A New Approach to Implement Proportional Share Resource Allocation

Ion Stoica, Hussein Abdel-Wahab

Department of Computer Science Old Dominion University Norfolk, Virginia, 23529-0162

Technical Report 95-05

e-mail: {stoica, wahab}@cs.odu.edu

Abstract

We describe a new approach to implement proportional share resource allocation and to provide different levels of service quality. We consider multiple clients that compete for a time-shared resource, and we associate to each client a certain amount of funds. At the beginning of every timeslice a client is selected and it is granted to use the resource during that time-slice. For selecting a client, we use a new *deterministic* scheme that allocates time-slices to every client in *proportion* to its funds. More precisely, we prove that our scheme ensures that out of m consecutive time-slices that are allocated to all clients, the difference between the actual number of time-slices allocated to a client and the expected one is at most proportional to $\log m$. The algorithm is extended in order to support different levels of service quality; at the highest level it guarantees that a client receives a share of the resource that is both superior and inferior bounded.

1 Introduction

One of the most challenging problems in modern operating systems is to design flexible and accurate algorithms to allocate resources among competing clients. This issue has become more important with the emergence of new types of applications, such as multimedia and real-time, that need guaranteed performances. To achieve this, the underlying operating system has to allocate sufficient resources to each application. For example, in order to display video images, an application needs enough I/O bandwidth to transfer the data, and enough memory and CPU cycles to process and display the frames at the specified rate.

Usually, the computation resources are shared in time (e.g. CPU, communication bandwidth) or/and in space (e.g. memory). In these cases, an allocation algorithm needs to allocate a certain *share* of a resource to each application. In order to guarantee performances, the algorithm has to ensure that the allocated share does not drop under a specified level. This helps to support both applications with predefined service rate objectives (e.g. multimedia) and applications that need to meet some specified deadlines (e.g. real-time).

Recently, Waldspurger and Weihl [15] have proposed a proportional share scheduling algorithm,

called *lottery scheduling*. In their algorithm the resource rights are encapsulated in lottery tickets, and every client has a certain number of tickets. At the beginning of every time-slice a lottery is held and the client with the winning ticket is selected and granted to use the resource. Since the ticket numbers are randomly generated, this scheme ensures that, on the average, a client receives a number of time-slices *proportional* to the fraction of tickets it has. They have shown that this algorithm can be successfully used to allocate various resources, such as processing time, I/O bandwidth, memory and access to locks.

The algorithm we present in this paper can be seen as an alternative to *lottery scheduling*. One of the main differences between the two algorithms is the way in which the winning numbers are generated; while in the lottery scheduling algorithm the numbers are generated randomly, we use a *deterministic* scheme that is both efficient and attains better accuracy than the randomized scheme. Namely, we will prove that out of *m consecutive* time-slices that are allocated to all clients, the difference between the number of time-slices a client actually receives and the expected number is at most proportional to $\log m$ within a factor less than 2. This compares favorably to the binomial distribution scheme used in the lottery scheduling algorithm, for which the standard deviation is \sqrt{m} [13].

In different contexts many other schedulers were proposed to achieve proportional share allocation. A large class of such schedulers is based on *priority* schemes [7, 5]. This was a natural approach since almost all existing operating systems rely on the concept of *priority* to allocate processing time to competing processes [11]. In this scheme a process with a higher priority has absolute precedence over any process with lower priority. Therefore a simple extension was to dynamically change the process priorities in order to achieve certain service rate objectives. One of the best-known schemes is *decay usage scheduling* [5] that tries to ensure the *fairness* by increasing the priority of the processes that have received few time-slices in the recent past. This mechanism was successfully implemented in many operating systems, such as Unix BSD [8] and System V [1]. The main drawback of this approach is that it does not offer a very good control of the resource *share* allocation over short periods of time.

At a higher level, fair share schedulers were developed to address the problem of fair resource allocations for a large system shared by multiple users [7]. In these schemes every user receives a certain amount of funds that is proportional to the share of the system resources allocated to it. The fairness is achieved by dynamically changing the priorities based on the relationship between the user consumption and the user share. As decay usage scheduling, this scheme offers a crude control of the allocated share. Moreover, the algorithm introduces a high overhead that limits the usage of this scheme to only large grained applications.

A different approach was proposed by *microeconomic* schedulers [9, 10, 14]. These schedulers use the auction mechanism to allocate resources among the competing clients. At the beginning of every timeslice, the resource initiates an auction at which the interested clients participate by bidding monetary funds that increase over time. The client that offers the highest bid is awarded and therefore acquires the resource for the next time-slice. The price per time-slice for acquiring a resource is directly related to the level of competition for that resource; if the competition increases, the price also increases. In this way, as in real *economic* environments, the clients are encouraged to maximize their profit, i.e. to devote their funds to resource sthat are more important for them. Although, this scheduling scheme successfully solves the resource allocation problem in *distributed* environments they are too complex to efficiently implement fine-grained resource control.

This paper is organized as follows. The next section presents the algorithm in detail, and discusses several of the possible extensions and some implementation issues. Section 3 proves some upper bounds for the algorithm. In Section 4 we discuss simulation results and finally, our conclusions are contained in Section 5.

2 The Algorithm

Let us consider *n* clients that share a common resource *R*. Throughout this paper the notion of client refers to any entity that needs to share one or more computational resources in order to achieve its objectives. Examples of such clients are: threads, processes, applications, users, or group of users. We assume that resources can be shared either in time (e.g. processor, communication bandwidth) or space (e.g. memory, disk storage). For the sake of simplicity we will restrict our discussion to the *time-shared* resources. In this case, time is assumed to be divided into intervals, called *time-slices*. Associated with each resource there is a *resource manager* that, at the beginning of every time-slice runs a scheduling algorithm in order to select one of the competing clients. Once selected, the client is granted to use the resource until the time-slice expires or until the client releases it voluntarily (e.g. it waits for an I/O operation to be performed). We assume that every client *i* has a certain amount of funds x_i , expressed in monetary units, which are used to acquire the resource. Let X denote the total amount of funds of all clients competing for the resource, i.e., $X = \sum_{i=0}^{n-1} x_i$. Our objective is to allocate the resource to every client in proportion to its funds. More precisely, we would like to ensure that that out of any *m* consecutive time-slices that were allocated to all clients, the client *i* to receive

$$m\frac{x_i}{X} \tag{1}$$

time-slices.

