Abstract

Very large size Peer-to-Peer systems are often required to implement efficient and scalable services, but usually they can be built only by assembling resources contributed by many independent users. Among the guarantees that must be provided to convince these users to join the P2P system, particularly important is the ability of ensuring that P2P applications and services run on their nodes will not unacceptably degrade the performance of their own applications because of an excessive resource consumption. In this paper we present *Interceptor*, a middleware-level application segregation and scheduling system that is able to strictly enforce quantitative limitations on node resource usage and, at same time, to make P2P applications achieve satisfactory performance even in face of these limitations. A proof-of-concept implementation has been carried out for the Linux operating system, and has been used to perform an extensive experimentation aimed at quantitatively evaluating *Interceptor*. The results we obtained clearly demonstrate that *Interceptor* is able to strictly enforce quantitative limitations on node resource usage, and at the same time to effectively schedule P2P applications.

1 Introduction

The Peer-to-Peer (P2P) paradigm is being increasingly used at the substrate for the development of highly scalable distributed systems aimed at delivering services of various type, such as efficient data lookup [3], media streaming and transcoding [10], DNS address resolution [11], distributed storage management [25], distributed file systems [13], and anonymous communication [24]. In order to properly work, these systems typically require the availability of a large number of nodes, possibly scattered across different administrative domain and dispersed on a geographic scale. The only viable option to build such very large size P2P systems is the aggregation of nodes belonging to many individual entities (single individuals or institutions). These entities, however, will contribute their own resources to a P2P system only if, besides receiving appealing incentives [16] and assurances concerning the trustworthiness [4] of P2P users and applications that will use their resources, are guaranteed that performance of their own applications will not be unacceptably degraded because of an excessive resource consumption caused by P2P applications. Providing such a guarantee requires the availability of *application segregation* mechanisms able to precisely enforce quantitative limitations on the usage of the various node resources (e.g., CPU cycles, RAM, disk space and bandwidth, etc.) set by the corresponding owner.

While for resource owners segregation mechanisms are of crucial importance, P2P users have a different requirement, namely they require the availability of scheduling mechanisms and policies able to make their applications achieve the maximum possible performance within the resource limits set by the node owners, so that services with a satisfactory quality level can be delivered. These mechanisms and policies must permit the specification and the enforcement of different resource limits for different P2P applications, so that priority can be given to critical services (e.g., DHT routing) with respect to less critical ones. Unfortunately, application segregation and effective scheduling are often conflicting goals, so achieving both of them simultaneously is non trivial and still represents an open research problem.

Application segregation mechanisms are not a new concept, and have been studied in various contexts, but the extreme hardware and software heterogeneity found in P2P systems introduces new requirements and constraints that must be met in order to make them usable in practice. As a matter of fact, segregation mechanisms suitable to P2P systems must not be tied to any specific operating system (OS) or platform, and must not require any modification to existing OSes and applications. Placing restrictions on the supported OSes would indeed result in limitations on the number of nodes that can join the P2P system, while requiring modifications to the OSes and applications would raise natural concerns about security and unacceptably increase software maintenance costs and problems.

In this paper we propose *Interceptor*, an application segregation and scheduling system able to simultaneously meet all the above requirements, that owes its name to its usage of mechanisms able to *intercept* the resource requests issued by individual applications. *Interceptor* allows node owners to specify quantitative constraints on the share globally allocated to P2P applications for the main resources of individual nodes (namely, the CPU, the disk and network bandwidth, and the physical memory), and enforces them by means of a set of resource-limitation mechanisms. Furthermore, it supports the allocation of different resource shares to individual P2P applications (*differentiated allocation*), and encompasses a set of scheduling mechanisms and policies aimed at maximizing P2P applications performance without violating resource limitations.

As will be discussed later, *Interceptor* is designed in such a way that can be implemented entirely at the middleware-level on standard OSes that provide facilities for the interception of the resource requests issued by individual applications (available on many modern OSes, such as Linux, Windows NT and subsequent versions, Solaris, and FreeBSD) and real-time fixed-priority process scheduling (a feature that is present in all POSIX-compliant systems, like are the ones mentioned before). Therefore, no modifications to the local OSes ran on individual P2P nodes is required. Furthermore, *Interceptor* is totally transparent to applications, so no application modifications are required as well.

The remainder of this paper is organized as follows. Section 2 discusses related work, while Section 3 presents the architecture, the mechanisms, and the scheduling policies used by *Interceptor*. Section 4 describes a proof-of-concept implementation we have devised for Linux, and reports the experimental results we obtained using it, that demonstrate the effectiveness of the application segregation mechanisms and of the various resource scheduling algorithms, as well as the fairness of the latter ones. Finally, Section 5 concludes the paper and outlines future research work.

2 Related work

The need of providing application segregation has arisen in various fields, and several solutions have been proposed in the literature. The simplest solution is the one adopted by most public-resource computing systems [1, 8], and consists in using an application-specific "client" program, executed on the hosting resource, that controls the execution of the external application and enforces the desired resource usage limitations. Although this solution is attractive because of its simplicity and ease of implementation, it provides a limited form of segregation, and does not support the execution and the scheduling of *multiple* P2P applications on the same node.

An alternative solution is to resort to a variant of *sandboxing*. More specifically, sandboxing consists in providing applications with a restricted environment in which certain functions are prohibited. Although sandboxing has been conceived to enforce qualitative restrictions (i.e., restrictions on what resources can be used by an application), it can be used to enforce quantitative constraints as well [6, 9]. Sandboxing may be achieved in various ways. The first one consists in intercepting the resource access requests issued by applications [5, 18, 31] and in actually carrying out only the allowed ones. A second approach, that is becoming increasingly popular thanks to the the rediscovery of virtualization technologies [17], consists in using a virtual machine (such as Xen [6] or VMWare [30]), rather than the real one, to execute sandboxed applications. The main problem of traditional sandbox approaches is that they do not support differentiated resource allocation among competing sandboxed applications. More precisely, in these approaches application scheduling is performed by the underlying OS, that more often than not does not incorporate these mechanisms as well. Furthermore, sandboxing lacks generality, as any sandboxing technique is platform specific, and even if the same technique can sometimes be exported on a different platform, the development effort usually is quite significant.

An alternative to sandboxing consists in kernel-level techniques that extend the OS kernel with suitable mechanisms able to enforce quantitative constraints on application resource usage, and to support differentiated resource allocation. Kernel-level techniques, however, require modifications to the kernel running on the various nodes of the P2P system, something that is usually impossible for proprietary or commercial OSes, not to mention the efforts necessary to implement and maintain these modifications. Furthermore, requiring resource owners to run a modified kernel appears unrealistic because of the obvious security concerns.

Middleware-level resource management solutions [14, 26, 27] have been proposed as an alternative to kernel-level techniques. As shown in [28] middleware approaches, despite their potentially high overhead, often result in application performance similar to those attained by kernel-level solutions. However, in general they require to modify or to relink existing application code, since typically each middleware layer requires that processes use a particular API to access system resources and middleware services. Consequently, applications may need to be reprogrammed to use a new API, which is costly and sometimes even impossible (for instance, when source code is not accessible).

Interceptor is instead designed to combine the advantages of all these approaches, while at the same time avoiding their drawbacks. More specifically, it provides effective segregation to P2P applications, and supports differentiated resource allocations among them, but does not require modifications to neither the OS running on individual nodes (since it works at the middleware level) nor to the applications. Furthermore, it relies on mechanisms available on practically all modern OSes, so it can be easily ported on a variety of platforms with a limited development effort. Finally, unlike all the alternatives described above, *Interceptor* provides effective resource scheduling able to maximize

the performance of sandboxed applications without violating the quantitative constraints set by the node owner.

3 The Interceptor system: architecture, mechanisms, and scheduling policies

Interceptor is a middleware system (i.e., it is placed between the local OS and the applications), executed on individual nodes of a P2P system, that intercepts the resource access requests issued by P2P applications, and handles them in such a way that it (a) provides application segregation for P2P applications (i.e., enforce quantitative limitations on P2P applications resource usage), and (b) maximizes their performance without violating the above limitations. *Interceptor* achieves its purposes by using a set of resource scheduling mechanisms and policies that are based on the principle discussed below.

Let $S(P2P, \tau)$ be the maximum share of resource τ (i.e., the fraction of τ's capacity) that can be allocated to P2P applications, and let $S(P_i, \tau)$ the resource share allocated to process P_i (henceforth referred to as the *nominal share* of P_i). Furthermore, let $R(P_i, T, \tau)$ the share of resource τ received by P_i in the interval going from the instant of its creation to time T, which is defined as:

$$R(P_i, T, \tau) = \frac{UT(P_i, \tau)}{T} \tag{1}$$

where $UT(P_i, \tau)$ denotes the amount of time in which P_i has used resource τ (for instance, if process P_i has used the CPU for 10 sec. in an interval lasting $T = 100$ sec., then $R(P_i, CPU) = 10/100 = 0.1 = 10\%$). Finally, let us define the $Lag(P_i, T, \tau)$ of process P_i for resource τ as:

$$Lag(P_i, T, \tau) = R(P_i, T, \tau) - S(P_i, \tau) \tag{2}$$

Intuitively, the lag corresponds to the difference between the nominal share of τ allocated to a given process P_i, and the resource capacity that it actually received in the interval going from the instant of its creation to time T. Thus, a positive lag value indicates that process P_i received a fraction of τ's capacity larger than its nominal share.

