

RFQ: Redemptive Fair Queuing

Ajay Gulati¹ and Peter Varman²

¹ VMware Inc, Palo Alto, CA

² Department of ECE and Computer Science
Rice University, Houston TX 77005, USA
agulati@vmware.com, pjv@rice.edu

Abstract. Fair-queuing schedulers provide clients with bandwidth or latency guarantees provided they are well-behaved *i.e.* the requested service is always within strict predefined limits. Violation of the service bounds results in nullification of the performance guarantees of the misbehaving client.

In this paper we relax this notion of good behavior and present a generalized service model that takes the current system load into consideration. Consequently clients may opportunistically consume more than their contracted service without losing future performance guarantees, if doing so will not penalize well-behaved clients. We present a new algorithm RFQ (Redemptive Fair Queuing) along with a generalized traffic model called the Deficit Token Bucket (DTB). RFQ incorporates the notion of redemption, whereby a misbehaving client may be rejuvenated and regain its performance guarantees. We characterize the conditions for rejuvenating a client, and prove that RFQ meets its performance guarantees in the DTB model.

1 Introduction

The popularity of hosted application services, and benefits in cost, energy, and manageability of a shared infrastructure has spurred interest in server virtualization and storage consolidation technologies [1, 2]. This has created the need for flexible and robust resource management strategies for client isolation and QoS provisioning in shared server systems [3]. *Resource scheduling* is used to provide each workload with the abstraction of having its own dedicated server, while flexibly sharing server capacity to handle bursts or low latency requests. A client's performance requirements are usually expressed by a combination of *throughput* and *latency* constraints. Throughput refers to the average rate of service completion, while latency or response time is the interval between the arrival time of a request and the time it completes service. A database workload, for instance, may have strict response time requirements for its transactions, whereas a back-up or other file transfer application might care more about overall throughput than the latency of individual requests.

The Service Level Agreement (SLA) of client c_i has three parameters $(\sigma_i, \rho_i, \delta_i)$: ρ_i is the *average* (long term) service demand of c_i , the *burst parameter* σ_i specifies the allowable instantaneous deviation from ρ_i (as made precise later), and δ_i is a bound on the *latency* of its requests. The scheduler aims to provide c_i a throughput of ρ_i and a response time guarantee of δ_i as long as c_i is well-behaved (*i.e.* honors its SLA as

defined precisely later). In current models [4–6] a client is considered *well-behaved* if in *every* time interval of size $T \geq 0$, the total amount of service requested is upper bounded by $U(T) = \sigma_i + \rho_i \times T$. Such an arrival model is also known as the token bucket model [7] with parameters σ_i and ρ_i . Well-behaved clients restrict the amount of service requested in any time interval (either voluntarily or by request dropping) in return for receiving guaranteed response times.

A major drawback of current QoS schedulers is their fragility with respect to errant client behavior. If a client violates its SLA by requesting more than $U(T)$ service in *any* interval T , the deadline guarantees for the client are effectively nullified. Unfortunately, this nullification is not restricted to just the offending requests but can persist indefinitely into the future. (See Example 1 in Section 2). In a practical setting this restriction is unacceptable. When the server has unused capacity (since not every client is sending at the maximum rate at all times), it is desirable to allow clients who can use the excess service to do so, without penalizing them in the future for their use of spare capacity. A robust model should distinguish between benign violations of input limits (as when a client merely utilizes spare capacity) from critical violations where the client should be penalized to prevent it from encroaching on the shares of other clients.

In this paper we make the following contributions: (a) we define a dynamic traffic model called the Deficit Token Bucket (DTB) model. DTB allows one to define a weaker notion of well-behaved clients that distinguishes between benign and critical arrival violations; (b) we present a new scheduling algorithm RFQ (Redemptive Fair Queuing) that provides performance guarantees under this weaker model of well-behaved clients, and includes mechanisms to accelerate the rehabilitation of a misbehaving client; (c) we provide an algorithm for proportionate bandwidth allocation by modifying certain parameters of RFQ. The resulting algorithm achieves the optimal worst case fairness index [8], an important measure of the quality of a fair-queuing scheduler. Note that RFQ meets its deadlines in all situations where the existing schedulers do, in addition to situations where the latter do not. We believe this is the first algorithm which can successfully distinguish between benign and critical use of spare capacity, that allows better server utilization and greater flexibility for the clients.