Let $L = \{x_0, x_2, \ldots, x_{n-1}\}^{-1}$ be the list containing all clients. In order to simplify the presentation and fix the ideas, first we make the following restrictive assumptions (these assumptions will be relaxed later):

- 1. All the clients arrive in the list L at the same time, and there are no other clients that leave or join the list before all clients complete.
- 2. A time-slice costs *exactly* one monetary unit.
- 3. Every client *i* needs exactly x_i time-slices to complete.

Notice that, according to assumption 2, the cost per time-slice is independent of the number of clients that compete for the resource. Also notice that the funds x_i attributed to every client *i* has three different meanings: first, it is proportional to the resource share that the client should receive (Formula 1); second, it represents the maximum number of time-slices that the client could buy (Assumption 2) and third, it represents the number of time-slices required by the client in order to complete (Assumption 3). Though for practical applications these assumptions are too restrictive, they are used as a starting point in developing the algorithm (later, they will be gradually relaxed).

Similarly to the lottery scheduling algorithm [15], the algorithm is based on the following simple idea: at the beginning of every time-slice a number t is generated and a client i is selected according to the following condition:

$$\sum_{j=0}^{i-1} x_j \le t < \sum_{j=0}^{i} x_j.$$
(2)

But, unlike the lottery scheduling algorithm in which t is a random generated number between 0 and X - 1, we use a deterministic scheme that generates all the numbers between 0 and $2^k - 1$, where $k = \lceil \log X \rceil$. More precisely, let l be an iterator taking all the values between 0 and $2^k - 1$ and let t be the number generated at the l^{th} iteration. If $t \leq X - 1$, then a client is selected according to Equation

¹For simplicity we assume that x_i also identifies the client *i* in the list *L*.



Figure 1: The time-slice allocation for a list consisting of two clients (client 0 has 4 monetary-units and client 1 has 2 monetary-units). The first column in the table represents the iteration index l, the second one contains the generated number rev(l), and the third one indicates the selected client.

2 and it is scheduled to use the resource for the next time-slice. On the other hand, if t > X - 1, no client is selected and the algorithm continues with the next iteration, until a number t less than X is generated (our scheme ensures that no more than one additional new iteration is required to generate such a number). Let $l = \sum_{i=0}^{k-1} b_i 2^i$ be the binary representation of l. Then, at the l^{th} iteration our scheme generates a number t that is the reverse binary representation of l (denoted rev(l)):

$$t = rev(l) = b_0 2^{k-1} + b_1 2^{k-2} + \ldots + b_{k-1}.$$
(3)

As an example, consider a list consisting of two clients that have the initial funds $x_0 = 4$ and $x_1 = 2$, respectively (see Figure 1). From here, we obtain X = 6, k = 3. Further, notice that when the iterator l goes from 0 to 7 the following numbers are generated $(l \rightarrow rev(l)): 000 \rightarrow 000, 001 \rightarrow 100, 010 \rightarrow 010, 011 \rightarrow 110, 100 \rightarrow 001, 101 \rightarrow 101, 110 \rightarrow 011, 111 \rightarrow 111$. The client selected at each iteration is shown in the last column of the table in Figure 1. Notice that at the 3^{rd} and at 7^{th} iterations no client is selected since the generated numbers, 6 and 7, are greater than X - 1 = 5. Here, it is important to make the distinction between the iteration index and the index of the time-slice that is actually allocated. As we have noted, if a number t greater than X - 1 is generated then it is ignored and the algorithm continues until a number less than X is generated. We can show that no more than two consecutive iterations are needed for selecting a client. First, notice that $2^{k-1} < X < 2^k$. Since the least significant bit of the iterator l is the most significant bit of rev(l) (see Equation 3), it is clear that all even iterations generate a number $\leq 2^{k-1}$ and thus select a client in the list. Therefore if the current iteration l_1 does not select any client, then l_1 must be an odd iteration. Consequently, the next iteration $l_1 + 1$ is an even one, and a client is selected.

As we will prove in Section 3, the algorithm guarantees that out of m consecutive time-slices that were allocated to all clients in L, the difference between the number of time-slices any client receives and the expected one (given by Formula 1) is at most proportional to $\log m$. This compares favorably to the binomial distribution scheme used in lottery scheduling algorithm for which the standard deviation is proportional to \sqrt{m} .

Next, to put things in perspective, we try to explain what is the intuition behind the algorithm. Recall that a client *i* is selected whenever the generated number *t* satisfies Equation 2. Next, if we associate with every client *i* the half-open interval $[a_i, a_{i+1})$, where $a_0 = 0$ and $a_i = \sum_{j=1}^{i-1} x_j$ $(1 \le i \le n)$, then the selection condition can be reformulated as follows: a client *i* is selected if the generated number *t* belongs to the interval $[a_i, a_{i+1})$. Therefore the number of time-slices a client receives is equal to the number of values generated in the corresponding interval. But, according to Formula 1, every client should receive a number of time-slices proportional to its funds (which is exactly the size of the interval). A straightforward strategy would be to generate a uniformly distributed sequence over the interval [0, X - 1). This ensures that a client *i* will receive a number of time-slices that is no larger than $[x_i/d]$ and no smaller than $\lfloor x_i/d \rfloor$, where d is the distance² between the generated numbers (see Lemma 1 for details). Thus, if we are given m time-slices, we can distribute them easily among the competing clients such that every client receives approximately its share. This solution is good for a specified value of m, but unfortunately it cannot be generalized for any value of m (which is the case of our problem). The key observation is that a uniform distributed sequence maximizes the minimum distance between any two neighbor³ numbers in the sequence. Conversely, it is easy to see that given a sequence of m numbers, the problem of finding a sequence that maximizes the minimum distance between any two neighbors in the sequence has as solution a uniformly distributed sequence ⁴. The algorithm is mainly based on this idea; a new number is generated in such a way that the minimum distance between it and the previous generated numbers is maximized. This can be easily verified for the first generated numbers: the distance between the first two is 2^{k-1} , the minimum distance between the first four is 2^{k-2} , etc. Generally, it can be shown that after m iterations the minimum distance between any two generated numbers is $2^{k-\lceil\log m\rceil}$.