Interceptor schedules the usage requests for τ issued by P2P processes in such a way that the following inequalities simultaneously hold at any time instant T:

$$\begin{cases} Lag(P_i, T, \tau) \geq 0, \quad \forall P_i \in P2P & (C_1) \\ \sum_{P_i \in P2P} R(P_i, T, \tau) \leq S(P2P, \tau) & (C_2) \end{cases}$$

where $P2P$ is the set of P2P processes running on the node. In practice, inequality (C_1) states that the lag of each P2P process P_i must be always larger than 0, meaning that P_i must have received, in the interval going from the instant of its creation to time T, at least its nominal share for resource τ, while inequality (C_2) states that the share of τ globally allocated to P2P processes must not exceed the upper limit $S(P2P, \tau)$ on τ's usage set by the node owner. To achieve this goal, *Interceptor* intercepts all the resource usage requests issued by P2P processes, stores them into waiting queues that are kept sorted according to scheduling policies specific for the various resource type (see the next subsections), and forwards them to the proper resource schedulers at specific points in time chosen in such a way that inequalities C_1 and C_2 are always satisfied.

In the rest of this section, we will illustrate the architecture of *Interceptor*, and the mechanisms and scheduling policies it uses to achieve its goals.

3.1 Architecture

The architecture of *Interceptor*, schematically depicted in Fig. 1, encompasses two management modules (the *Creator* and the *Catcher*) and four resource schedulers, one for each of the resource types that it is able to control (namely, the CPU, the disk, the network interface, and the physical memory). The *Creator* represents the interface between the schedulers and the entities (either human users or external applications) that require the execution of P2P processes on the node. When the *Creator* receives a request, creates a process and places it into the *ready-to-run* queue managed by the *CPU Scheduler* module, that is kept sorted according to a specific scheduling policy (see Sec. 3.3). From this

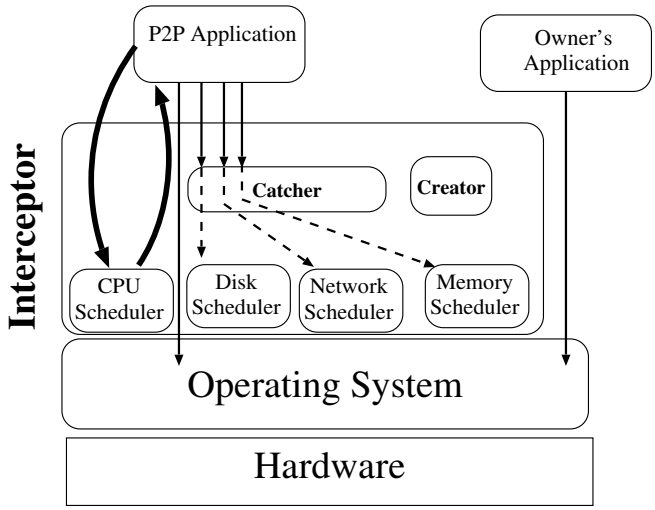

Figure 1: The architecture of Interceptor. Continuos (dotted) thin lines represent issued (intercepted) resource request calls, thick lines correspond to actions performed bo control process execution.

time on, the rate of execution of the P2P process is controlled by the *CPU Scheduler*, that uses specific mechanisms (see Sec. 3.2) to enforce quantitative constraints on CPU usage and to promote performance. During their execution, P2P processes request access to the resources of the node by issuing a set of system calls, either directly or by means of library function calls (that in the rest of this paper we denote, without distinction, *resource request calls*). These resource request calls are transparently intercepted by the *Catcher* module, that forwards them to the proper resource scheduler. This scheduler will in turn place these requests in its own *request queue*, that is kept sorted according to a resource-specific scheduling policy, and delays their forwarding to the local OS in such a way that inequalities (C_1) and (C_2) are satisfied. It should be noted that, as shown in Fig. 1, only those requests issued by P2P applications are intercepted, while applications belonging to the node owner do not undergo the same restrictions.

3.2 Mechanisms

As mentioned before, *Interceptor* works at the middleware level, so it must be able to co-exist with the local OS run on the P2P node. This implies that *Interceptor* must be able to impose its scheduling decisions to the local OS, that otherwise would schedule application requests by using its own policies. This is achieved by means of the following mechanisms, that are combined in such a way that the scheduling decisions made by the underlying OS are overridden whenever necessary:

- *Real-time, fixed-priority round-robing CPU scheduling*: in this policy, the CPU is scheduled in a round-robin fashion using time slices of constant duration. Each process is associated with a fixed priority, and always the process with the highest priority is run, possibly preempting lower priority processes. A process that is preempted by a higher priority process will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. A process that becomes runnable, or that uses an entire time slice, is inserted at the end of the list for its priority. This policy is available in POSIX-compliant OSes, where is named SCHED_RR.

- *High-resolution timers* (such as those specified by the POSIX Realtime Extensions), that are necessary in order to accurately control the allocation and release of node resources.

- *Interception facilities for resource request calls*, needed to transparently mediate resource access. Interception facilities of this type are available in practically all the modern OSes (including Linux and various Windows variants), either in the form of *library interposition* [12, 19] or *system call interception* (e.g., by using facilities like *ptrace* in Unix-like system or similar ones available on other OSes).

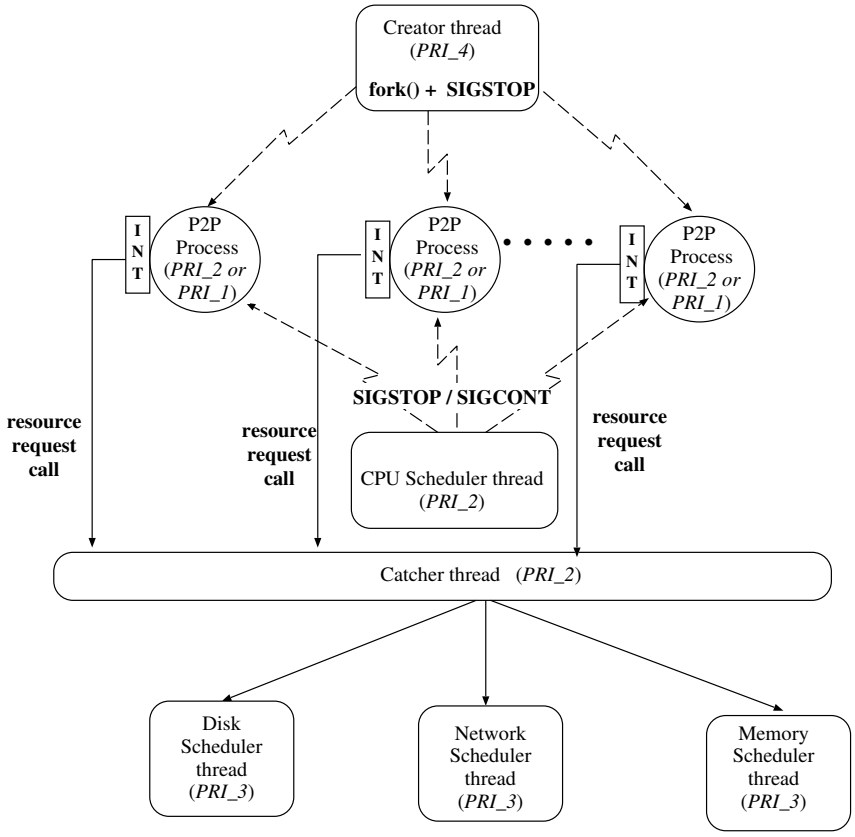

Figure 2: Implementation mechanisms of the Interceptor architecture. Circles, squares, continuous arrows, and dotted arrows correspond to P2P processes, Interceptor's processes, resource requests, and command/signals issued by Interceptor's components, respectively. Each process is labelled with the corresponding fixed priority value.

- *Multi-threading support*, required to implement the various modules of *Interceptor* and to orchestrate their interactions.

Let us explain now how these mechanisms are used to concretely implement the abstract architecture shown in Fig. 1. As shown in Fig. 2, *Interceptor* consists in a set of threads, each one corresponding to a module of the architecture of Fig. 1. These threads, as well as the P2P processes, are scheduled according to the SCHED_RR policy using four priority levels PRI_1, PRI_2, PRI_3, and PRI_4 (where the higher the final number, the higher the priority). The *Creator* thread runs at the highest priority level (PRI_4), so that it is able to preempt any other thread to serve as quickly as possible any new P2P process creation request. Upon receiving a request, the *Creator* creates the process (for instance, as indicated in Fig. 2, by means of a $fork()$ system call), puts it into the stopped state by sending a SIGSTOP signal, places it into the ready-to-run queue of the *CPU Scheduler*, and suspends itself so that the other threads may perform their activities, that are described in the three subsections.

