (absolute QoS), but also as a way to enable *differentiated* QoS by behaving more generously with high-priority requests than with basic ones. Many ways lead to this goal, but the essential idea is usually to fix several trigger points instead of one single threshold to decide whether or not to accept incoming events, given their priority level. The different possible admission control policies and their implementation will be discussed in section 4.2.

Once the admission control policy is chosen, it is very easy to introduce it in a SEDA-based application. Actually, the admission control is implemented in the Sandstorm platform as an interface called `EnqueuePredicateIF`, in which the method `accept()` returns a boolean telling whether or not the given queue element can be enqueued onto the given queue. It is easy to adapt the existing class, or create a new one, implementing the `EnqueuePredicateIF` interface, to take priority tags into account and allow a privileged access to queue sinks for higher-priority elements, according to the chosen admission policy. At the application level, the programmer only has to choose which admission control policy—i.e.

which class implementing the `EnqueuePredicateIF` interface—should be used for each queue.

An important point is that whenever an enqueue operation fails, the application may wish to take some service-specific action, such as sending an error message to the client, or performing an alternate function, such as providing degraded service. This is always better than abruptly stopping the service and closing the connection with the client. This is very easily done in a SEDA-based application: it supposes to test the returning value of every enqueue operation.

### 3.3 Scheduling Policy in *Queue Inboxes*

There are many ways to consider queues and to handle their elements. Let us introduce some basic concepts about queues:

**First In First Out (FIFO) Queue:** this is a well known type of queue. Elements are pushed on its tail and pulled from its head.

**Priority Queue:** here, elements are placed according to their priority level as they come in. The reordering algorithm is then part of the internal queue implementation.

**Queue Inboxes:** instead of dequeuing each element after another, the whole FIFO queue is regularly captured within a box (all the elements are thus pulled off of the queue). The user have then access to an inboxed batch of elements and can treat those elements separately.

The Sandstorm stage interface was designed to make use of such inboxes. Let us have a closer look at the Sandstorm platform to understand this last point.

Originally, the only class implementing the interface `QueueIF` in the Sandstorm platform is the `FiniteQueue` class, which implements a basic FIFO queue. However, the Sandstorm programming model hides the details of queue management from the application. Actually, the only component that a Sandstorm application has to implement is a set of *event handlers*. An event handler implements the core event-handling logic of a stage. It is represented by the interface `EventHandlerIF` in which two particular methods are defined:

`handleEvent(QueueElementIF elem):` this method handles the given incoming event; it is always called by the following method.

`handleEvents(QueueElementIF elem_arr[]):` this method handles an array of incoming events, calling the `handleEvent()` method for each element of the array, after having possibly reordered the array. This array is nothing but the *inbox* we have already mentioned.

Obviously, and as it is said in the Java Documentation of Sandstorm, the `handleEvents()` method permits the programmer to reorder inboxed elements easily at the application level and thus to implement indirectly a scheduling policy. But, as the author *does not* say, this scheme is not equivalent to a priority queue which would reorder elements as they come in: actually, the `handleEvents()` method has only access to the inboxed image of the queue

at a moment  $t$ . While the event handler is handling this inboxed queue, other high-priority events may be enqueued, but they would have to wait for the next `handleEvents()` call before appearing in a new inbox, and thus being handled. The advantage of the inbox scheme is that it hides the queue management from the application level and, at the same time, gives the illusion to be able to reorder incoming events, even if it is only partially correct.

However, it is possible to deal with this interface to reach our goal, which is to be able to handle events according to a certain scheduling policy based on the priority level of each event. Actually, a solution to our problem is to re-enqueue low-priority events—e.g. just once, or as long as there are high priority events pending—, before handling them. This is easily made in a SEDA-based application, since every stage is allowed to enqueue elements to any other stage's queue, including its own one.