2.1 Late Join and Early Completion

Recall that in the previous section we have made three assumptions: (1) all clients enter the competion at the same time, (2) every time-slice costs exactly one monetary unit, and (3) every client completes exactly after it spends all its funds. Clearly, these restrictions are not acceptable in a real system. For example, in a dynamic environment we cannot impose that all clients should start at the same time, or a client should terminate only after it has spent all of its funds. Therefore, in this section we relax the first and the third assumptions. More precisely, we allow a client to enter in competition any time (*late join*) and finish its computation before spending all its funds (*early completion*). In this case, x_i no longer represents the number of time-slices that client *i* needs to complete. Instead it represents only the maximum number of time-slices the client can buy (at this point assumption 2 is still valid). Therefore, it would be better to interpret x_i as an *expense account* from which the resource manager withdraws a monetary unit whenever it allocates a time-slice to the client.

Again, let $L = \{x_0, x_1, \ldots, x_{n-1}\}$ be the list of funds of clients that compete for the resource R. For every client *i*, besides the variable x_i we introduce a new variable y_i that counts the number of time-slices that were allocated to client *i* (y_i is incremented, whenever client *i* is selected). Since every time-slice costs exactly one monetary unit, y_i represents the funds the client has already spent for buying time-slices. Now assume a new client *i'* with $x_{i'}$ funds wants to enter in competition for the resource R. At this time every other client *i* in L has already spent y_i from its expense account and therefore it has $x_i - y_i$ monetary units left. Following are the steps performed by the algorithm when a new client joins the list:

- 1. Insert client i' in the list L.
- 2. Update the expense account of every client *i* in *L*: $x_i = x_i y_i$ and $y_i = 0$. Notice that since $y_{i'} = 0$, the $x_{i'}$ is not modified by this operation.
- 3. Recompute the total funds X, the number of bits k of X, and re-initialize the iterator l: $X = \sum_{i=0}^{n} x_i, k = \lfloor \log X \rfloor, l = 0.$

²The distance between two integers a, b is defined as |a - b|.

³Two numbers in the sequence S are said to be *neighbors* if there is no other number in S that has the value between them. For example, in the sequence S = (1, 8, 2, 6), 6 has two neighbors: 2 and 8, while 1 has only one neighbor: 2.

⁴As a simple example, consider the sequence (2, x, 10), such that $2 \le x \le 10$. Then, the value of x that maximizes the minimum distance between any two neighbors in the sequence is clearly x = 6; in this case the minimum distance is 4 = min(6 - 2, 10 - 6).

We can group together all the update operations in steps 2 and 3 (i.e. updates of x_i , y_i , X, k and l) in one procedure, called *restart* procedure. Now, the *late join* algorithm can be easily described as follows: the new client is first inserted in the list and next the restart procedure is executed.

As an example, suppose $L = \{4, 2\}$ (see Figure 1) and consider that at the beginning of the iteration 3 a new client with 7 monetary units enters in competition for the resource. Since the first client from L has already received 2 time-slices and the second 1 time-slice, we have $y_0 = 2$, $y_1 = 1$, and therefore $x_0 = 4 - 2 = 2$, $x_1 = 2 - 1 = 1$. Thus, after the restart procedure is executed we obtain $L = \{2, 1, 7\}$, X = 10, k = 4 and l = 0.

Conversely, suppose a client finishes before spending all its funds. In this case we proceed similarly: first the client is removed from the list and next the restart procedure is called.

2.2 Bounded and Unbounded-Share Services

The major limitation of the above algorithm is that a single parameter, x_i , has still two meanings: it accounts both for the funds in the client's expense account, and for the share of the resource allocated to that client (Formula 1). In other words, it is not possible for a client to ask for a share other than x_i/X . This policy may be too inflexible in practice. To see why, let us consider two clients; one that has 10 monetary units and needs 10 time-slices to complete, and the other that has 100 monetary units and requires 100 time-slices. Then, according to the previous algorithm the second client obtains a share 10 times greater than the other, and both clients will complete at the same time. Intuitively, one can argue that this is not fair; since both clients pay one monetary unit per time-slice, he or she would expect that both clients would have the same share when they compete and therefore the first client would finish 5.5 times faster⁵ (i.e., it only takes approximately 20 time-slices instead of 110 time-slices).

To extend the algorithm, we introduce two new variables: \bar{x}_i that represents the share of the resource the client *i* should receive, and p_i the price per time-slice payed by client *i*. In this way, we eliminate the last restriction of the algorithm (i.e., a time-slice costs exactly one monetary-unit).

Let \bar{X} be the sum over the shares of all clients, i.e., $\bar{X} = \sum_{i=0}^{n-1} \bar{x}_i$, and let \bar{L} be the list $\{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{n-1}\}$. Then, we simply modify the algorithm as follows: first, when the client is selected, the resource manager considers the list \bar{L} (instead of L); second, for every time-slice that the client i receives it is charged with p_i .

The algorithm can be slightly improved if we chose \bar{X} to be a power of two, i.e. $\bar{X} = 2^{\bar{k}}$. In this case, since all the generated numbers are $< \bar{X}$, the algorithm selects a client at each iteration (provided the list is not empty). Also the iterator l may be incremented modulo $2^{\bar{k}}$ (there is no need to re-initialize it when a client enters or leaves the list).

These modifications allow us to easily support different levels of service quality, as defined below:

- share-bounded. A share-bounded service guarantees that the share of the resource allocated to a client never drops, or exceeds some specified bound l_i , and u_i^6 , respectively, i.e., $l_i \leq \bar{x}_i \leq u_i$. If $l_i = 0$ we say that the service is *inferior* unbounded; similarly, if $u_i = \infty$, we say that the service is *superior* unbounded.
- share-unbounded. A service is shared-unbounded if it is both inferior and superior unbounded.