2 Relation to Previous Work

Formal work related to Fair Queuing algorithms for QoS-based resource allocation falls into two categories. First is a class of scheduling algorithms for proportionate bandwidth allocation such as PGPS [9], Virtual Clock [10], WFQ [11, 12], WF^2Q [8], SFQ [13], SCFQ [14], Leap Forward Virtual Clock [15], Latency-rate Servers [16], Greedy Fair Queuing [17], Elastic Round Robin [18], and Time Share Scheduling [19], which guarantee weight-proportional throughput to clients by dividing up the server bandwidth fairly between them. A fundamental limitation of these algorithms is the strong coupling between their throughput and response time guarantees. The latency of a client’s requests is fixed by its bandwidth allocation, resulting in over-provisioning for clients with low throughput and low latency requirements. A corollary to this is the inability of these algorithms to handle bursts effectively. The algorithms do not distinguish a client that sends its requests at a uniform rate from a client sends the requests

in periodic bursts, as long as the average request rates are the same. Both clients will receive the same bandwidth allocation, but the bursty client will incur large latencies. On the other hand, an algorithm with independent latency and throughput controls can schedule the requests to be sensitive to the latencies.

The second class of scheduling algorithms [4–6, 20, 21] are latency-sensitive in that both throughput and response time constraints may be independently specified provided certain capacity constraints are met. The fundamental result in this regard is the SCED [4, 5] algorithm to schedule workloads specified by a given set of service curves that meet the capacity constraints. However, this solution and its successors have the fundamental drawback that a client that uses spare capacity may get starved in the future when resource contention is high. We present a detailed example below to show the issue of starvation and the possibility for indefinite response time penalties in the SCED algorithm. In the example tag refers to the value assigned to a request that is used as a scheduling priority. Tags are spaced by the inverse of the throughput ρ_i when c_i is backlogged; the scheduler dispatches requests to the server in the order of these tags.

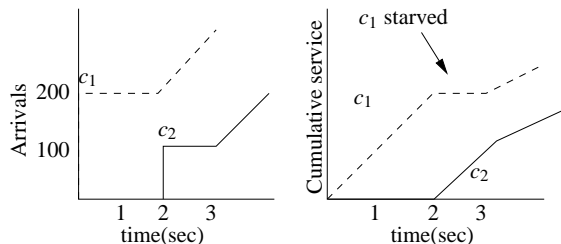


Fig. 1. c_1 is starved during [2,3] for using spare capacity during [1,2]

Example 1: Consider a system with two clients c_1 and c_2 with throughput requirements $\rho_1 = \rho_2 = 50$ req/s. Assume the system capacity is 100 req/s. Suppose that c_1 sends a burst of 200 requests at $t = 0$, and then sends requests at its desired rate of 50 req/s from $t = 2$ onwards. Suppose that c_2 is idle till $t = 2$, sends a burst of 100 requests at $t = 2$, and then sends requests at a steady rate of 50 req/s after $t = 3$. The input profiles are shown in Figure 1.

Now in the interval $[0, 2]$, c_1 can utilize the capacity unused by c_2 and will receive 100 req/s instead of its stipulated rate of 50 req/s. All these 200 requests will complete service by $t = 2$, but its tags (that are spaced apart by $1/\rho_i = 1/50$) will have reached a value of $200 \times 1/50 = 4$ (much higher than the real time). Hence future requests of c_1 arriving after $t = 2$ will have tags beginning at 4 and increasing in increments of $1/50$. When c_2 becomes active at time $t = 2$, the 100 requests of c_2 will receive tags starting at 2 and spaced by $1/50$; all these tags are less than the tags of the pending requests of c_1 . Hence, c_2 will get all the service, and complete its burst of 100 requests in 1 second at $t = 3$, starving c_1 of any service in this interval. After time $t = 3$ the requests of c_1 and c_2 are interleaved, since the tags of both c_1 and c_2 now begin at 4 and are separated by $1/50$. However, requests of c_1 have been pending since $t = 2$ and therefore these and all future requests of c_1 incur a latency of at least 1 second.