Otherwise, the Sandstorm API should be modified so that the reordering process is made within the queue itself, as events are enqueued. This would imply to write a new class implementing the `QueueIF` interface, say a `PriorityQueue` class, which would put incoming events at their right place, according to a particular comparator method given as an argument. Then, at the application level, the programmer should only have access to the first event pending on its queue (i.e. the `handleEvents()` and the concept of queue inboxes should be dropped). Note that this solution would still be faithful to the SEDA philosophy since the queue management would remain part of the core implementation of the Sandstorm platform: the application programmer would only have to choose which comparator method to use within priority queues during the initialization of each stage.

Even though theoretically possible, this solution would imply more changes in the Sandstorm source code, and has not been implemented yet. The current implementation uses the concept of queue inboxes and re-enqueueing operations as described above. Details about the different possible scheduling policies and their implementation are given in section 4.3.

## 4 Implementation Details

### 4.1 Priority Tag

The Sandstorm Java APIs use a special interface for queueable elements: the `QueueElementIF` interface. The only change we had to make in order to attach a priority tag to some of these elements is the creation of a new interface that extends the basic `QueueElementIF` interface. Its name is `PriorityTaggedQueueElementIF`. This interface just adds an integer representing the priority level of the element, and two methods which respectively *set/get* the priority level of/from the given element.

Some of the classes which formerly implemented the `QueueElementIF` interface, were then slightly changed, so they now implement the new

`PriorityTaggedQueueElementIF` instead. In fact, all queueable elements corresponding to internal events (such as error signals used internally by the Sandstorm platform) were left unchanged and thus do not have a priority tag attached to them, whereas elements like HTTP requests and other events directly related to a client connection/request (in fact, all queueable elements visible from the application level) now implement the new priority-tagged interface.

As already mentioned, the priority tag is implemented as an integer. The problem is now to give a significance to this integer. One approach is to interpret each integer—within a fixed range—as a state code. For example:

Code	Signification
0	Basic event
1	Basic event already re-enqueued once
2	Premium event

The state evolves according to some Markov decision process (i.e. basically, a function  $(code, action) \mapsto newCode$  where  $code$  is the state code of the event and  $action$  is the action to be performed—like handling the event or re-enqueueing it—and  $newCode$  is the new state code to be attached to the event). The scheduling policy is entirely described by a total order on the states, and the admission control policy by a function  $(code, environment) \mapsto boolean$ .

This scheme is the most general approach for interpreting the priority tag integer. Practically, we use very few states. As a matter of fact, the current implementation uses the 3-stated interpretation as shown in the table above. Now let us have a closer look to the different ways the priority tag can be taken into account in admission control and scheduling policy implementations.

## 4.2 Admission Control

There are many different possible policies for priority-aware admission controls. All of them should result in rejecting basic requests rather than high-priority ones in case of an overload. One of them is to fix a *numerus clausus*—let it be  $i$ —for basic requests and an overall threshold—let it be  $I$ : the total number of pending elements should never exceed the overall threshold  $I$ , while the number of basic elements should be kept lower than the numerus clausus  $i$ . Another one, even stronger, is to fix two thresholds,  $i < I$ : as long as the queue size is beneath the lowest threshold  $i$ , all incoming elements are admitted, but as soon as it reaches  $i$ , only the high-priority elements are admitted. The second threshold  $I$  is the overall threshold that limits the total queue size. Note that these two simple methods have to be slightly changed if more than two priority levels are taken into account. However, the concept remains the same: one has to fix a certain number of thresholds to limit the number of elements of each class. We can also rely on probabilistic decisions, such as rejecting basic requests 50% of the time in case of overloads.

For our experiments, we used as a basis the original implementation of the `EnqueuePredicateIF` interface using a threshold control (the

`QueueThresholdPredicate` class). The only change we made is to virtually multiply the threshold by 2 for high-priority items (state code 2) and multiply it by 3/2 for items of state code 1 in the `accept()` method. This results in an implementation of the second example of admission policy, with three different thresholds:  $i$ ,  $3i/2$  and  $2i = I$ : the maximum queue size is  $I$  but new basic requests are not admitted if the queue size exceeds  $i$ , neither requeued basic requests when the size reaches  $3i/2$ .