3.2.1 CPU Scheduling

The *CPU Scheduler* carries out its tasks by relying on the SCHED_RR policy provided by the underlying OS (that we assume to be POSIX compliant), but uses its own *ready-to-run* process queue. More specifically, it works by performing an endless loop (the *scheduling loop*) in which it (a) selects the process P_{first} at the head of its ready-to-run queue, (b) dispatches it on the CPU, (c) waits to be scheduled again by the local OS, (d) places the currently running P2P process either back in the ready-to-run queue or into the *I/O-blocked queue* (i.e., a queue that holds those

processes that are blocked waiting for the completion of an I/O operation), and (e) removes from the I/O-blocked queue those processes whose pending I/O operations have been completed since the last iteration of the scheduling loop.

The *CPU Scheduler* forces the local OS scheduler to dispatch process P_{first} by using a variant of the *Dual-Priority Assignment* scheme [7]. In particular, it uses the SCHED_RR policy with two priority levels: a higher level (PRI_2), assigned to the running process, and a lower level (PRI_1), assigned to ready-to-run processes. The *CPU Scheduler* always runs at the PRI_2 priority level, while the priority of a P2P processes alternates between PRI_1 and PRI_2. When the *CPU Scheduler* wants to make the local OS scheduler dispatch P_{first} on the CPU, it raises the priority of this process to PRI_2, and voluntarily yields the CPU. As a consequence of the yield operation, the local OS dispatches P_{first} (it is the ready-to-run process with the highest priority), that then runs until the current time slice expires, or it issues a blocking I/O operation, whatever event occurs first. When one of the above event occurs, the local OS deschedules P_{first}, and gives back the CPU to the *CPU Scheduler*, since it is the ready-to-run real-time process (in the sense of POSIX-scheduling) with the highest priority level (PRI_2). After getting back the CPU, the *CPU Scheduler* decreases the priority of P_{first} by setting it to PRI_1, and places it either into the ready-to-run or into the I/O-blocked queue (depending on the event that caused it to be descheduled).

At the end of each iteration of the scheduling loop, the *CPU Scheduler* suspends itself (thus not selecting any other P2P process to dispatch on the CPU) for an amount of time T_{susp}^{CPU} chosen in such a way that inequality (C_2) holds true, computed as:

$$T_{susp}^{CPU} = \frac{1 - S(P2P, CPU)}{S(P2P, CPU)} * TSD \tag{3}$$

were TSD denotes the duration of the time slice used by the SCHED_RR policy. For instance, if $TSD = 100$ ms. and $S(P2P, CPU) = 0.2$, $T_{susp}^{CPU} = (0.8/0.2) * 100 = 4 * 100 = 400$ ms. This means that the *CPU Scheduler* cannot be active for more that 100 ms. in a period lasting 500 ms., so P2P processes will not be allowed to use the CPU for more that 100 ms. in a period of 500 ms., and consequently will not receive a share larger than $100/500 = 0.2$ (that was the value of $S(P2P, CPU)$ in this example).

3.2.2 Resource request call interception

To use the CPU, a process does not need to issue any specific request: the fact that it is ready-to-run implies that it is implicitly requesting the use of the processor, and the *CPU Scheduler* relies on the information concerning the state of P2P processes to decide whether they are eligible to receive the CPU or not. Conversely, to use the other resources of a node, namely the disk, the network interface, and the physical memory, processes have to issue a set of resource request calls, as already mentioned before. Therefore, the basic mechanism supporting the scheduling of this type of resources consists in the interception of request calls, that is carried out by two components of *Interceptor* that cooperate tightly. The first component is an *interception plug-in*, attached to each P2P process (depicted in Fig. 2 as a box labeled with INT), that intercepts all the resource request calls issued by the process, while the second one is the *Catcher* thread. Upon intercepting a request call, the plug-in sends a message to the *Catcher* thread, and waits for a reply (thus blocking the execution of the corresponding P2P process). When the *Catcher* receives the message, it inspects the contents of the message sent by the plug-in in order to select the scheduler that will handle the resource request, and inserts a *request hook* (containing the identity of the process that issued the call) into a request queue sorted according to the particular scheduling policy in use (as discussed in Sec. 3.3). As described later, the resource scheduler will send a reply message after a given amount of time (computed by the scheduler according to the policy it uses), and this will unblock the the plug-in, that will then issue the call, thus making the execution of the process continue normally.

The interception plug-in may use any interception facility available on a given OS, provided that interacts with the *Catcher* according to a protocol we devised for this purpose. For instance, the implementation of the first version of *Interceptor* [2] used system call interception (achieved by means of the Linux *ptrace* command), while in the implementation of the *Interceptor* version described in this paper we resorted to *library interception* (also known as *function hooking* on Windows [19]) since it is more efficient.

3.2.3 Disk and Network Scheduling

Let us describe now the *Disk* and *Network Scheduler*, that are both based on the idea of delaying, for a suitable amount of time, the posting of the reply messages to the processes that issued the resource request calls. In this way, they can

control the progress rate of these processes, and consequently enforce the quantitative constraints on resource usage set by the node owners.

Let us start with the description of the *Disk Scheduler*, that works by performing an endless scheduling loop in which it (a) selects the request hook at the head of the resource request queue, (b) sends a reply message to the corresponding process, and (c) suspends itself for an amount of time T_{susp}^{Disk} computed in such a way that inequality $(C2)$ is satisfied. When the sleep time elapses, the scheduler wakes up, preempts any other running process (it runs at the PRI_3 priority level, so it will preempt both the *CPU Scheduler* and the running P2P process), and starts over its scheduling cycle.

In order to satisfy inequality $(C2)$, T_{susp}^{Disk} is computed in such a way that at any point in time T the difference $D(T)$ between the nominal disk capacity $C_{Nom}^{P2P}(Disk, T)$ globally allocated to P2P applications and the capacity $C_{Act}^{P2P}(Disk, T)$ they actually received in the same interval is equal to 0. The nominal disk capacity $C_{Nom}^{P2P}(Disk, T)$ corresponds to the number of bytes that can be read/written during in the time interval $[0, T]$ (where time 0 corresponds to the beginning of *Interceptor*'s activity), that is:

$$C_{Nom}^{P2P}(Disk, T) = C_{Disk}(T) \cdot S(P2P, Disk), \tag{4}$$

where $S(P2P, Disk)$ denotes the nominal disk share allocated to P2P applications, and $C_{Disk}(T)$ is the maximum amount of bytes that can be read/written from/to the disk in the $[0, T]$ interval, that is:

$$C_{Disk}(T) = D_B \cdot T \tag{5}$$

where D_B is the disk bandwidth (measured in bytes/sec.). Likewise, the actual disk capacity $C_{Act}^{P2P}(Disk, T)$ received by P2P applications corresponds to the total number of bytes actually read/written up to time T.

In order to make $D(T) = 0$ for any time instant T, each time a read/write request is scheduled, the *Disk Scheduler* updates $C_{Act}^{P2P}(Disk, T)$ by adding the number of transferred bytes and computes $D(T) = C_{Nom}^{P2P}(Disk, T) - C_{Act}^{P2P}(Disk, T)$. A positive value of $D(T)$ indicates that P2P applications have not exceeded $S(P2P, Disk)$, so the *Disk Scheduler* schedules the next request in the queue. If, conversely, $D(T)$ is negative, the *Disk Scheduler* suspends itself for an amount of time $T_{susp}^{Disk} = |D(T)|/D_B$. For example, if $D_B = 40MBytes/sec.$ and $D(T) = 2MBytes$, $D(T + T_{susp}^{Disk}) = 0$ if $T_{susp}^{Disk} = 2/40 = 0.05sec.$

The operations of the *Network Scheduler* are completely identical to those of the *Disk Scheduler*, so we do not report it here. The only obvious difference is that for the *Network Scheduler* uses the information concerning the network bandwidth available to the node, and the amount of bytes sent/received to/from the network.

3.2.4 Memory Scheduling

The last scheduler of *Interceptor* that has to be described is the *Memory Scheduler*. The mechanisms used by this scheduler must be necessarily different from those used by the other *Interceptor* schedulers, since the memory – once allocated to a process – is not released after a predefined amount of time (as instead done for the CPU, the disk and the network interface). Consequently, the principle of enforcing quantitative constraints by delaying the forwarding of system calls cannot be applied anymore. Devising a mechanisms able to override the actions performed by memory management system of the underlying OS is not trivial, and is part of our ongoing work. At the moment, the *Memory Scheduler* acts a simple broker, that keeps track of the amount of memory allocated to P2P processes and forwards an allocation request to the local OS only if this does not make the amount of allocated memory exceed the limit set by the node owner, while it returns an allocation error otherwise.