A better schedule can be constructed as follows. Serve the 200 requests of c_1 in the interval $[0, 2]$ as before. From $t = 2$ onwards, give both c_1 and c_2 their guaranteed

service rate of 50 req/s by interleaving the requests of c_1 and c_2 . This ensures that neither c_1 nor c_2 will be starved of service, and that all requests of c_1 arriving after $t = 2$ will be served without delay. Note that c_2 missing its deadlines is justified since it is requesting more than its share during a period when there is no spare capacity in the system. In contrast, in the traditional fair scheduler c_1 misses deadlines indefinitely even though its only excess is to use unused service capacity between $[0, 2]$. Our recent algorithm p Clock [6] provided a solution to this problem, and showed empirically that it was able to avoid starvation in many cases. However, the formal conditions under which spare capacity can be safely used were no stronger than earlier results.

3 Model and Definitions

The system provides shared service to a set of m clients, $c_i, 1 \leq i \leq m$. Each request brings in a demand for a specified amount of service (e.g. IO requests, CPU time etc.). Clients' requests are held in private queues, and are dispatched to the server one at-a-time by the scheduler. The server capacity denoted by C is the rate at which it provides service. Time is represented by discrete time steps $t = 0, 1, 2, \dots$. The SLA parameters of c_i are denoted by $(\sigma_i, \rho_i, \delta_i)$. The size of the request made by c_i at time t is denoted by $s_i(t)$, $0 \leq s_i(t) \leq \sigma_i$. The total amount of *service requested* by c_i in the interval $[a, b]$, is denoted by the *arrival function* $\mathcal{R}_i(a, b) = \sum_{t=a}^b s_i(t)$. The amount of *service provided* to c_i in the interval $[a, b]$ is denoted by $\mathcal{S}_i(a, b)$. A client is backlogged at some time instant if it has one or more requests pending in the queue or currently in service.

Definition 1. The **backlog** $B_i(t)$ of a client c_i at time t is defined as $B_i(t) = \mathcal{R}_i(0, t) - \mathcal{S}_i(0, t)$. Client c_i is said to be **backlogged** at time t if $B_i(t) > 0$. A **system busy period** is a maximal-sized time interval during which at least one client is backlogged. The maximum size of any request is denoted by R_{max} . The **maximum service time** of a request is denoted by $\varepsilon = R_{max}/C$.

Requests are classified as either *good* or *bad* based on two factors: (i) the total amount of service requested by the client relative to its SLA parameters, and (ii) the actual rate at which the client has received service. Good requests will be serviced within their stipulated response time, while bad requests cannot be guaranteed. This is the fundamental point of departure from previous schemes. In earlier models only the first factor is used to classify requests, while in our new model the load of the server is implicitly taken into account. If the server has sufficient capacity to absorb extra service without hurting the other clients, then future requests of this client are classified as good, and compete fairly with the other clients.

In order to classify requests we use a modified form of a token bucket algorithm [7]. For a client with arrival parameters σ and ρ we refer to it as (σ, ρ) -DTB (Deficit Token Bucket) model. Initially the bucket contains σ tokens. A request of size s will reduce the number of tokens in the bucket by s . If this results in the number of tokens becoming negative the request is classified as *bad*; else the request is *good*. The bucket is continually refilled with tokens at the constant rate of ρ but the maximum number of tokens is capped at σ . In addition, at a *synchronization point*, the number of tokens of a client c_i with no current backlog is increased to σ_i . Intuitively a synchronization point detects

that there is unused server capacity and makes it available to clients. Clients that have a backlog are still paying the price for past misbehavior and therefore no tokens are added to their buckets.

Each client c_i is controlled by its own (σ_i, ρ_i) -DTB. Figure 2 shows the number of tokens as they change in DTB model. The solid line shows the total number of tokens accumulated and the dotted line shows the total service requested as a function of time. The difference between the two is the tokens available at that time instant. If the total service requested exceeds the cumulative tokens (e.g. beyond time a), the number of tokens will be negative. On the other hand, if a client gets idle the number of tokens will continue to increase at a rate ρ_i , but will be capped at σ_i .