### 4.3 Scheduling Policy

Once the admission control phase is passed, the scheduling policy has to handle the inboxed queue. We have to decide *which* scheduling policy to implement.

*Strict Policy:* the strongest policy consists in handling basic requests *if and only if* no high-priority event occurs in the inbox. Otherwise, only high-priority events would be handled, others being re-enqueued. This would result in a dramatic increase of the number of events asking to be re-enqueued and is not the solution we have adopted.

*Re-enqueue Changes Priority:* in order for basic requests not to be pending too long, we can change the priority level of re-enqueued events, either by lowering it (pertinent in streaming applications, for example) or by increasing it (so that *old* events are treated in higher priority than *new* ones). Then, the scheduling policy consists in handling the highest-priority events and re-enqueueing the lowest-priority ones (one has to define how to differentiate those two categories if there are more than two priority levels).

Again, many other scheduling policies could be imagined. For example, a random policy, which would consist in handling events of priority level  $l$  with a probability  $p_l$  and re-enqueueing them with a probability  $(1-p_l)$ . The highest-priority events would correspond to a probability 1.

In our experiments, we used the second policy: events with a state code of 1 or 2 are considered high-priority and are thus handled immediately, whereas events with a state code of 0 are re-enqueued once.

## 5 Application in a Web Server

In order to experiment the benefits of the SEDA architecture in implementing differentiated QoS in Internet applications, we chose a particular application which plays a preponderant and central role in the World Wide Web today: the web server. Our work has consisted in modifying the GNU public-licensed Java code of *Haboob*, Matt Welsh's SEDA-based implementation of a web server. In fact, very few modifications had to be made, once the Sandstorm platform was prepared to accept priority-tagged queue elements. Essentially, we had to add a few lines in the `HttpRecv` stage code in order to initialize the priority tag of each incoming HTTP request according to the requested URL. Then, we had to take into account this priority-tag in every `handleEvents()` method of later stages.

## 6 Design of Experiments

**Server** For the need of experiments, we made use of a simulated bottleneck stage in the server: this stage simply sleeps for 20 ms for each handled request and sends an 8 KB response to the client. This way, the maximum response rate is 50 responses per second (even if error responses can be sent at a much higher rate). Requests are classified by parsing the requested URL in the HTTP header, at the `HttpRecv` stage of Haboob. Admission control is only activated at the bottleneck stage whereas scheduling policies are activated both at the bottleneck stage and the `HttpSend` stage. Whenever an enqueue operation fails onto the bottleneck stage's queue (actually, this is the only queue which is susceptible to reject events), a "503 Service Unavailable" response is enqueued onto the `HttpSend` stage.

**Client** The client load generator we used is a modification of the one provided in the Sandstorm-Haboob package. Its behavior is as follows:  $n$  clients continually request a Web page to the server, receive the response, read it, sleep for 20 ms before sending another request, and so forth. So the number of clients indicates basically the number of requests that are simultaneously pending on the server queues at any time. Among those  $n$  clients, some will exclusively send premium requests while others will exclusively send basic requests. The proportion of each is fixed during the initialization of the HTTP loader.

**Environment** All measurements were taken with the server running on a 2-way SMP 800 MHz Pentium III system with 896 MB of RAM and Linux 2.4.3. The client machine was a 2-way SMP 350 MHz Pentium II with 256 MB of RAM and Linux 2.2.19. IBM J2RE 1.3.1 was used as the Java platform. The two machine are connected via a 10 Mbps Ethernet network, with an average round-trip-time of 1 ms.

**Goal** Our experiments aim at evaluating the effects of admission control and scheduling policy on Haboob's behavior, confronting response times and ratio of completed requests for basic clients against premium ones. Note that the Sandstorm platform provides ways to meet *absolute* QoS very easily, by making the runtime environment dynamically tune thread pool sizes and queue thresholds. Actually, this is precisely what the SEDA architecture was designed for originally: these features have already been successfully tested by Matt Welsh. Here, our concern is *relative* QoS: so we will focus on the difference of service offered to the two different types of client requests in various situations, rather than on the absolute quality of service offered to each, taken separately.

