Interceptor: Middleware-level Application Segregation and Scheduling for P2P Systems [*]

Cosimo Anglano
Dipartimento di Informatica
Università del Piemonte Orientale, Alessandria (Italy)
email:cosimo.anglano@unipmn.it

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.*

1 Introduction

The Peer-to-Peer (P2P) paradigm has recently emerged as an effective way of organizing highly scalable distributed systems, as demonstrated by quite a few P2P systems designed for the provision of services of various type, such as efficient data lookup [5], media streaming and transcoding [12], DNS address resolution [13], distributed storage management [21], distributed file systems [14], and anonymous communication [20]. In order to properly work, these services require the availability of a large number of nodes, possibly scattered across the globe. The only viable option to build such very large size P2P systems seems to be the aggregation of nodes belonging to many individual entities (single individuals or institutions), that must be encouraged to join the system with their own resources. This can be achieved only if each node owner, besides receiving appealing incentives [16] and being convinced about the thrustworthiness [6] of P2P users and applications, is guaranteed that P2P applications will not unacceptably degrade the performance of his/her own applications because of an excessive resource consumption. 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 respective owner.

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 and platform, and must not require any modification to existing operating systems and applications. Placing restrictions on the supported operating systems would indeed results in limitations on the number of nodes that can participate to the P2P system, while requiring modifications to the operating systems and applications would raise natural concerns about security and unacceptably increase software maintenance costs and problems.

In this paper we propose *Interceptor* [1], an application segregation system that is able to simultaneously meet all the above requirements. *Interceptor* allows node owners to specify quantitative constraints on the share allocated to P2P applications for each node resource, and enforces them by means of a set of resource-limitation mechanisms. Furthermore, *Interceptor* allows one to allocate to individual P2P applications different resource shares, so that more resources can be allocated to critical P2P services (e.g., DHT routing). Finally, *Interceptor* encompasses a set of resource scheduling mechanisms and policies aimed at maximizing P2P applications performance without violating resource

[*]This work has been supported by the Italian FIRB "WebMINDS" project

[1]The name *Interceptor* has been choses to indicate that, as discussed later, it is based on the idea of intercepting the resource access requests issued by P2P applications.

limitations. *Interceptor* is designed in such a way that can be implemented entirely at the middleware-level on standard operating systems that provide system call interception facilities (like many modern operating systems, 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). As will be discussed in later sections, this makes *Interceptor* totally transparent to applications and able to co-exist with the local operating systems run on individual nodes. Furthermore, *Interceptor* is designed in such a way that it is possible to change the scheduling policy adopted for any resource without having to modify anything else.

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. At the moment of this writing, only the CPU scheduling policy has been fully implemented, so we will discuss only results concerning with CPU scheduling. Our results demonstrate the effectiveness of the application segregation mechanisms, and the fairness and effectiveness of the CPU scheduling algorithm. 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 in most public-resource computing systems [3, 10], consisting 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 simultaneously run on the node.

An alternative solution, able to support multiple P2P applications, is *sandboxing*, that consists in providing a virtual execution environment that mediates applications resource accesses. Sandboxing uses code transformation techniques (such as binary modification [26] and API interception [7, 17]) to monitor and control the application's interactions with the underlying OS, in such a way that the desired behavior is enforced. Sandboxing has been mainly used to enforce qualitative restrictions (i.e., restrictions on what resources can be used by an application), but recently it has been used to enforce quantitative constraints as well [8, 11]. The main problem of sandboxing is its lack of generality, as any sandboxing technique is platform specific. Even if the same technique can sometimes be exported on a different platform, the development effort usually is quite significant. Moreover, the use of a sandboxing technique that requires unmediated access to the hardware (e.g., Xen [8]) raises obvious security concerns, while resorting to user-level sandboxing does not give the complete guarantee that the segregation mechanisms are not bypassed (either intentionally or not) by applications [11].