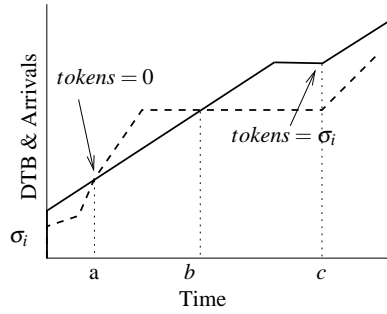


Fig. 2. DTB model for a client c_i

Definition 2. A request from c_i of size s that arrives when the number of tokens in its bucket is at least s is **good**; else the request is **bad**. Client c_i is **well-behaved** in the interval $[a, b]$ if all its requests that arrive in the interval are good; else it is said to **misbehave** in that interval.

Service guarantees can only be met if admission control ensures that the system has sufficient capacity to meet requirements of admitted clients. A *lower-bound* on the system capacity, referred to as the **System Capacity Constraint**, is stated below. This is derived by considering the situation in which all clients c_i , $1 \leq i \leq m$, simultaneously send their maximum bursts (σ_i) at $t = 0$, followed by sending requests continuously at their designated throughput rates (ρ_i). If any of the inequalities in Definition 3 is violated, at least one request in the above arrival set will miss its deadline. The first constraint is needed to ensure that all c_i can receive throughput ρ_i . The second constraint follows by noting that for $k \leq i$, c_k must complete $\sigma_k + \rho_k(\delta_i - \delta_k)$ amount of service by δ_i for it to meet its deadlines.

Definition 3. Let the clients be arranged in order of non decreasing latencies, represented by $\delta_1 \leq \delta_2 \leq \dots \leq \delta_m$.

The **System Capacity Constraint** is defined by the following equations:

$$\sum_{\forall i} \rho_i \leq C \quad (1)$$

$$\forall i, \sum_{k=1}^i \sigma_k + \rho_k(\delta_i - \delta_k) \leq C \times \delta_i \quad (2)$$

In Section 5, we show that these conditions are also *sufficient* to guarantee that no good requests are delayed.

4 Scheduling Algorithm RFQ

The algorithm *RFQ* is based on tagging requests to reflect their scheduling priority. Each request receives a *start tag* and a *finish tag*. The start tags determine eligibility of requests for scheduling; at any instant, only requests with start tags no more than the current time are eligible. Among eligible requests the one with the smallest finish tag is chosen and dispatched to the server. Pseudo code of algorithm is presented in Algorithm 1 below.

```

1 Request Arrival:
2 Let  $t_a$  be arrival time, of request  $r$  from  $c_i$ ;
3 UpdateNumtokens();
4 ComputeTags();

1 Request Completion:
2 Let  $t_c$  be time the current request completes;
3 AdjustTags ();
4 Dispatch ();

```

Algorithm 1: RFQ algorithm

There are two actions performed by the scheduler on a request arrival: *UpdateNumtokens* implements the DTB model by updating the number of available tokens for this client; *ComputeTags* assigns start and finish tags to the request to be used by the dispatcher. When a request completes service two actions are again required: *AdjustTags* is used for synchronizing tags with real time and rejuvenating clients as necessary, while routine *Dispatch* is used to select the next request to send to the server. These components are detailed in Algorithms 2 and 3.

UpdateNumtokens: For each client c_i , the routine maintains a variable *numtokens_i* that tracks the amount of tokens in its bucket at an arrival instant. The initial value of *numtokens_i* is set to σ_i at the start of a system busy period. The amount of tokens increases at the uniform rate of ρ_i . Hence in an interval of Δ seconds it will be incremented by $\Delta \times \rho_i$, but the total amount of tokens is capped at σ_i .

Compute Tags: This routine assigns *start* and *finish* tags (S_i^r and F_i^r respectively) to the request r arriving from c_i . The value assigned to the start tag S_i^r depends on whether the request is good or bad. Let the size of the request be s_i . If the request is good (*numtokens_i* $\geq s_i$), S_i^r is set to the current time t . The start tag of a bad request is set to a time in the future, specifically the earliest time at which the bucket would have accumulated s_i tokens if there are no further withdrawals. If there are $n \geq 0$ tokens currently in the bucket, the additional $(s_i - n)$ tokens required will be accumulated in a further $(s_i - n)/\rho_i$ time. The condition $n < 0$ signifies that there are pending requests with start tags beyond the current time; the request needs to earn s_i tokens and join the

end of the queue of pending requests. This is done by setting its start tag to s_i/ρ_i beyond the largest start tag of c_i at this time. F_i^r is set to the sum of S_i^r and the latency bound δ_i . The count of tokens for the client is decremented by the service required (s_i), and the number of pending requests for the client is updated. The variables $MinS_i$ and $MaxS_i$ track the smallest and largest start tags of c_i respectively, and are updated as necessary.