3.3 Scheduling policies

Interceptor uses a set of proportional-share resource scheduling policies aimed at allocating to each P2P process its nominal share, and to achieve the highest possible performance that can be attained without exceeding the above nominal share. Generally speaking, proportional-share algorithms work by scheduling processes with a frequency proportional to their nominal shares. Unlike the proportional-share algorithms published in the literature for the various resource types (e.g., [22, 32] for the CPU, [29] for the disk, and [21, 33] for the network bandwidth), the ones used by

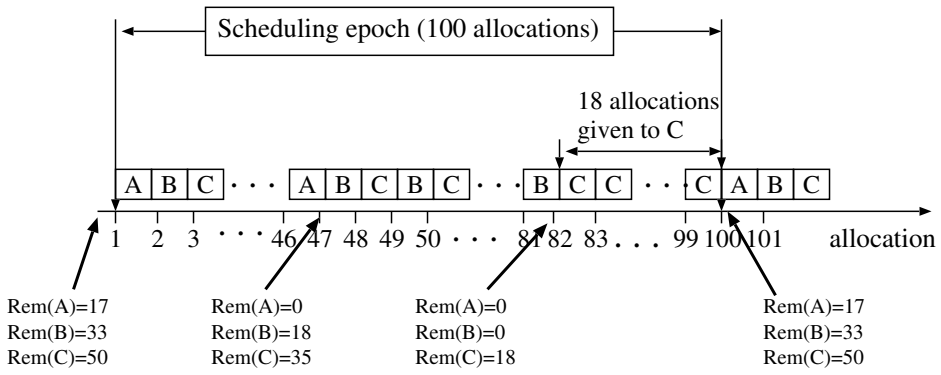

Figure 3: Scheduling example using PNS with $S(A,\tau) = 0.17$, $S(B,\tau) = 0.33$, $S(C,\tau) = 0.5$.

the *Interceptor* have been designed in such a way that they fairly schedule processes whose execution may block for arbitrary amounts of time, and are based on a policy called *Proportional Number of Slices* (or *PNS* for brevity).

PNS works by dividing time into *scheduling epochs*, each one lasting a fixed amount of time T^{τ}_{epoch}, where τ denotes a specific resource. During each epoch, the capacity of resource τ is allocated in discrete fixed quantities called *allocations*. In the case of CPU scheduling, an allocation corresponds to a fixed-duration time slice lasting TS time units, and each CPU scheduling epoch consists in $A_{CPU} = T^{CPU}_{epoch}/TS$ allocations. For instance, if $T^{CPU}_{epoch} = 1$ sec. and $TS = 100$ ms., each scheduling epoch includes $A_{CPU} = 1000/100 = 10$ allocations.

In the case of I/O scheduling, an allocation corresponds instead to a byte that is read from/written to the I/O device (i.e., the disk or the network interface card). Therefore, a scheduling epoch for an I/O device $iodev$, having a nominal capacity of C_{iodev} bytes/sec. consists in $A_{iodev} = C_{iodev} \cdot T^{iodev}_{epoch}$ allocations. For instance, if $T^{Disk}_{epoch} = 1$ msec. and $C_{Disk} = 40$ MBytes/sec., each scheduling epoch includes $A_{disk} = 40 \cdot 10^6 \cdot 8 \cdot 0.001 = 320000$ allocations.

The *PNS* algorithm is based on the idea that in each scheduling epoch a process P_i whose nominal share for resource τ is $S(P_i,\tau)$ should receive the resource for *at least* $Av(P_i,\tau) = C \cdot S(P_i,\tau)$ allocations, provided that it continuously requests to use the resource for the entire epoch. For instance, in an epoch lasting 100 allocations, if $S(P_i,\tau) = 0.5$ then process P_i should receive at least 50 allocations. The actual PNS algorithm, described later in this section, is a variant of the simpler proportional-share algorithm described below. This simpler algorithm works by associating with each process P_i a variable, $Rem(P_i,\tau)$, set to $Av(P_i,\tau)$ at the beginning of each scheduling epoch, that is decremented of one unit each time P_i receives an allocation. Ready-to-run processes are kept into a queue, sorted in non-increasing order with respect to their $Rem()$ values, and the process at the head of the queue always runs. An example is shown in Fig. 3, where resource τ is shared among three processes (A, B, and C) having nominal shares $S(A,\tau) = 0.17$, $S(B,\tau) = 0.33$, and $S(C,\tau) = 0.5$. As shown in the above figure, the resource is multiplexed in a round robin way among the three processes until A completes its $Rem(A,\tau)$ allocations (at time 46). From time 47 to 82 the resource is then multiplexed between B and C only, until also B terminates its $Rem(B,\tau)$ allocations for the current scheduling epoch. Finally, C receives 18 consecutive allocations (from time 83 to 100). This pattern repeats in all the subsequent scheduling epochs.

Unfortunately, this simple algorithm fails to allocate the nominal shares to processes that block their executions for arbitrary amounts of time. For instance, if B blocks its execution from time 48 to 100, at the end of the scheduling epoch it will have received only 15 allocations instead of 33 (the ones it receives from time 1 to 47) and, being $Rem(B,\tau)$ reset to $Av(B,\tau)$ at the beginning of the new epoch, it will never receive these missing 18 allocations. To correctly deal with these situations, *PNS* associates with each process P_i an additional variable, $Done(P_i,\tau)$, that keeps track of the scheduling epochs completed by P_i, and sorts the request queue for resource τ in increasing order with respect to the scheduling epochs completed by the processes issuing the requests (requests of processes with the same $Done()$ values are sorted in non-decreasing order with respect to their $Rem()$ values). To exemplify, let us show in Fig. 4 how the blocking of process B (in Fig. 3) is handled by the *PNS* algorithm. As show in Fig. 4, after B blocks (time 48) C gets 35 consecutive allocations, since it has precedence over A (being $Done(C) = 0$) and B is blocked.

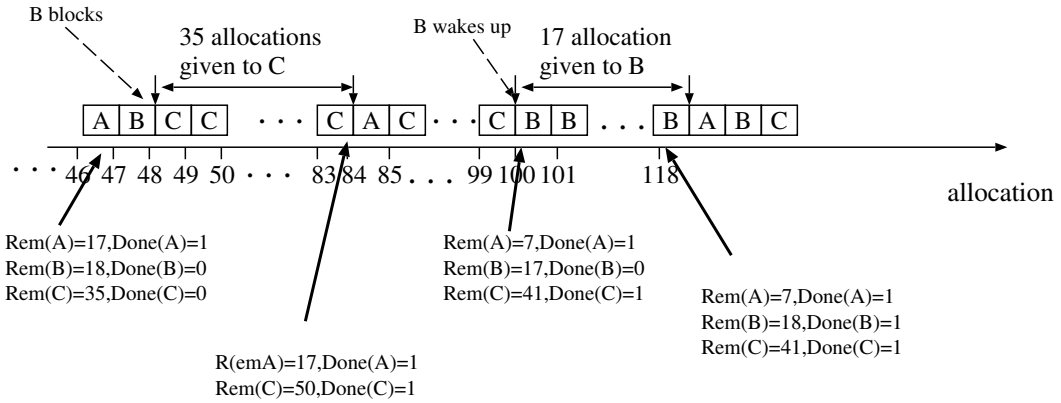

Figure 4: Dealing with blocking processes in PNS.$R()$ denotes the $Rem()$ values, while $D()$ denotes the $Done()$ values for the processes.

After C completes its current scheduling epoch (time 84), the resource is multiplexed between A and C (they both have completed 1 epoch) until time 100, when B wakes up. After waking up, B receives the 18 next allocations (i.e., until it completes its first epoch at time 118), and only after that time the resource is multiplexed again among the three processes.

Interceptor uses the *PNS* algorithm to schedule all the resources of a node. While the PNS scheduler for the CPU works exactly as discussed above, the ones used to schedule the disk and the network interface capacities adopt a simple optimization in which they do not schedule individual bytes, but rather individual read/write requests (each one concerning a different number of bytes). This means that after an individual request is completed, the corresponding process might have received a resource share larger than its nominal one. However, this will result in a $Rem()$ value smaller than the other competing processes (or in a larger $Done()$ value), so the subsequent requests for the process that temporarily exceeded its allotted disk share will be scheduled after a delay directly proportional to the amount of extra capacity it received.

4 Experimental evaluation

In order to assess the feasibility and effectiveness of *Interceptor*, we have developed a proof-of-concept implementation, and we used it to carry out a thorough experimentation aimed at quantitatively evaluating, for a large set of operational scenarios, the capability of *Interceptor* to provide the following properties:

- *Segregation*, i.e. the ability of enforcing the quantitative limitations set by the node owner on the amount of resources allocated to P2P applications;
- *Scheduling fairness*, i.e. the ability of allocating to each P2P application a fraction of resource capacity no lower than its nominal share;
- *Scheduling performance*, i.e. the ability of maximizing the performance of P2P applications without exceeding the quantitative limitations on resource usage;
- *Scheduling overhead*, i.e. the ability of slowing down as little as possible the execution of P2P applications with respect to the case in which *Interceptor* is not used to schedule them.

These properties have been evaluated by submitting to *Interceptor* a variety of workloads, comprising CPU-bound, I/O-bound and interactive applications, and by measuring the following set of performance metrics:

- The *Global Allocation Lag* $GAL(\tau)$ for a given resource τ, defined as:

$$GAL(\tau) = \sum_{P_i \in P2P} Lag(P_i, WT(P_i), \tau) \tag{6}$$

  where P_i is a P2P process, $WT(P_i)$ is the *wallclock time* of process P_i (i.e. the time elapsing between its creation and the termination of its execution), and $Lag()$ (defined as in Eq.(2)) is the resource lag accumulated by P_i during its entire execution. Intuitively, $GAL(\tau)$ represents the difference between $S(P2P, \tau)$ and the total share of τ globally received by all P2P applications. Therefore, if $GAL(\tau) \geq 0$, then segregation has been achieved. Furthermore, the closer is $GAL(\tau)$ to 0, the better is the scheduler in precisely allocating to P2P process their nominal shares.