**Variables set/measured** The two main parameters that are pertinent to make vary are the total number of clients and the proportion of premium ones. In a first set of experiment, we kept the proportion of premium clients constant, equal

to 20%, while the total number of clients increases from 5 to 800. This allows to see the behavior of Haboob both in normal load and in overload. In a second set of experiments, we made both parameters vary together so that the total number of premium clients remains constant, equal to 25. This allows to show how the service offered to premium clients can be disturbed by a heavy increase of basic requests.

The variables measured are response times, response rates and percentage of rejections. For each variable, we distinguish those that correspond to basic/premium requests, and also those that correspond to true completions or to “503 Service Unavailable” response.

**Expected Results** First, we expect that the mechanisms employed to introduce differentiated QoS do not diminish the global performances of Haboob. In particular, we expect the total true completion rate to remain constant, equal to 50 completions per second (maximum rate possible due to the 20 ms sleep in the bottleneck stage).

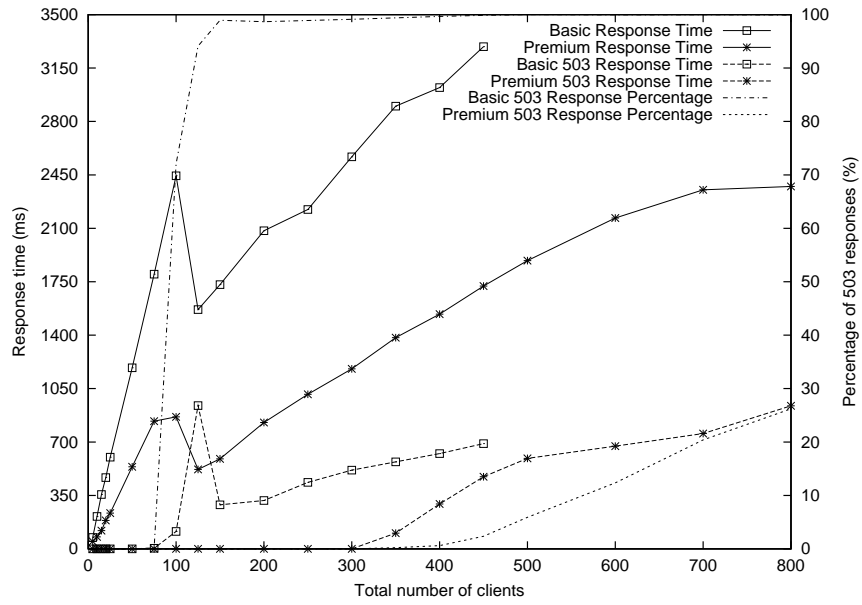
In a normal load, we expect that premium requests are answered much more quickly than basic ones, none of them being ever rejected. In a reasonable overload, we expect that a majority of basic requests are dropped and sent a 503 response, while premium requests keep being answered normally, yet with a slight increase in the response time due to the additional computation needed to send 503 responses. Eventually, in case of an important overload, we expect an increasing proportion of premium requests to be also answered with a 503 response, whereas no basic request should pass through the admission control barrier.

## 7 Results of Experiments

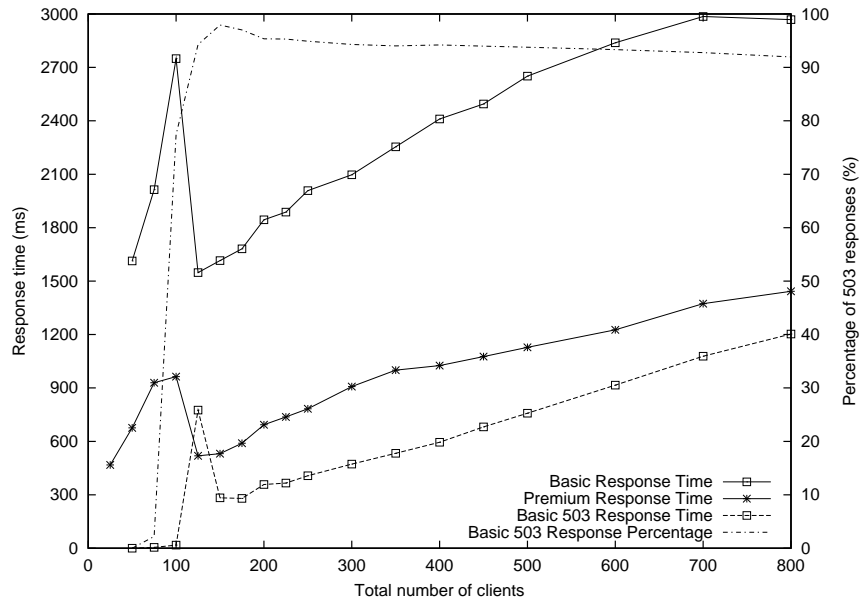
First of all, the experiments showed that in any circumstance, the total true completion rate reaches the maximum of 50 completions per second.