UpdateNumtokens:

On arrival of request r of size s_i from client c_i at time t_a ;
 Let $\Delta = t_a -$ the arrival time of the previous request of c_i ;
 $numtokens_i += \Delta \times \rho_i$;
if ($numtokens_i > \sigma_i$) **then**
 $numtokens_i = \sigma_i$

ComputeTags:

On arrival of request r of size s_i from client c_i at time t_a ;
if ($numtokens_i \geq s_i$) **then**
 $S_i^r = t_a$;
else
 if ($numtokens_i > 0$) **then**
 $S_i^r = t_a + (s_i - numtokens_i)/\rho_i$;
 else
 $S_i^r = MaxS_i + s_i/\rho_i$

$MaxS_i = S_i^r$;
 $F_i^r = S_i^r + \delta_i$;
 $numtokens_i = numtokens_i - s_i$;
 $backlog_i = backlog_i + 1$;
if ($backlog_i = 1$) **then**
 $MinS_i = S_i^r$

Algorithm 2: Components of RFQ algorithm at request arrival

AdjustTags: The routine checks for the condition where the start tags of all pending requests are greater than the current time t_c . We call this a *synchronization point*. Rather than allowing the tags to keep running ahead of real time (which is the fundamental cause for starvation; see Example 1 of Section 2), the algorithm synchronizes the backlogged clients and the idle clients. At a synchronization point, the algorithm shifts the tag values of all requests at the server by a fixed amount, so that the smallest start tag after the adjustment coincides with the current time t_c . The relative values of the tags are not changed: they just shift as a block by an offset equal to the difference between the smallest start tag in the system and the current time. Since new clients will begin their tags from the current time as well, all clients compete fairly from this point on, avoiding starvation. Note that an implementation just needs to maintain an offset value equal to the the shift amount, and does not need to explicitly alter each tag.

The occurrence of a synchronization point also indicates that there is spare server capacity that can be reallocated without risking future guarantees. Hence the routine also checks if there are any clients that can be rejuvenated. If client c_i at a synchronization point has a backlog of zero, then it can be rejuvenated and the number of tokens is changed to σ_i .

Note that this shift in tag values and infusion of tokens by rejuvenation raises the possibility that some clients may now miss their deadlines, since more requests are pushed into a given time interval. However, as we show in Theorem 1 the readjustment of tags does not result in any missed deadlines, since it exactly compensates for the spare capacity detected by the synchronization point.

Dispatch: This routine is invoked on the completion of each request. It selects a pending request to dispatch to the server. E is the set of *eligible requests* consisting of the requests whose start tags are less than or equal to the current time. Note that the *AdjustTags* routine guarantees that E is not empty as long as there is at least one request in the system, by forcing a synchronization if necessary. From the eligible requests, the one with the earliest finish tag is selected for service. The number of pending requests ($backlog_k$) and minimum start tag ($MinS_k$) of the selected client c_k are updated appropriately.

AdjustTags:

Let A be the set of currently backlogged clients at time t_c ;

if ($\forall j \in A, MinS_j > t$) **then**

$mindrift = \min_{j \in A} \{MinS_j - t_c\}$;

$\forall j \in A$, Subtract $mindrift$ from $MinS_j, MaxS_j$ and all *start* and *finish* tags;

$\forall j \notin A, numtokens_j = \sigma_j$;

Dispatch:

On completion of request from client c_i at time t_c ;

Let E be the set of pending requests with start tags no more than t_c ;

From E , select the request w with minimum finish tag and dispatch it to the server. Let the chosen request be from client c_k ;

$backlog_k = backlog_k - 1$;

if ($backlog_k > 0$) **then**

Let the start tag of the next pending request of c_k be S_k^r ;

$MinS_k = S_k^r$;

Algorithm 3: Components of RFQ algorithm on request completion

5 Proof of Correctness

In this section we provide a proof of the scheduling guarantees of RFQ. We will show that if the system capacity satisfies the constraints noted in Definition 3, then every good request will meet its deadline. Note that the definition of good requests includes clients that may have misbehaved in the past, but have since been rejuvenated.