- The *Average Relative Allocation Error* $ARAE(\tau)$ for a given resource τ, defined as the average of the relative allocation errors of all the P2P applications composing the workload, namely:

$$ARAE(\tau) = \frac{1}{N} \sum_{P_i \in P2P} \frac{Lag(P_i, WT(P_i), \tau)}{S(P_i, \tau)} \cdot 100\% \tag{7}$$

  where N is the number of P2P applications. Intuitively, the closer is $ARAE(\tau)$ to 0, the smaller the difference between the share received by individual P2P applications. Therefore, the $ARAE$ metric is used to quantify scheduling fairness.

- The *Average Wallclock Time (AWT)*, defined as the average wallclock time of all the P2P applications composing the workload, is used to quantify scheduling performance.

- The *Relative Wallclock Time Increase (RWTI)*, defined as:

$$(WT_{Interceptor} - WT_{Alone})/WT_{Alone} \tag{8}$$

  where $WT_{Interceptor}$ (WT_{Alone}) denotes the wallclock time measured when *Interceptor* was (not) used, that quantifies scheduling overhead.

The results of our experiments, discussed in the remainder of this section, demonstrate the ability of *Interceptor* to simultaneously provide very good application segregation, fairness and scheduling performance, while at the same time introducing a negligible overhead.

The rest of this section is organized as follows. After a brief description of the proof-of-concept implementation used to carry out the experiments (Sec. 4.1), we describe the workloads and the experiments we performed to evaluate the *CPU Scheduler* (Sec. 4.2). We then continue with the workloads and the experiments concerning the *Disk Scheduler* (Sec. 4.3) (the results concerning the *Network Scheduler* do not present significant differences, so we do not to include them here), and then conclude the section by discussing the results concerning the overhead introduced by the above schedulers (Sec. 4.4).

4.1 The Linux implementation

Our proof-of-concept implementation, written in C++, runs on the Linux OS that, being fully POSIX-compliant, provides the basic thread mechanisms and real-time scheduling support discussed before. The high-resolution timing support was provided by the PAPI library [23], since Linux lacks the implementation of high-resolution POSIX timers. For resource call interception we used the basic library interposition mechanisms [12] provided by all Unix systems, that consists in rewriting the code for the library calls that must be intercepted, storing them into a library, and using the $LD\_PRELOAD$ environment variable to force the dynamic loader to search this library for each called function before looking up the standard library. This mechanism works only for dynamically linked applications, but we decided to use it to avoid the excessive performance overhead characterizing system call interception in Linux. It is worth to point out that, however, this has been a mere implementation choice, and not a requirement posed by the *Interceptor* design. Therefore, if support for statically linked applications is desired, an alternative mechanism like system call interception can be used.

4.2 Evaluation of the CPU Scheduler

The evaluation of the *CPU Scheduler* has been carried out by considering the following workloads:

- *CPU-intensive workloads*, comprising a set of identical CPU-intensive applications. We chose *EMAP* [15], a real-world CPU-bound ElectroMagnetic Analysis Program, and we considered 10 different workloads obtained by progressively increasing from 1 to 10 the number of *EMAP* instances simultaneously executed;
- *Interactive workloads*, comprising a set of identical interactive applications. We chose *Video*, an application taken from the Interbench Benchmark Suite [20], that emulates a soft real-time video application requiring to be scheduled 60 times per second and using a predefined percentage of the CPU each time it is scheduled. We considered four variants of *Video*, namely *Video-20%*, *Video-40%*, *Video-60%*, and *Video-80%*, obtained by setting to $20\%, 40\%, 60\%$, and 80% the CPU percentage requested. For each of these variants, we built 10 different workloads by progressively increasing from 1 to 10 the number of *Video* instances ran simultaneously. Globally, we thus obtained 40 different workloads.
- *Mixed workload*, comprising a mix of CPU-intensive and interactive applications. More precisely, we considered four different workloads, obtained by combining an instance of *EMAP* with an instance of each variant of *Video*.

4.2.1 Experiments with CPU-intensive workloads

The first set of experiments was aimed at evaluating the behavior of *Interceptor* when dealing with CPU-intensive workloads. For each of the 10 different workloads built by increasing the number N of *EMAP* instances from 1 to 10, we ran experiments in which the CPU share $S(P2P, CPU)$ globally allocated to P2P applications was progressively increased from 0.1 to 0.9. In all experiments the nominal share of each *EMAP* instance was set to $S(P_i, CPU) = S(P2P, CPU)/N$, in order to ensure that the share actually received by any instance is not larger than its nominal share (a set of identical instances receiving the same share will indeed terminate their executions at the same time, thus maintaining a constant CPU demand). The results concerning the Global Allocation Lag (Fig. 5(a)), reporting values included in the range $[-1.5\%, 1.5\%]$ for all the experiments but one (where it reached 4%), clearly demonstrate the ability of *Interceptor* to effectively segregate P2P applications, since in none of the experiments the CPU share received by P2P applications exceeded $S(P2P, CPU)$. Furthermore, the values obtained for the Average Relative Allocation Error (Fig. 5(b)) indicate that *Interceptor* provides a rather good scheduling fairness, since on average the allocation error for each instance was very low (between -2% and 2.5%) for all workloads and experiments. Finally, as shown in Fig. 5(c) (reporting the Average Wallclock Time of the *EMAP* instances) *Interceptor* is able to efficiently schedule P2P applications. As a matter of fact, for a given number of instances run simultaneously, the average execution time we measured was directly proportional to the share globally allocated them. For instance, the average execution time $AWT_{10}(10\%)$ obtained for 10 instances with a share of 10% (1286.1 sec.) is about 9 times larger than the execution time $AWT_{10}(90\%)$ obtained for the same number of instances with a share of 90% (145.7 sec.). Furthermore, for a given P2P CPU share value $X\%$, the $AWT_n(X\%)$ measured with n instances was very close to $n \cdot AWT_1(X\%)$. For instance, when the P2P CPU share was 20%, we measured $AWT_1(20\%) = 64.4$ sec., $AWT_2(20\%) = 129.6$ (i.e., almost equal to $2 \cdot AWT_1(20\%)$, $AWT_4(20\%) = 258.2$ sec. (i.e. almost equal to $4 \cdot AWT_1(20\%)$), up to $AWT_{10}(20\%) = 645.3$ sec. Likewise, for a 50% P2P CPU share, we had $AWT_1)(5o\%) = 26$ sec., $AWT_2(50\%) = 52.3$, $AWT_4(50\%) = 104.3$ sec., and $AWT_{10}(50\%) = 260.6$ sec. This demonstrates that all the instances are properly scheduled independently from the CPU share globally allocated to P2P applications, and from their number.

4.2.2 Experiments with interactive workloads

The second set of experiments was aimed at evaluating the behavior of *Interceptor* when dealing with interactive workloads. For each of the 40 workloads we considered (as discussed in Sec. 4.2), we ran experiments in which we progressively increased from 10% to 90% the CPU share $S(P2P, CPU)$ globally allocated to P2P applications, and we allocated to all instances the same nominal share. As can be observed from Fig. 6(a), also for interactive applications the segregation capabilities of *Interceptor* are rather good (the GAL never exceeded 3%, and in the almost totality of experiments was included between -2% and 1%). An analogous consideration can be done for the fairness, as indicated