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. Kernel-level techniques do not present the same drawbacks of sandboxing, since resource access is mediated by the OS, but 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 [15, 22, 23] have been proposed as an alternative to kernel-level techniques. As shown in [24], middleware approaches, despite their potentially high overhead, often result in application performance similar to those attained by adopting 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. As sandboxing, kernel-level techniques and middleware solutions, it supports the simultaneous execution of P2P applications, but unlike them it is not OS specific (since it relies on facilities available on practically all modern operating systems), nor it requires any modification to the operating system running on P2P nodes. Moreover, it is able to co-exist with the OS running on the node, but unlike traditional middleware-level techniques it does not require any modification to P2P applications. Finally, unlike these solutions (with the notable exception of kernel-level techniques) it provides also effective P2P applications scheduling.

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

Interceptor is a software layer, placed on top of the local operating system, that intercepts the resource access requests issued by P2P applications, and controls them, in order to (a) provide application segregation for P2P applications (i.e., enforce quantitative limitations on P2P applications resource usage), and (b) maximize 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 $S(P2P, \tau)$ the maximum share of resource τ (i.e., the fraction of its capacity) that can be allocated to P2P applications, and let $S(P_i, \tau)$ the resource share allocated to the P2P process P_i (henceforth referred to as the *nominal share* of P_i). *Interceptor* schedules the usage requests for τ issued by P2P processes in such a way that the following inequalities simultaneously hold:

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

where $P2P$ is the set of P2P processes running on the node, and $R(P_i, T, \tau)$ is the share of τ received by P_i in the interval going from its creation to time T. In practice, inequality (C_1) states that each P2P process must receive at least its nominal share for resource τ, while inequality (C_2) states that the share globally allocated to P2P applications must not exceed the upper limit on P2P resource usage. $R(P_i, T, \tau)$ is in turn 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 uses the CPU for 10 sec. in an interval lasting 100 sec., then $R(P_i, CPU) = 10/100 = 0.1 = 10\%$.

3.1 Architecture

The architecture of *Interceptor* is schematically depicted in Fig. 1, and encompasses two management modules (the *Creator* and the *Catcher*) and four resource schedulers (one for each of the main node resource types). The *Creator* creates the processes corresponding to P2P applications, upon receiving the corresponding request. Once created, the control of a P2P process is passed to the *CPU Scheduler*, that – by using the mechanisms and policies discussed in the next subsection – schedules it. During its execution, the P2P process requests access to the resources of the node by

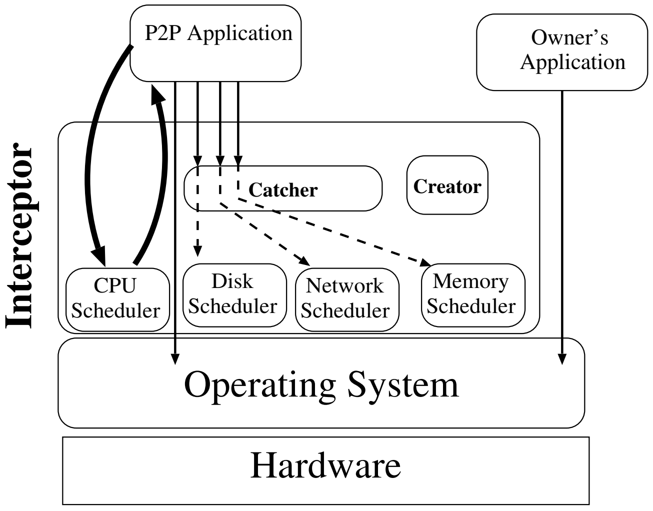


Figure 1. The architecture of Interceptor. Continuos (dotted) thin lines represent issued (intercepted) system calls, thick lines correspond to actions performed bo control process execution.

issuing a set of system calls. These system calls are transparently intercepted by the *Catcher*, that forwards them to the proper resource scheduler. Each scheduler places the requests it receives in a *request queue*, sorted according to a resource-specific scheduling policy, and delays their forwarding to the local operating system 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 system calls that correspond to requests for resource usage issued by P2P applications are intercepted.