Definition 4. A **synchronization point** is a departure instant t at which all start tags are greater than t . Immediately after the synchronization, the minimum start tag is t and every flow with zero backlog at t is rejuvenated so that it has σ_i tokens.

Lemma 1. If a good request completes service before the finish tag assigned to it by RFQ, then the request meets its latency bound.

Proof. Consider a request r of c_i that arrives at some time instant t . Since the request is good, RFQ will set its start tag to the arrival time t and the finish tag to $t + \delta_i$. Hence if r finishes service by its finish tag, it meets its latency bound.

Lemma 2. Consider an interval $[a, a + \tau]$ in which a is a synchronization point and there are no more synchronization points between a and $a + \tau$. For any c_i , the amount of its service that is assigned start tags in the interval $[a, a + \tau]$, is upper bounded by $\sigma_i + \tau \times \rho_i$.

Proof. (Sketch) Within the interval there is no adjustment of tags or addition of tokens through rejuvenation. Recall that start tags are assigned so that a request has to pay for the service it requests by either getting the tokens from the bucket or delaying the start tag till it generates the required number of tokens, at the rate ρ_i . Since there are at most σ_i tokens initially in the bucket, the result follows.

Lemma 3. For any time interval $[a, b]$ in which a is a synchronization point and there are no synchronization points between a and b , the total amount of service with start tags greater than or equal to a and finish tags less than or equal to b , is no more than $C \times (b - a)$.

Proof. Without loss of generality, let the clients be indexed in non-decreasing order of their latencies so that for any two clients c_i and c_j , $1 \leq i < j \leq m$ implies that $\delta_i \leq \delta_j$. Let $n \leq m$ be the largest index such that $\delta_i \leq b - a$.

Consider a client $c_i, i \leq n$. Since there are no synchronization points in $(a, b]$, tags are not changed by AdjustTags at any time during this interval. Since a request of c_i with finish tag less than or equal to b must have its start tag no more than $b - \delta_i$, we must bound the amount of service of c_i with start tags in the interval $[a, b - \delta_i]$. From Lemma 2, the amount of service with start tags in the interval is upper bounded by $\sigma_i + (b - \delta_i - a)\rho_i$. For clients with index $j > n$, the amount of such service is 0.

Summing the bounds over all the clients, the amount of service is bounded by:

$$\sum_{i=1}^n (\sigma_i + \rho_i(b - a - \delta_i)) \quad (3)$$

Now from the Capacity Constraint equation 2 applied to the case $i = n$ we have:

$$\sum_{k=1}^n \sigma_k + \sum_{k=1}^n (\delta_n - \delta_k)\rho_k \leq C \times \delta_n \quad (4)$$

Applying equation 1 of the Capacity Constraint to the non-negative interval $b - a - \delta_n$ we have

$$\sum_{k=1}^n (b - a - \delta_n)\rho_k \leq C \times (b - a - \delta_n) \quad (5)$$

Combining equations 4 and 5 we get:

$$\sum_{k=1}^n \sigma_k + \sum_{k=1}^n (b - a - \delta_k) \rho_k \leq C \times (b - a) \quad (6)$$

Hence, the service required (Equation 3) is bounded by $C \times (b - a)$.

Theorem 1. *Consider the set of requests of c_i that arrive during an interval in which it is well behaved. The irrespective of the behavior of other clients, all these requests have a latency bounded by $\delta_i + \varepsilon$.*

Proof. We will prove the theorem by contradiction. Let Γ denote the set of requests of c_i that arrive in the interval. Since all requests in this interval are good the finish tag and the deadlines are the same. Assume to the contrary that a request in Γ with a finish tag t_d completes later than time $t_d + \varepsilon$.

Let t_0 be the start of the system busy period in which t_d occurs. Let t be the last time instant before t_d during which a request having finish tag greater than or equal to t_d begins service. If there is no such request it follows that all requests that are serviced in the interval $[t_0, t_d]$ have finish tags no more than t_d . Since the start tags of these requests are greater than or equal to t_0 , from Lemma 3 we know that all these requests can be serviced in the interval $[t_0, t_d]$, which contradicts our assumption proving the theorem.