In the first set of experiments (see figure 1(a)), we observed that premium requests are always answered in a far less amount of time than basic ones, approximatively twice more quickly, in any circumstance. As soon as the number of clients exceeds 50, some basic requests start being rejected, as it was expected, given the queue size of 50. From 200 to 400 clients, more than 90% of basic requests are rejected, while all premium requests keep being handled normally. When the number of clients reaches 400, even premium requests start being rejected, but in a very lower rate than basic ones, with a response time tending to be around 2 s. Beyond 450 clients, basic requests are all rejected. An important point too, is that 503 responses are sent very quickly (for basic clients as well as for premium ones): a user would not want to wait for a long time just to get an error message.

In the second set of experiments (see figure 1(b)), we observed that the service offered to the 25 premium clients is merely affected by the increasing number of basic clients. Actually, only a slight increase of the response time can



(a) 20% of premium clients



(b) 25 premium clients

Fig. 1. Results of the two sets of experiments

be observed, while basic requests are rapidly rejected with a 90% rate and more as soon as the total number of clients exceeds 100.

## 8 Discussion and Conclusion

We encountered no real difficulty in tuning the Sandstorm platform and the Haboob web server to incorporate differentiated QoS in them. The SEDA architecture appeared very attractive at first sight to enable differentiated QoS in its applications, and has finally revealed to be so. It provides a very simple environment for application programmers. A web server is one of many possible applications. Actually, Matt Welsh also implemented successfully a Gnutella packet router, using the same architecture and the Sandstorm APIs. Moreover, the performance gap between Java and statically compiled languages are closing; as a matter of fact, the Java-based SEDA web server Haboob outperforms two popular web servers implemented in C—Apache and Flash—in Matt Welsh’s experiments (see [8]). This is why we believe that the SEDA architecture is very profitable to design highly concurrent Internet services enabling differentiated QoS. In future work, we will use control theory to analyse scheduling policies and find optimal ones in order to improve SEDA-based application’s behavior.

## Authors

Audran Le Baron is a French student from the École Polytechnique. This work was made in the context of a research internship with Professor Charles Elkan, in the Computer Science Department of UCSD.

## References

1. Jussara Almada, Virgílio Almada, and David J. Yates. Measuring the Behavior of a World-Wide Web Server. 1996.
2. Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing Differentiated Levels of Service in Web Content Hosting. Technical Report CS-TR-1998-1364, 1998.
3. N. Bhatti and R. Friedrich. Web Server Support for Tiered Services, September 1999.
4. Zhen Liu, Mark S. Squillante, and Joel L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proceedings of the ACM Conference on Electronic Commerce (EC’01)*. ACM, 2001.
5. Sook-Hyun Ryu, Jae-Young Kim, and James Won-Ki Hong. Approaches to Support Differentiated Quality of Web Service. 2001.
6. Nikolaos Vasaliou and Hanan Lutfiyya. Providing a Differentiated Quality of Service in a Wold Wide Web Server. 2000.
7. Nikolaos Vasiliou. Overview of Internet QoS and Web Server QoS, April 2000.

8. Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 230–243, New York, October 21–24 2001. ACM Press.