 $^{^{5}}$ Notice that this objective can be accomplished using the previous algorithm if the first client declares that it has 100 monetary-units, instead of 10 monetary-units. In this case, it will finish 5.5 times faster than the other and will spend exactly 10 monetary-units.

⁶The need for a superior bounded service may not be intuitively clear. A situation in which one can take advantage of this service would be in designing a simple communication protocol that avoids the buffer overflow at the receiver by imposing an upper bound for the bandwidth allocated to it.

Notice that although these services only guarantee bounds for the share allocated to every client $i(x_i/X)$, this is enough to guarantee bounds for the number of time-slices the client i actually receives, since as we will prove in Section 3, the difference between this number and the expected number (mx_i/X) is also bounded.

Whenever a new client wants to compete for a resource, it must negotiate its share with the resource manager. Based on a function of the resource utilization and the client requests, the resource manager establishes a price per time-slice. For simplicity we assume that the price per time-slice does not vary during the client life (this would be similar to a rent based mechanism). Next, let \bar{X}_l be the sum over the lower bounds and \bar{X}_u be the sum over all finite upper bounds (not including upper bounds equal to ∞) of all clients in L. Since the resource manger guarantees that no client will have a share less than its lower bound, we clearly have $\bar{X}_l \leq \bar{X}$. Therefore when a new client enters in competition for the resource, it can compete only for the remaining share, i.e. $\bar{X} - \bar{X}_l$. Obviously, if the client requests a lower bound greater than the available share, then it is refused and the client will be asked to reduced its lower bound or to wait until enough share is available. If a new client does not specify a lower bound (i.e., it is inferior unbounded) then it is accepted immediately. When a time-slice is allocated to client *i* then it is charged with p_i . In this paper, we will not discuss further the negotiation mechanism and the price setting policies as they are topics of future research.

Let N_s denote the number of clients that are superior unbounded. Then, in the general case, the shares of the resource are allocated according to the following rules:

- 1. If $\bar{X}_u \leq \bar{X}$ and $N_s = 0$, then every client *i* receives a share equal to u_i . Notice that in this case the share $(X \bar{X}_u)/\bar{X}$ of the resource is not utilized.
- If \$\bar{X}_u > \bar{X}\$ and \$N_s = 0\$, then every client \$i\$ receives a share equal to \$\bar{l}_i + x'_i\$, where \$x'_i = u_i(\bar{X} \bar{X}_l)/\bar{X}_u\$. Thus, in this case the share is allocated in two rounds. In the first round every client \$i\$ receives a share equal to its lower bound \$l_i\$, and in the second the remaining un-allocated share \$\bar{X} \bar{X}_l\$ is proportionally divided among all the clients.
- 3. If $N_s > 0$, then for any superior unbounded client *i*, take $u_i = x_i$. Next, if the new computed $\bar{X}_u \leq \bar{X}$, then apply rule 1; otherwise apply rule 2.

It is easy to verify that the above rules guarantee share-bounded services. Although different allocation policies might be devised, we will not study them in this paper.

2.3 Implementation Issues

Since the scheduling algorithm is invoked at the beginning of every time-slice, its performance is critical. In this section we briefly discuss some of the implementation issues.

First, the generation scheme can be easily implemented using only "right shift with carry" and "left shift with carry" processor instructions. In this way a number can be generated using 2k such operations, where k is the number of bits in its binary representation. Since at most 2 numbers must be generated before a time-slice is allocated, clearly at most 4k shift operations are required.

As long as the client list remains unchanged, the most expensive operation is to search through the list for the winning client. If the number of clients is small, then the search operation can be efficiently implemented by using a simple linked list. On the other hand, if the number of clients is large, then a *interval tree* data structure [3] that supports a logarithmic time search operation, is better suited.

When a new client joins or leaves the list, the restart procedure is invoked. Since this procedure needs to scan all the list in order to recompute the funds and update the expense accounts for every client, it is clearly linear in time. Notice that if much complex negotiation mechanisms are considered, the time-complexity may increase.

3 Algorithm Analysis

In this section we prove some upper bounds for the difference between the expected number of timeslices a client has to receive and the actual number of time-slices it receives. More precisely, consider a static list containing n clients, i.e. $L = \{x_0, x_1, \ldots, x_{n-1}\}$. Then, the main result of this section can be stated as follows: Out of m consecutive time-slices allocated to the clients in L, the difference between the number of time-slices the client i has to receive (i.e., mx_i/X) and the number of time-slices allocated to it is at most $(1 + x_i/X)(\lceil \log m \rceil + 3)$ (Theorem 1). Further, for the first m time-slices that are allocated, we show that the bound can be improved to $(1 + x_i/X)(\lceil \log m \rceil + 3)/2$ (Lemma 4). Finally, we show that after $2^{\lceil \log X \rceil}$ iterations, every client in L receives exactly the number of time-slices it is supposed to receive (Lemma 6). From here we suggest an improvement in the algorithm in order to guarantee better bounds.

Let us consider the partition of the interval I = [0, X), induced by Equation 2, in *n* half-open intervals $I_i = [a_i, a_{i+1}), 0 \le i < n$, where $a_0 = 0$ and $a_i = \sum_{j=0}^{i-1} x_j$ $(1 \le i \le n)$. Now, the selection procedure can be restated as follow: the client *i* is selected at iteration *l* if and only if the generated number rev(l) is contained in the interval I_i .

Next, let $S = Seq(a, b) = \{t \mid t = rev(l); a \leq l < b\}$ be the sequence generated by the algorithm between a^{th} and b^{th} iterations and, for any interval⁷ J, let c(J, S) be the number of elements from Scontained in J. It is easy to see that if S_1 and S_2 are two disjoint sequences, then $c(J, S_1 \cup S_2) = c(J, S_1) + c(J, S_2)$. Notice that the total number of time-slices received by all clients in list L can be expressed as c(I, S), and the number of time-slices received by a client i as $c(I_i, S)$. Now, the difference between the number of time-slices a client i receives and the expected number can be written as

$$|c(I_i,S) - c(I,S)\frac{x_i}{X}|.$$
(4)

Throughout the remaining of this section we use the following notations: |S| denotes the size of the sequence S (i.e., the number of elements in S), and |J| denotes the length of the interval J. Also, as before, $k = \lfloor \log X \rfloor$ denotes the number of bits in the binary representation of X.