Otherwise $t_d > t \geq t_0$. Since a request with finish tag greater than or equal to t_d begins service at time t , it implies that there were no eligible requests with finish times less than t_d at t .

Partition the requests of Γ into two sets. Let \mathbf{P} be the set of eligible requests at time t (*i.e.* having a start tag less than or equal to t), and \mathbf{Q} be the set of request with start tags greater than t . The request scheduled at t must be from \mathbf{P} and since it has a finish tag greater than t_d , either this was the only request in \mathbf{P} or all requests of \mathbf{P} must have finish tags greater than t_d . In both these cases, no additional request of \mathbf{P} can be scheduled before t_d . This is because in the first case there are no more requests, and in the second case it would contradict the definition of time t (*i.e.* the last instant before t_d when a request with deadline after t_d is scheduled). Hence the only requests scheduled between t and t_d are one request from \mathbf{P} and requests from \mathbf{Q} . Since the requests of \mathbf{Q} have start tags greater than t the service required by the requests of \mathbf{Q} with deadline t_d or less, is bounded by $C \times (t_d - t)$ by Lemma 3. The service required by the request of \mathbf{P} that was scheduled at t is no more than $C \times \varepsilon$. Adding this to the service bound on \mathbf{Q} , the total service required after t is no more than $C \times (t_d - t + \varepsilon)$. Hence all of these requests will finish by $t_d + \varepsilon$ – a contradiction.

5.1 RFQ as a Pure Bandwidth Allocator

We now show how to modify RFQ to act as a pure bandwidth allocator. We ignore σ_i and $numtokens_i$ and assign $\delta_i = s_i / \rho_i$, where s_i is the size of the current request of client c_i . We simplify *ComputeTags* to ignore the different conditions based on *numtokens*, and instead to always assign tags as follows: $S_i^r = \max\{t_a, MaxS_i\}$, $F_i^r = S_i^r + \delta_i$ and $MaxS_i = S_i^r + s_i / \rho_i$. Similar as before, the scheduling is done using the minimum finish tag from the set of eligible requests.

In this case, the latency encountered by any request (referred to as the *intrinsic delay*) should ideally depend on the total service in its queue (including this request) and its guaranteed rate of service only, and not on the other clients [8]. We show below that RFQ meets these bounds.

Theorem 2. *The intrinsic delay of a request r from client c_i arriving at time τ is bounded by $\frac{Q_i(\tau)}{Cw_i} + \varepsilon_i$, where w_i is c_i 's share of system capacity, i.e. $w_i = \rho_i / (\sum_{\forall j} \rho_j)$, $Q_i(\tau)$ is the amount of service pending in queue of c_i at time τ , and ε_i is a parameter independent of the other clients in the system.*

Proof. (Sketch) Let the total service of all pending requests of c_i including r be $Q_i(\tau)$. From the assignment it can be seen that r will get a finish tag no more than $\tau + Q_i(\tau)/\rho_i + s_i/\rho_i$, where s_i is the size of last completed request of c_i . We count the total amount of pending service from clients $c_j, j \neq i$, with finish tags no more than $\tau + Q_i(\tau)/\rho_i + s_i/\rho_i$. Now either all the start tags from other clients are higher than $\tau - R_{max}/C$ (which means that no one was eligible for scheduling) or the finish tags are $\geq \tau + s_i/\rho_i$. In the first case, the total service requested by client c_j with finish tags in the interval is bounded by $\rho_j \times (Q_i(\tau)/\rho_i + s_i/\rho_i + R_{max}/C)$. At most these requests will be serviced before r completes. The total amount of service from the tagged requests of all clients is bounded by $((Q_i(\tau) + s_i)/\rho_i) + R_{max}/C \sum_j \rho_j \leq (Q_i(\tau) + s_i)/w_i + R_{max}$. The time required to service this is bounded by: $(Q_i(\tau) + R_{max})/(Cw_i) + R_{max}/C$. Thus we get a bound for the maximum delay as: $Q_i(\tau)/(Cw_i) + \varepsilon_i$, where $\varepsilon_i = R_{max}/(Cw_i) + R_{max}/C$.