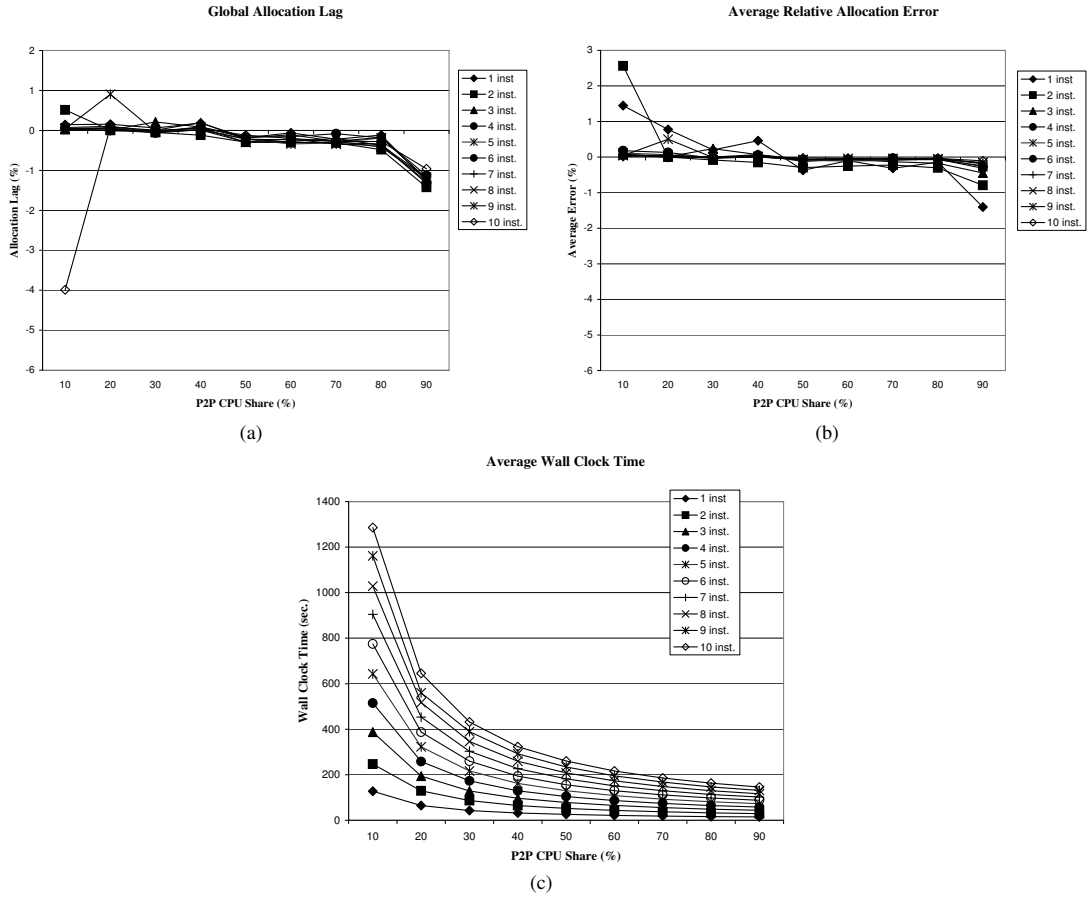

Figure 5: CPU Workload1 results: (a) Global Allocation Lag (GAL), (b) Average Relative Allocation Error (ARAE), and (c) Average Wallclock Time (AWT) of *EMAP* instances.

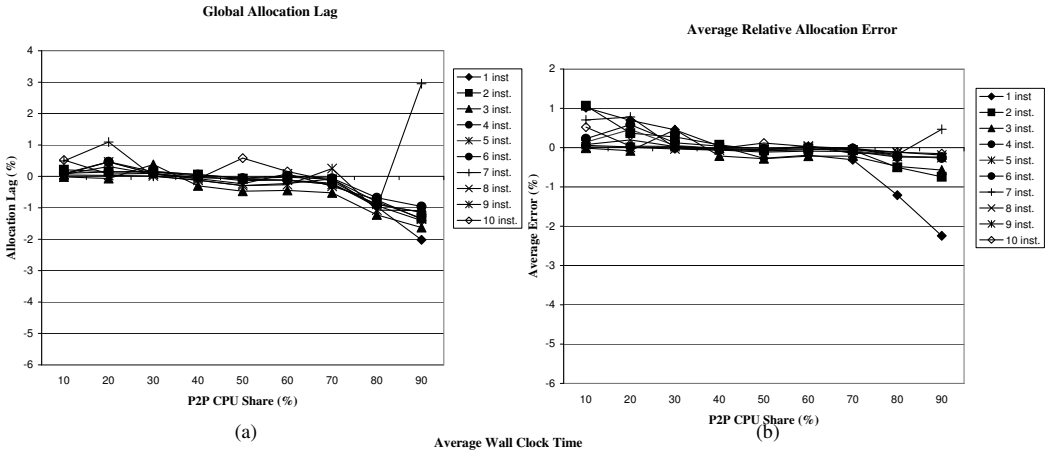

Figure 6: CPU Workload2 results: (a) Global Allocation Lag (GAL), (b) Average Relative Allocation Error (ARAE), and (c) Average Wallclock Time (AWT) of *Video* instances.

by the results concerning the ARAE (Fig. 6(b)), that show values almost always very close to 0%. Finally, scheduling effectiveness is demonstrated by the results concerning the AWT (Fig. 6(c)), that show a behavior practically identical to the one exhibited by *Interceptor* when dealing with CPU-intensive workloads. That is, also in this case we observed that the AWT_n obtained with a given number n of instances was very close to $n \cdot AWT_1$. Moreover, in addition to the AWT, we measured an additional performance indicator for the *Video* application, namely the number of *missed deadlines* (a deadline is considered to be *missed* each time the application is scheduled later than an integer multiple of 1/60th of second). In practically all the experiments we ran with Workload 2, the number of missed deadlines was 0, thus reinforcing our conclusions concerning the scheduling effectiveness of *Interceptor* when dealing with interactive applications.

4.2.3 Experiments with mixed workloads

In order to verify whether the *CPU Scheduler* is able to properly handle workloads comprising *both* CPU-intensive and interactive applications, we ran a third set of experiments in which the workload consisted in an instance of *EMAP* simultaneously executed with an instance of *Video* (as already discussed, we built 4 workloads – one for each variant of *Video*). As in the previous two sets of experiments, also in this case we progressively increased the share $S(P2P, CPU)$ allocated to P2P applications from 0.1 to 0.9. For each value of $S(P2P, CPU)$ we considered three different allocation scenarios in which *Video* and *Emap* were allocated (a) 30% and 70%, (b) 50% and 50%, and (c) 70% and 30% of $S(P2P, CPU)$, respectively. The results concerning scenario (a) are reported in Fig. 7, while those concerning scenarios (b) and (c) – that are practically identical to these ones – are not reported here. First of all, these results indicate that the scheduling performance of the *CPU Scheduler* are rather good also for workloads that include both CPU-intensive and interactive applications. As a matter of fact, the AWT values measured for *EMAP* (Fig. 7(a)) exhibit a similar behavior of those measured in the previous two sets of experiments. Furthermore, the AWT values depend only on the CPU share allocated to this application (and not on the behavior of the competing application, as indicated by the fact that the four AWT curves corresponding to the different versions of *Video* are overlapped). A similar consideration can be done for the AWT values measured for the various *Video* versions (Fig. 7(b)). We note, however, that for a given CPU share, the AWT values measured for the various versions of *Video* differ among them. For instance, for *Video-20%* the AWT measured for $S(Video, CPU) = 3\%$ is approximately 250 sec., while for *Video-80%* is more than 600 sec. This phenomenon depends on the fact that for a given $S(Video, CPU)$ value, the higher the CPU percentage requested by *Video* each time it is scheduled, the higher the execution time, since a larger CPU share is requested to satisfy the requirement of the application. Interestingly, the results concerning the Global Allocation Lag (Fig. 7(c)) are apparently unsatisfactory, since relatively high GAL values have been obtained. This was direct consequence of the fact that the *Video* application always received a share exceeding its nominal one. This, however, is explained by the fact that *EMAP* was always the first application to terminate, and it received exactly its nominal share (as the CPU demand during its execution was constant), while *Video* – in addition to its nominal share – received also the P2P share left "free" by *EMAP* after its termination. Therefore, P2P applications did not etch the resource share reserved to owner's applications (this has been confirmed by measurements we performed by running a CPU-intensive application not controlled by *Interceptor*). This demonstrates, once again, the good segregation and fairness properties of *Interceptor*'s CPU Scheduler.

4.3 Evaluation of the I/O Schedulers

Let us discuss now the experiments we performed to evaluate the segregation, fairness, and performance capabilities of the *Disk* and *Network* schedulers of *Interceptor*. As already discussed in Sec. 3.2.3, these schedulers are practically identical, and thus exhibit very similar performance, so in this paper we report only the results concerning the *Disk Scheduler*. For our evaluation, we considered workloads obtained by running several instances of two base applications, namely cp (the standard file copy utility available in Unix systems) and gzip (the classical file compression utility). These applications have been chosen since they are representative of two distinct application classes: cp is a typical disk-intensive application, that uses a very small amount of CPU capacity, while gzip requires a mix of disk and CPU capacity. Starting from these applications, we built two sets of workloads:

- *Disk-intensive workloads*, comprising several instances of cp. In particular, we considered 5 different workloads obtained by progressively increasing the number of instances of cp executed simultaneously from 1 to 5.