The next lemma determines the lower and upper bounds for c(J,S) in the special case in which the elements in S are uniformly distributed between 0 and 2^k , and the distance between any two neighbors is 2^{k-k_1} .

Lemma 1 Let $S = Seq(p2^{k_1}, (p+1)2^{k_1})$, such that $0 \le k_1 < k$ and $0 \le p < 2^{k-k_1}$. Then for any interval $J \subset [0, 2^k)$ we have

$$\lfloor \frac{|J|}{2^{k-k_1}} \rfloor \le c(J,S) \le \lceil \frac{|J|}{2^{k-k_1}} \rceil.$$
(5)

Proof. Let $l = \sum_{i=0}^{k-1} b_i 2^i$ be the binary representation of l. Clearly, the first $k - k_1$ most significant bits of l $(p2^{k_1} \leq l < (p+1)2^{k_1})$ are identical, while the last k_1 bits take all the possible values between 0 and $2^{k_1} - 1$. Since from Equation 3 we have $rev(l) = \sum_{i=0}^{k-1} b_i 2^{k-i-1}$, the first k_1 most significant bits of rev(l) takes all possible values between 0 and $2^{k_1} - 1$, while the last $k - k_1$ bits are identical. It is easy to see that the elements of S are uniformly distributed in the interval $[0, 2^k)$ at the distance 2^{k-k_1} each from the other. From here, the Inequality 5 follows.

In the following two lemmas we derive bounds ⁸ for c(J, S), when the sequence S is either of form $Seq(p2^{k_1}, p2^{k_1} + r)$ (Lemma 2), or of form $Seq(p2^{k_1} - r, p2^{k_1})$ (Lemma 3), where $0 < r < 2^{k_1}$.

⁷Throughout this section all intervals have integer limits.

⁸To simplify the presentation the bounds we derive in these lemmas are independent of |J|. By taking into account the length of the interval J we can prove tighter bounds. Mainly, it can be shown that the term $\lceil \log |S| \rceil$ would be replace by the term $max(\lceil \log |S| \rceil + \lceil \log |J| \rceil - \lceil \log |X| \rceil, 0) + 1$.

Lemma 2 Let $S = Seq(p2^{k_1}, p2^{k_1} + r)$, where $0 \le k_1 < k$, $0 \le p < 2^{k-k_1}$ and $0 < r < 2^{k_1}$. Then for any interval $J \subset [0, 2^k)$ we have

$$|c(J,S) - |S|\frac{|J|}{2^k}| < \frac{\lceil \log |S| \rceil}{2} + 1.$$

Proof. To keep the notation simple we prove the result for p = 0 (for p > 0 the proof is identical). Let $k_r = \lceil \log |S| \rceil = \lceil \log r \rceil$, and let $r = \sum_{i=0}^{k_r-1} b_i 2^i$. Next, consider the monotonically increasing sequence of integers defined by the recurrence: $s_0 = 0$ and $s_i = s_{i-1} + b_{k_r-i} 2^{k_r-i}$ ($0 < i \leq k_r$). For example, if k = 6, r = 010110 ($k_r = 5$), then we have $s_0 = 000000$, $s_1 = s_2 = 010000$, $s_3 = 010100$ and $s_4 = s_5 = 010110$. Now, consider the family of sequences $S_i = Seq(s_i, s_{i+1})$ ($0 \leq i < k_r$), where clearly $S = S_0 \cup S_1 \cup \ldots \cup S_{k_r-1}$. Since the last $k_r - i$ bits of s_i are zero and $S_i = Seq(s_i, s_i + b_{k_r-i-1}2^{k_r-i-1})$, by using Lemma 1 we can write

$$b_j \lfloor \frac{|J|}{2^{k-j}} \rfloor \le c(J, S_i) \le b_j \lceil \frac{|J|}{2^{k-j}} \rceil.$$

where $j = k_r - i - 1$. Notice that if $b_j = 1$, the above equation reduces to Equation 5, otherwise $c(J, S_i) = 0$ since $s_i = s_{i+1}$. Because S is the union of all sequences S_i , by summation, we obtain:

$$\sum_{i=0}^{k_r-1} b_i \lfloor \frac{|J|}{2^{k-i}} \rfloor \le c(J,S) \le \sum_{i=0}^{k_r-i} b_i \lceil \frac{|J|}{2^{k-i}} \rceil.$$
(6)

Let ones(r) be the number of ones in the binary representation of r. Using the well-known inequalities: [x] < x + 1 and |x| > x - 1, the Inequality 6 become

$$r\frac{|J|}{2^{k}} - ones(r) < c(J,S) < r\frac{|J|}{2^{k}} + ones(r).$$
(7)

Next, we derive alternative bounds for c(J,S). First, we compute c(J,U), where $U = Seq(r, 2^{k_r+1})$ (notice that $Seq(0, 2^{k_r+1}) = S \cup U$). Let $s = 2^{k_r+1} - r = \sum_{i=0}^{k_r-1} b'_i 2^i$ and consider the monotonically decreasing sequence given by the recurrence: $u_0 = 2^{k_r+1}$, $u_i = u_{i-1} - b'_{k_r-i} 2^{k_r-i}$ ($0 < i \leq k_r$). As an example, for r = 0.0110, we have s = 0.01010, and $u_0 = u_1 = 1.00000$, $u_2 = u_3 = 0.00000$, $u_4 = u_5 = 0.010110$. Next, define the disjoint sequences $U_i = Seq(u_{i+1}, u_i)$ ($0 \leq i < k_r$). From here, proceeding similarly to the previous case, we obtain

$$s\frac{|J|}{2^{k}} - ones(s) < c(J, U) < s\frac{|J|}{2^{k}} + ones(s).$$
(8)