In second case we know that all the requests with finish tag $\leq \tau + s_i/\rho_i$ will be able to finish by the time $\tau + s_i/\rho_i$, because $C \geq \sum_{\forall k} \rho_k$. Thus the total amount of finish tags in the interval $[\tau + s_i/\rho_i, \tau + Q_i(\tau)/\rho_i + s_i/\rho_i]$ from all clients is again bounded by $(Q_i(\tau) \sum_{j \neq i} \rho_j)$. The overall bound in this case would be $Q_i(\tau)/(Cw_i) + R_{max}/(Cw_i)$.

6 Conclusions

We presented a novel algorithm RFQ to provide independent bandwidth and latency guarantees to multiple clients sharing a server. Our algorithm improves upon previous schemes significantly by differentiating between benign and critical overuse of server resources. In doing so, we presented the DTB traffic model (deficit token bucket) and the RFQ scheduling algorithm, that provide a more relaxed characterization of client behavior. In this model a misbehaving client can be forgiven for exceeding its contracted limit on service requests, if the system determines that it will not affect the guarantees made to any well-behaved client. This flexibility allows clients to use spare capacity without fear of penalty, increasing system utilization and providing greater scheduling flexibility. We provided a formal model to characterize this behavior analytically. We also show its superior properties over existing schemes empirically in [6] and in the full version of this paper.

Acknowledgements

This research was done while the first author was a student at Rice University. The support of this research by the National Science Foundation under NSF Grant CNS-0541369 is gratefully acknowledged.

References

1. VMware, Inc.: Introduction to VMware Infrastructure. (2007) <http://www.vmware.com/support/pubs/>.
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2003) 164–177
3. Waldspurger, C.: Memory resource management in vmware esx server (2002)
4. Sariowan, H., Cruz, R.L., Polyzos, G.C.: Scheduling for quality of service guarantees via service curves. In: Proceedings of the International Conference on Computer Communications and Networks. (1995) 512–520
5. Cruz, R.L.: Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications* **13**(6) (1995) 1048–1056
6. Gulati, A., Merchant, A., Varman, P.: *p*Clock: An arrival curve based approach for QoS guarantees in shared storage systems. In: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press (2007) 13–24
7. Evans, J., Filsfils, C.: Deploying IP and MPLS QoS for multiservice networks. Morgan Kaufman (2007)
8. Bennett, J.C.R., Zhang, H.: WF^2Q : Worst-case fair weighted fair queueing. In: INFOCOM (1). (1996) 120–128
9. Parekh, A.K., Gallager, R.G.: A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.* **1**(3) (1993) 344–357
10. Zhang, L.: VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst.* **9**(2) 101–124
11. Demers, A., Keshav, S., Shenker, S.: Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience* **1**(1) (September 1990) 3–26
12. Greenberg, A.G., Madras, N.: How fair is fair queuing. *J. ACM* **39**(3) (1992) 568–598
13. Goyal, P., Vin, H.M., Cheng, H.: Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin (January 1996)
14. Golestani, S.: A self-clocked fair queueing scheme for broadband applications. In: INFOCOMM'94. (April 1994) 636–646
15. Suri, S., Varghese, G., Chandramenon, G.: Leap forward virtual clock: A new fair queueing scheme with guaranteed delay and throughput fairness. In: INFOCOMM'97. (April 1997)
16. Stiliadis, D., Varma, A.: Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking* **6**(5) (1998) 611–624
17. Shi, H., Sethu, H.: Greedy fair queueing: A goal-oriented strategy for fair real-time packet scheduling. In: RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (2003) 345
18. Kanhere, S.S., Sethu, H., Parekh, A.B.: Fair and efficient packet scheduling using elastic round robin. *IEEE Trans. Parallel Distrib. Syst.* **13**(3) (2002) 324–336
19. Cobb, J.A., Gouda, M.G., El-Nahas, A.: Time-shift scheduling—fair scheduling of flows in high-speed networks. *IEEE/ACM Trans. Netw.* **6**(3) (1998) 274–285
20. Stoica, I., Zhang, H., Ng, T.S.E.: A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.* **8**(2) (2000) 185–199
21. Ng, T.S.E., Stephens, D.C., Stoica, I., Zhang, H.: Supporting best-effort traffic with fair service curve. In: Measurement and Modeling of Computer Systems. (1999) 218–219