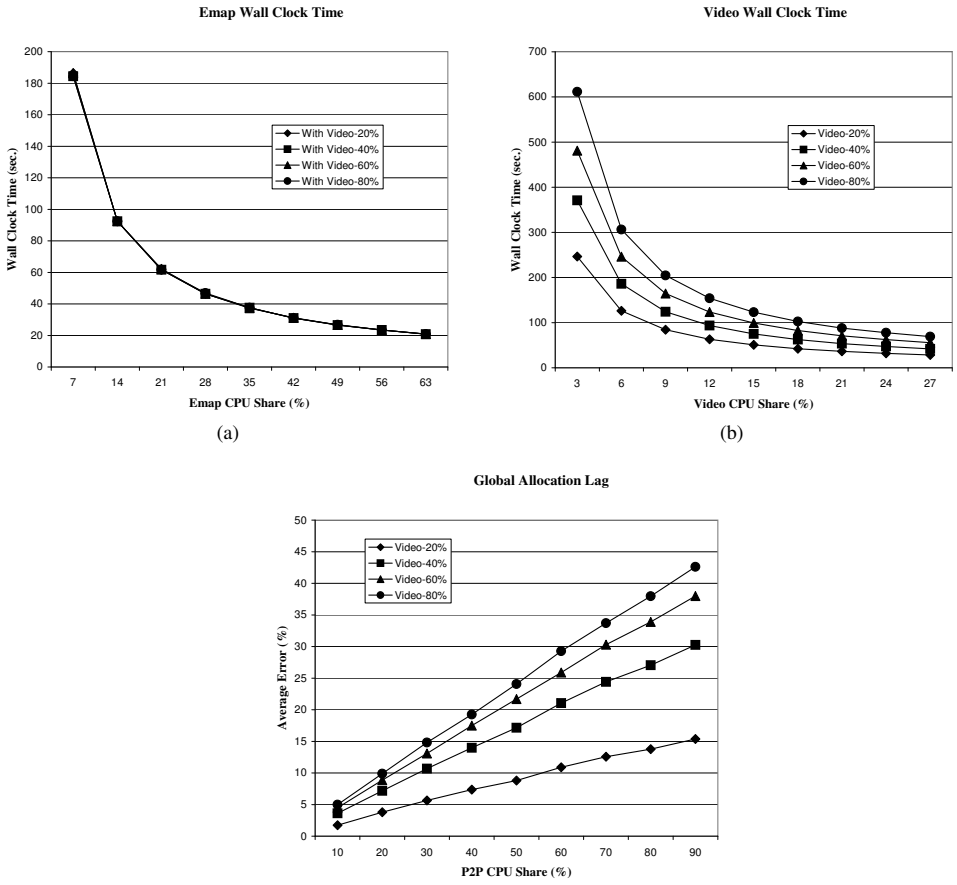

Figure 7: Workload 3 results: (a) Average Wallclock Time for *EMAP*, (b) Average Wallclock Time for *Video*, (c) Global Allocation Lag.

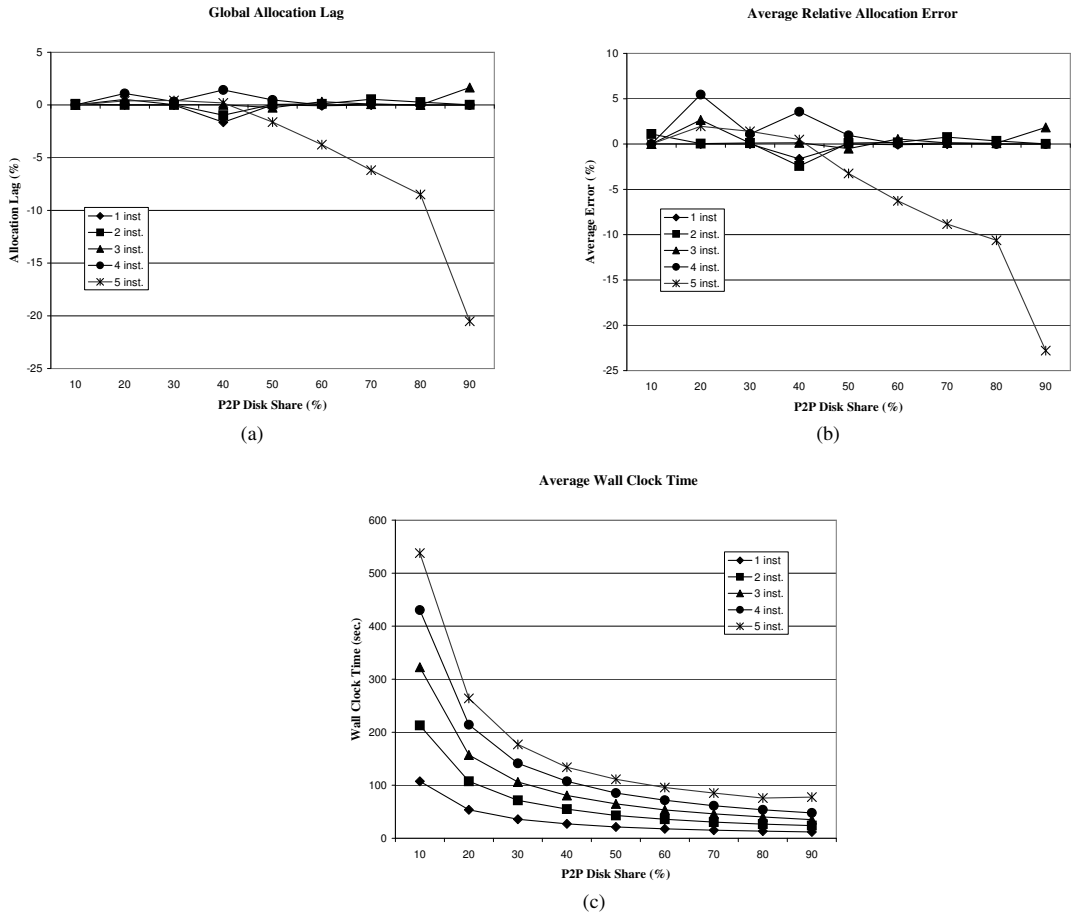

Figure 8: Disk Workload 1 results: (a) Global Allocation Lag (GAL), (b) Average Relative Allocation Error (ARAE), and (c) Average Wallclock Time (AWT) of cp instances.

- *Mixed disk+CPU workloads*, comprising several instances of gzip. As for the cp-based workloads, also in this case we considered 5 workloads obtained by progressively increasing from 1 to 5 the number of gzip instances executed simultaneously.

For all the workloads, the size of the file provided as input to both applications was set to 200 MBytes. Furthermore, in all the experiments all the instances of applications received the identical nominal shares for both the CPU and Disk capacity.

4.3.1 Experiment with the disk-intensive workloads

Let us start our discussion by describing the results we obtained for the workloads including instances of the cp application only, that are reported in Fig. 8. As can be seen from Fig. 8(a), the *Disk Scheduler* provides a good segregation level, as indicated by very low GAL for all the experiments but one. The only exception is represented by the experiments in which 5 instances of cp were simultaneously executed, that resulted in increasingly larger GAL values for increasing values of the P2P disk share. This is due to the fact that, although cp is disk-intensive, it still requires the use of the processor, and when each instance has to compete for the CPU with other 9 instances, it does not receive enough CPU capacity to issue a number of disk request high enough to exploit all its nominal disk share. Fig. 8(b) reports instead the $ARAE$ whose low values (exhibiting a behavior similar to the GAL) indicate a very good fairness. Finally, the AWT (Fig. 8(c)) clearly indicate the good performance characteristics of the *Disk Scheduler*.

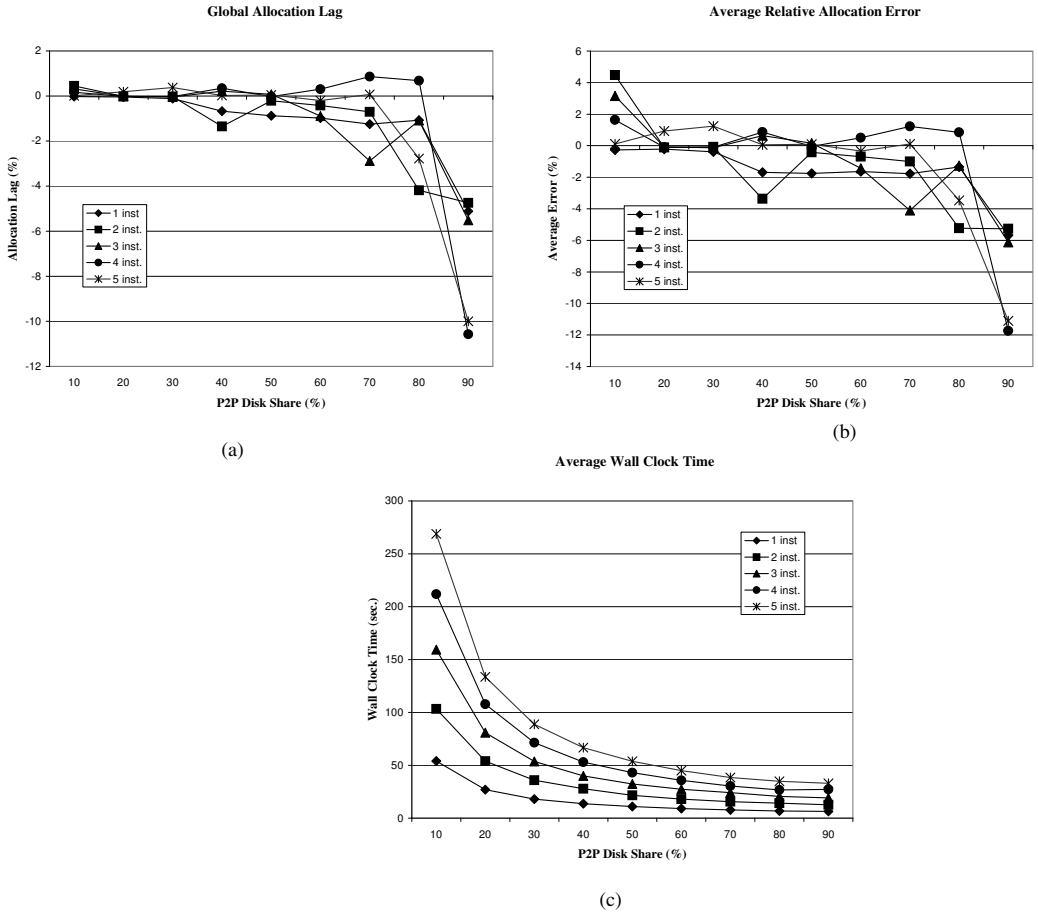

Figure 9: Disk Workload 2 results: (a) Received share (b) Wall clock time.