From Lemma 1 we have $\lceil |J|/2^{k-k_r-1} \rceil \leq c(J, S \cup U) \leq \lfloor |J|/2^{k-k_r-1} \rfloor$. Combining this relation with Inequality 7 and using the equality $c(J, S) = c(J, S \cup U) - c(J, U)$, after some simple algebra we obtain

$$r\frac{|J|}{2^k} - ones(s) - 1 < c(J,S) < r\frac{|J|}{2^k} + ones(s) + 1.$$
(9)

Let zeros(r) denotes the number of zeros in the binary representation of r. Then it can be shown⁹ that $ones(s) \leq zeros(r) + 1$ and therefore the Inequality 9 become

$$r\frac{|J|}{2^k} - zeros(r) - 2 < c(J,S) < r\frac{|J|}{2^k} + zeros(r) + 2.$$
(10)

Now, by combining Inequalities 7 and 10 we obtain

$$r\frac{|J|}{2^{k}} - \min(zeros(r), ones(r) + 2) < c(J,S) < r\frac{|J|}{2^{k}} + \min(zeros(r), ones(r) + 2).$$

Finally, since $zeros(n) + ones(r) = k_r$ and $min(x, y) \le (x+y)/2$, we obtain $min(zeros(r), ones(r)+2) \le k_r/2 + 1 = \lceil \log r \rceil/2 + 1$ which completes our proof. \Box

The next lemma is given without proof (the proof is very similar to the one of Lemma 2).

⁹Assuming that r is represented on only $k_r + 1$ bits, then $s = 2^{k_r+1} - r$ can be interpreted as the 2-complement of r. Therefore s can be obtained by complementing all bits of r that are to the left of the least significant (rightmost) bit of value 1 ([12], pp. 203).

Lemma 3 Let $S = Seq(p2^{k_1} - r, p2^{k_1})$, where $0 \le k_1 < k$, $1 \le p < 2^{k-k_1}$ and $0 < r < 2^{k_1}$. Then for any interval $J \subset [0, 2^k)$ we have

$$|c(J,S) - |S| \frac{|J|}{2^k}| < \frac{\lceil \log |S| \rceil}{2} + 1.$$

Next we derive the upper bounds for the Expression 4 in the case of the particular sequences $S = Seq(p2^{k_1}, p2^{k_1} + r)$ (Lemma 4) and $S = Seq(p2^{k_1} - r, p2^{k_1})$ (Lemma 5), where $0 < r < 2^{k_1}$.

Lemma 4 Let $S = Seq(p2^{k_1}, p2^{k_1} + r)$, where $0 \le k_1 < k$, $0 \le p < 2^{k-k_1}$, $0 < r < 2^{k_1}$, and let m = c(I, S). Then for any interval $I_i \subset I$ we have

$$|c(I_i, S) - m\frac{x_i}{X}| < (1 + \frac{x_i}{X})\frac{\lceil \log m \rceil + 3}{2}$$

Proof. First denote $\alpha = |S|/2^k$ and $\beta = (\lceil \log |S| \rceil)/2 + 1$. Next recall that $|I_i| = x_i$, |I| = X, $I_i, I \subset [0, 2^k)$ and therefore by Lemma 3 we have

$$\alpha X - \beta < c(I,S) < \alpha X + \beta \tag{11}$$

$$\alpha x_i - \beta < c(I_i, S) < \alpha x_i + \beta \tag{12}$$

By multiplying (11) with x_i/X and combining it with (12) we obtain

$$-\beta(1+\frac{x_i}{X}) < c(I_i, S) - c(I, S)\frac{x_i}{X} < \beta(1+\frac{x_i}{X}).$$

Notice that the interval $[p2^{k_1}, p2^{k_1} + r)$ contains at least $\lceil |S|/2 \rceil$ even integers and since $|I| = X > 2^{k-1}$ we obtain $m = c(I, S) \ge \lceil |S|/2 \rceil \Rightarrow |S| \le 2m$. Finally, we have $\beta = \lceil \log |S| \rceil/2 + 1 \le (\lceil \log m \rceil + 3)/2$ which completes the proof. \Box

Lemma 5 Let $S = Seq(p2^{k_1} - r, p2^{k_1})$, where $0 \le k_1 < k$, $1 \le p < 2^{k-k_1}$, $0 < r < 2^{k_1}$, and let m = c(I, S). Then for any interval $I_i \subset I$ we have

$$|c(I_i,S) - m\frac{x_i}{X}| < (1 + \frac{x_i}{X})(\frac{\lceil \log m \rceil}{2} + 2).$$

Proof. The proof is similar to the one of Lemma 4. There are only two minor differences. First, to determine the bounds for c(I, S) and $c(I_i, S)$ we use Lemma 3, instead of Lemma 2. Second, notice that the interval $[p2^{k_1} - r, p2^{k_1})$ contains at least $\lfloor |S|/2 \rfloor$ even integers and thus we have $|S| \leq 2m + 1$. Further we can write $\beta = \lceil \log |S| \rceil / 2 + 1 \leq (\lceil \log(2m + 1) \rceil) / 2 + 1 < \lceil \log m \rceil / 2 + 2$ which proves our lemma. \Box

Now we are in position to prove the main results of this section. The next theorem gives the upper bound for the difference between the number of time-slices a client i receives, out of any m consecutive time-slices that were allocated to all clients in list L, and the expected number.

Theorem 1 Let S = Seq(a,b), where $0 \le a < b < 2^k$, and let m = c(I, S). Then for any sub-interval $I_i \subset I$ we have

$$|c(I_i, S) - m\frac{x_i}{X}| < (1 + \frac{x_i}{X})(\lceil \log m \rceil + 3).$$
(13)

Proof Let $k_1 = \lfloor \log b \rfloor$, $r = b - 2^{k_1}$, and $s = 2^{k_1} - a$. Next define the sequences $S_1 = Seq(2^{k_1} - s, 2^{k_1})$ and $S_2 = Seq(2^{k_1}, 2^{k_1} + r)$, where clearly $S = S_1 \cup S_2$. Since $c(J, S) = c(J, S_1) + c(J, S_2)$, by using Lemma 4 and Lemma 5 we obtain

$$(r+s)\frac{|J|}{2^k} - \frac{\lceil \log r \rceil}{2} - \frac{\lceil \log s \rceil}{2} - \frac{7}{2} < c(J,S) < (r+s)\frac{|J|}{2^k} + \frac{\lceil \log r \rceil}{2} + \frac{\lceil \log s \rceil}{2} + \frac{7}{2}.$$

Further we have $\lceil \log r \rceil + \lceil \log s \rceil \leq \lceil \log rs \rceil + 1$. Since the maximum achieved by rs is $(|S|^2)/4$ we obtain $\log rs \leq 2 \log |S| - 2$, and finally we get $\lceil \log rs \rceil \leq \lceil 2 \log |S| - 2 \rceil = \lceil 2 \log |S| \rceil - 2 \leq 2(\lceil \log |S| \rceil - 1)$. From here, Inequality 13 follows directly. \square

Next, we prove that after the first 2^k consecutive iterations every client receives *exactly* the number of time-slices it is supposed to get.