4.3.2 Experiments with the mixed disk+CPU workloads

This second set of experiments was aimed at assessing the behavior of the *Disk Scheduler* for workloads consisting of applications requiring both CPU and disk capacity. As indicated by the results in Fig. 9, also for these workloads both the segregation capabilities (Fig. 9(a)), the allocation fairness (Fig. 9(b)), and the scheduling performance (Fig. 9(c)) are rather good. It is worth noticing that also in this case the GAL values for the 5 gzip instance become higher for higher values of the P2P disk share, as observed for the cp-based workloads.

4.4 Evaluation of the scheduling overhead

A natural question that might arise at this point is whether the good scheduling and segregation performance exhibited by *Interceptor* comes at a high price in terms of the overhead induced on the performance of the application it controls. In order to answer this question, we performed a set of experiments in which we measured the Relative Wallclock Time Increase for all the applications we used to build the workloads described above. In particular, we executed each of these applications in isolation first by using the local OS scheduler, and then by using *Interceptor* to schedule it. For each experiment, we measured the wallclock time, and computed the $RWTI$ as defined in Eq.(8). In both cases, the application had exclusive access to the resources of the machine on which the experiment was performed. In the experiments that did not use *Interceptor*, we put the machine in single-user mode, and we set the scheduling class of application to SCHED_RR with a priority level set to 99. Conversely, in the experiments using *Interceptor*, each application was executed alone after setting to 100% both its CPU and disk nominal shares. The results we obtained

Table 1: Overhead of the *Interceptor* CPU scheduler

Application	**Local OS time (sec.)**	**Interceptor time (sec.)**	**RWTE (%)**
EMAP	29.55	29.93	1.3
Video-20%	20.16	20.3	0.69
Video-40%	40.07	40.14	0.17
Video-60%	60.06	60.20	0.23
Video-80%	60.06	60.20	0.23

Table 2: Overhead of the *Interceptor* Disk scheduler

Application	**Local OS time (sec.)**	**Interceptor time (sec.)**	**RWTE (%)**
cp	1.032	1.1	6.5
gzip	5.56	5.92	6.48

for the CPU scheduling overhead are reported in Table 1, where the column **Local OS time**, **Interceptor time**, and **RWTI** report the wallclock time (in seconds) measured when the local OS was used to schedule the application, when *Interceptor* was used, and the resulting $RWTI$, respectively. As can be seen from the above table, the overhead of the *CPU Scheduler* is negligible (ranging from 0.17% to 1.3%).

The results we obtained for the disk scheduling overhead are instead reported in Table 2, that shows $RWTI$ values relatively higher than for the CPU scheduling case, but still reasonable.

5 Conclusions and future work

In this paper we have presented *Interceptor*, a middleware-level application segregation and scheduling able to strictly enforce quantitative limitations on resource usage. Interceptor is designed in such a way that can be easily ported on any POSIX-compliant OS, and does not require any modification to existing applications or to the local OS used on individual P2P nodes. The feasibility of *Interceptor* has been demonstrated through the development of a Linux based implementation, that has been used to carry out an extensive experimentation in which we quantitatively evaluated the capabilities of *Interceptor* for a large set of workloads comprising real applications. Our results clearly indicate that *Interceptor* is able to enforce quantitative limitations on resource usage (thus providing application segregation for P2P applications), and to fairly and effectively schedule P2P applications, while at the same time introducing a negligible overhead.

Future work includes the study of memory management mechanisms, working at the user level, in order to develop a better memory scheduler, and its integration into the current Linux-based implementation. Furthermore, we plan to port *Interceptor* on Windows-based platforms. On the methodological side, we plan to use *Interceptor* to investigate issues like the benefits and drawbacks of independently allocating and scheduling the various node resources versus allocating them in a coordinated way.

References

[1] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proc. of the 5th IEEE/ACM Int. Workshop on Grid Computing*, Pittsburgh, PL, USA, November 2004.

[2] C. Anglano. Interceptor: Applications Segregation and Scheduling for P2P Systems. In *Proc. of the 3rd Int. Workshop on Hot Topics in Peer-to-Peer Systems (HotP2P)*, Rhodes, Greece, April 2006.

[3] H. Balakrishnan, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up Data in P2P Systems. *Communications of the ACM*, 46(2), February 2003.

[4] S. Balfe, A.D. Lakhani, and K.G. Patterson. Trusted Computing: Providing Security for Peer-to-Peer Networks. In *Proc. of the Fifth IEEE Int. Conference on Peer-to-Peer Computing (P2P '05)*, Konstanz, Germany, Sept. 2005. IEEE Press.

[5] R. Balzer and N. Goldman. Mediating Connectors. In *ICDCS Workshop on Electronic Commerce and Web-based Applications*. IEEE Press, 1999.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfile. Xen and the Art of Virtualization. In *Proc. of ACM Symp. on Operating Systems Principles*. ACM Press, 2003.

[7] A. Burns and A.J. Wellings. Dual Priority Assignment: A Practical Method for Increasing Processor Utilization. In *Proc. of 5th Euromicro Workshop on Real-Time Systems*. IEEE Press, 1993.

[8] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, and O. Lodygensky. Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. *Future Generation Computer Systems*, 2004. To appear.

[9] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. In *Proc. 4^{th} USENIX Windows Systems Symposium*, Aug 2000.

[10] F. Chen, T. Repantis, and V. Kalogeraki. Coordinated Media Streaming and Transcoding in Peer-to-Peer Systems. In *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium*, Denver (CO), USA, April 2005. IEEE.

[11] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[12] T.W. Curry. Profiling and Tracing Dynamic Library Usage via Interposition. In *Proc. of 1994 USENIX Conference*, 1994. http://www.usenix.org/publications/library/proceedings/bos94/curry.html.

[13] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[14] E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004.

[15] The ElectroMagnetic Analysis Program (EMAP). http://www.ecmlab.umr.edu/emap.html. Accessed on Nov. 13th, 2005.

[16] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust Incentives Techniques for Peer-to-Peer Networks. In *Proc. of ACM Conference on Electronic Commerce (EC '04)*, New York, NY, USA, May 2004.

[17] R. Figueiredo, P. Dinda, and J. Fortes. Guest Editors' Introduction: Resource Virtualization Renaissance. *IEEE Computer*, 38(5), May 2005.

[18] L. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proc. of 6^{th} USENIX Security Symp.*, 1996.

[19] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, WA, USA, July 1999.

[20] The Linux Interactive Benchmark (InterBench). http://members.optusnet.com.au/ckolivas/interbench. Accessed on Nov. 13th, 2005.

[21] K. Jeffay, F.D. Smith, A. Moorthy, and J. Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proc. of IEEE Real Time Systems Symposium*, Madrid, Spain, Dec. 1998.

[22] J. Nieh and M.S. Lam. A SMART Scheduler for Multimedia Applications. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[23] The Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi. Accessed on Nov. 13th, 2005.

[24] M. Rennhard and B. Plattner. Practical Anonymity for the Masses with MorphMix. In *Proc. of 8th Int. Conf. on Financial Cryptography*, LNCS, Key West, FL, USA, February 2004. Springer.

[25] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. In *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP-18)*, Canada, November 2001.

[26] D. Schmidt, D. Levine, and S. Mungee. The Design and Performance of a Real-time Object Request Broker. *Computer Communications*, 21(4), April 1998.

[27] C. Shen, O. Gonzalez, R. Ramamritham, and I. Mizunuma. User Level Scheduling of Communicating Real-Time Tasks. In *Proc. of 5th IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.

[28] P. Shenoy, S. Hasan, P. Kulkarni, and K. Ramamrithan. Middleware versus Native OS Support: Architectural Considerations for Supporting Multimedia Applications. In *Proc. of 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, USA, Sept. 2002.

[29] P. Shenoy and H.M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real Time Systems*, 22(1), Jan. 2002.

[30] Vwmare. http://www.vmware.com. Visited on Dec. 20, 2006.

[31] T. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-based Fault Isolation. In *Proc. of 14^{th} ACM Symp. on Operating Systems Principle*, 1993.

[32] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Computer Science, Massachussets Institute of Technology, 1995.

[33] M. Zhang, R. Wang, L. Peterson, and A. Krishnamurthy. Probabilistic Packet Scheduling: Achieving Proportional Share Bandwidth Allocation for TCP Flows. In *Proc. of INFOCOM 2002*, New York, NY, USA, June 2002.