Lemma 6 Let $S = Seq(0, 2^k)$, and let m = c(I, S). Then for any sub-interval $I_i \subset I$ we have $|c(I_i, S) - m\frac{x_i}{X}| = 0.$

Proof. The proof is immediate. First notice that the function $rev: P \to P$, where $P = \{l \in \mathbb{N} \mid 0 \leq l < 2^k\}$, is a bijection. Therefore for any interval $J \subset [0, 2^k)$, c(J, S) is equal to the number of integers it contains. But every interval I_i contains exactly x_i integers and the interval I contains X integers. Thus, we have $c(I_i, S) = x_i$ and c(I, S) = X which completes the proof. \Box

The above lemma suggests a way to improve the algorithm in order to guarantee better bounds. Let y be the greatest common divisor of all x_i $(0 \le i < n)$. Next, consider the list $L' = \{x'_0, x'_1, \ldots, x'_{n-1}\}$ where $x'_i = x_i/y$ $(0 \le i < n)$ and denote $X' = \sum_{i=0}^{n-1} x'_i$ and $k' = \lceil \log X' \rceil$. Then instead of running the algorithm for 2^k iterations over the list L, we can run the algorithm for $y2^{k'}$ iterations over the list L. After each $2^{k'}$ iterations, according to Lemma 4, client *i* receives x'_i time-slices and therefore after $y2^{k'}$ it will receive $x'_iy = x_i$ time-slices. Further it is easy to verify that, in this case, the Equation 17 becomes

$$|c(I_i,S) - m\frac{x_i}{X}| < (1 + \frac{x_i}{X})(\lceil \log(m \mod y) \rceil + 3).$$

4 Experimental Results

To evaluate the algorithm we use simulation. Mainly, we were interested to evaluate the allocation *accuracy*, the algorithm *responsiveness* and the ability to guarantee the bounded-share services.

To measure the allocation accuracy, we compute the maximum difference (error) between the number of time-slices a client actually receives and the number estimated by Formula 1. For this, we consider all the possible lists $L = \{x_0, x_1, \ldots, x_{n-1}\}$, such that $X = \sum_{i=0}^{i=n-1} x_i \leq 128$. Since the number of timeslices received by client *i* is equal to the total number of values generated in the corresponding interval I_i (see Section 3), we have performed this experiment by considering for any interval I = [0, X) all the possible sub-intervals $J \subset I$. The maximum error we found was 3.04 and it occurred for I = [0, 75)and J = [11, 64). In Figure 2.a we plot the measured error versus time (here, a time-slice is assumed to be equal to one time-unit) and the upper bound given by Lemma 4 (we use this relation because we counted the number of time-slices received by the client beginning with iteration 0). In Figure 2.b we show the number of time-slices allocated to the client versus time. Notice that, even in this case, that is the worst case we found, the number of time-slices allocated to the client approximates well the line with the slope 53/75 that represents the expected number of time-slices the client has to receive at any moment in time.

Here, two things worth mentioned. First, the actual measurements are even much better than the upper bound we have derived in Section 3 (that is mainly because various approximations we have used). In fact, in all the other cases the error is *strictly* less than 3. Second, notice that as proved in Lemma 5 once all X time-slices were allocated the error is 0. In other words the client receives exactly the number of time-slices it is supposed to receive, i.e. in our case 53.

Similarly, the Figures 2.c and 2.d depict the error and the allocated time-slices, respectively, for the case when I = [0, 128) and J = [0, 85). This is the *worst* case among all the lists of length 128. Since in this case $X = 128 = 2^7$, the upper bound is given by Lemma 2. The maximum error we have have



Figure 2: Figure a) plots the measured error (filled line) and the estimated upper bound (dashed line) for the case in which the L's corresponding interval is I = [0, 75) and the client's corresponding interval is J = [11, 64). Figure b) shows the number of time-slices allocated to the client versus time, for the same case. Figures c) and d) plots the error and the number of allocated time-slices for the case in which I = [0, 128) and J = [0, 85). In all figures a time-slice is equal to one time-unit.

obtained in this case was only 2.45. Consequently, the number of allocated time-slices approximates much better the ideal line which has the slope 85/128 (see Figure 2.d).

Another important criterion we evaluate is the algorithm responsiveness. We define the responsiveness as the propriety of the algorithm to rapidly adjust the client shares, when a new client arrives or departs the list L. It is easy to see that the algorithm adapts immediately to the changes. This is because every time a change occurs the restart procedure recomputes the share for every client, which determine directly the allocated time-slice rates. For further investigation we have conducted a simple experiment in which we consider a list of two clients $L = \{x_0, x_1\}$, where $x_0 = 400$ and $x_1 = 200$. After the first 200 time-slices were allocated, a new client with $x_3 = 300$ joins the list. At this moment the ratio of the allocated time-slices for the first two clients was 135 : 65 (compared to 400 : 200) and therefore the remaining funds for the first two clients are $x_0 = 265$ and $x_1 = 135$ respectively. Next, assume the last client completes after it has received exactly 100 time-slices (this happens after 431 time-slice were allocated). If we consider only the time-slices allocated during the time interval when all three clients were active we obtain the ratios 87 : 44 : 100 (1 : 0.505 : 1.149), which compares well with the expected ones 265 : 135 : 300 (1 : 0.509 : 1.132). Once the client 2 completes, only the other



Figure 3: The cumulative time-slices allocated to three clients. Client 0, with 400 monetary units, and client 1, with 200 monetary-units, begin to compete at the same time. Client 2, with 300 monetary-units, joins the list after 200 time-slices were allocated to clients 0 and 1, respectively, and terminates after it receives exactly 100 time-slices.

two will remain to compete for the resource. At this point, the remaining funds are: $x_0 = 178$ and $x_1 = 91$. The clients complete roughly at the same time, after a total of 700 time-slices were allocated (i.e. 400 for client 0, 200 for client 1 and 100 for client 2).

Finally, to see how the bounded-share services are ensured, we modify slightly the previous experiment, i.e., we assume that the client 1 has both superior and inferior bounds equal to \bar{x}_1 , where $\bar{x}_1/\bar{X} = 1/3$. In other words, client 1 should receive exactly one third share of the resource. As can be seen in Figure 4, the slope corresponding to client 1 is constant (1/3) and *independent* of the other clients that competes for the resource. Since, in this case, client 2 competes only for the remaining share $\bar{X} - \bar{x}_1$, it takes longer to complete than in previous test, i.e. it finishes after 483 time-slices were allocated, instead of 431. On the other hand, client 1 completes after 598 time-slices were allocated which is very closed to estimated value, i.e. $600 = 3 \cdot 200$. Once the client 1 completes, the client 0 can fully use the resource (notice that the slope increases to 1) and it finishes after exactly 700 time-slice were allocated.

5 Conclusions and Future Work

We have described a new approach to implement proportional-share resource allocation that provides a flexible control and a high degree of accuracy in allocating shared resources among competing clients. Specifically, we have proved that out of any m consecutive time-slices, the difference between the number of time-slices a client i receives and the expected number is less that $(1 + x_i/X)(\lceil \log m \rceil + 3)$, where x_i/X represents the share of the resource allocated to the client i. Moreover, the simulation results show that the actual difference is much smaller. For example, among all the possible cases in which the resource share is allocated in 1/128 increments, we found that the difference was never larger than 2.45 for any first m consecutive time-slices that were allocated (see Figure 2.c).

Further, simple extensions of the algorithm guarantee that every client receives a fraction from the resource that can be inferior and/or superior bounded. This feature is important in supporting both,



Figure 4: The cumulative time-slices allocated to three clients. Client 0, with 400 monetary units, and client 1, with 200 monetary-units, begin to compete at the same time. In addition, client 1 is guaranteed to receive a share of 1/3. Client 2, with 300 monetary-units, joins the list after 200 time-slices were allocated and terminates after it receives exactly 100 time-slices.

multimedia applications that require certain rate objectives and real-time applications that have to meet specified deadlines.

However, many other problems remain open. First, we have to consider the case in which the client does not use the entire time-slice. As suggested in [15], a solution would be to use *compensation* funds, i.e. to give the client additional funds proportional to the average fraction from the time-slices that is not used. An alternative approach would be to charge the client with a fraction of the price that is equal to the the fraction of the time-slice that is actually used.

Second, it will be interesting to consider both hierarchical and heterogeneous scheduling schemes. As an example, consider the case in which the clients are grouped in several classes. Then, at a higher level a proportional scheduler may be used to allocate resource shares to each class, while at a lower level, among the clients in the same class, a simple scheduler (e.g. round-robin) may be used. As a second example, consider the process-thread hierarchy. Here, similarly, a proportional scheduler may be used to allocated CPU time-slices to each process, and in turn, the process may select a thread to run by using a different policy. Ultimately, we think that this may be a valuable approach to implement flexible resource management in the new generations of operating systems [2, 4] in which the applications would be able to implement their own algorithms for managing the shares of the resources allocated to them.

References

- [1] M. J. Bach. "The Design of the Unix Operating System," Prentice-Hall, 1986.
- [2] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage and E. G. Sirer. "SPIN An Extensible Microkerenel for Application-specific Operating System Services," Technical Report 94-03-03, Dept. of Comp. Science and Engineering, University of Washington, Feb. 28, 1994.
- [3] T. H. Cormen, C. E. Leiserson and R. L. Rivest. "Introduction to Algorithms," MIT Press, 1992.

- [4] D. R. Engler, M. F. Kaashoek and J. W. O'Toole Jr. "The Operating System as a Secure Programmable Machine," Proc. of the Sixth SIGOPS European Workshop, 1994.
- [5] J. L. Hellerstein. "Achieving Service Rate Objectives with Decay Usage Scheduling," IEEE Transactions on Software Engineering, Vol. 19, No. 8, August 1993, pp 813-825.
- [6] K. E. Drexler and M. S. Miller. "Incentive Engineering for Computational Resource Management," *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988, pp. 231-266.
- [7] J. Kay and P. Lauder. "A Fair Share Scheduler," Communication of the ACM, Vol. 31, No. 1, January 1988, pp 44-45.
- [8] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman. "The Design and Implementation of the 4.3BSD UNIX Operating System," Addison-Wesley, 1989.
- [9] T. W. Malone, R. E. Fikes, K. R. Grant and M. T. Howard. "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments," *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988, pp. 177-205.
- [10] M. S. Miller and K. E. Drexler. "Markets and Computation: Agoric Open System," The Ecology of Computation, B. Huberman (ed.), North-Holland, 1988, pp. 133-176.
- [11] A. Silberschatz and P. B. Galvin. "Operating Systems Concepts," fourth edition, Addison-Wesley, 1994.
- [12] I. Tomek. "Introduction to Computer Organization," Computer Science Press, 1981.
- [13] R. Jain. "The Art of Computer System Performance Analysis. Techniques for Experimental Design, Measurments, Simulation, and Modeling," John Wiley & Sons, 1991.
- [14] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart and W. S. Stornetta. "Spawn: A Distributed Computational Economy," *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February 1992, pp. 103-117.
- [15] C. A. Waldspurger and W. E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," Proc. of the First Symposium on Operating System Design and Implementation, pp. 1-12, November 1994.