3.2 Mechanisms

The consequence of implementing *Interceptor* at the middleware level is that it must co-exist with the local operating systems run on P2P nodes. This in turn 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 used to override the scheduling decisions made by the underlying OS:

- *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 systems, 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.
- *System call interception facilities*, required to transparently mediate resource access. System call interception facilities are available in practically all the modern operating systems (including Linux and various Windows variants).
- *Multi-threading support*, required to implement the various modules of *Interceptor* and to orchestrate their interactions.

Let us explain now, with the help of Fig. 2, how these mechanisms are used to concretely implement the abstract architecture discussed above. As indicated in Fig. 2, each module of the architecture of *Interceptor* corresponds to one or more threads. In particular, the *Catcher* is implemented as a set of threads (one for each P2P process), while all the other modules correspond to a single thread. All these threads, as well as the P2P processes, are scheduled according to the SCHED_RR policy using five priority levels $PRI\_1, PRI\_2, PRI\_3, PRI\_4$, and PRI_5 (where the higher the final number, the higher the priority). The *Creator* thread runs at the highest priority level (PRI_5), 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* does not directly create the process, but instead it starts a new *Catcher Thread* and waits for another creation request, thus blocking itself and allowing another (lower priority) process to be scheduled by the underlying OS scheduler. As soon as it is started, the *Catcher Thread* will create the P2P process (for instance, as indicated in Fig. 2, by means of a $fork()$ system call), will put it into the stopped state by sending a SIGSTOP signal, will *attach* to it in order to perform system call interception, and will place it into the ready-to-run queue of the *CPU Scheduler*. After doing so, the *Catcher Thread* will enter an endless loop in which it waits for the P2P process to issue a new system call, and forwards it to the proper resource scheduler. After process creation, the *Catcher Thread* blocks waiting for the first system call issued by the process, so either the *CPU Scheduler* or one of the other resource schedulers may be scheduled by the underlying OS. In the current implementation, that runs under Linux, the basic ptrace mechanism is used to perform interception, although alternative mechanisms can be used as well.

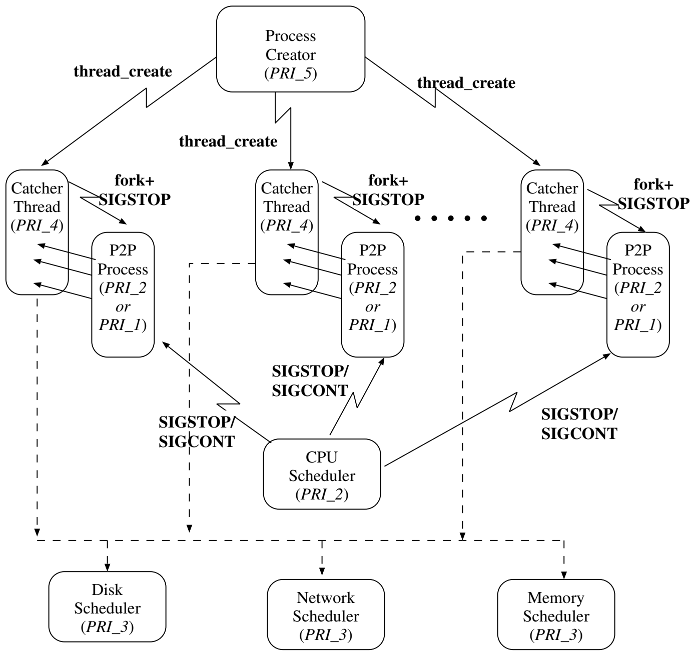


Figure 2. Implementation mechanisms of the Interceptor architecture. Each thread or process is labelled with the corresponding fixed priority value.

The *CPU Scheduler* works by performing an endless loop in which it (a) selects the P2P process 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 running P2P process either back in the ready-to-run queue or into the *I/O-blocked queue*, (e) removes from the I/O-blocked queue those processes that have completed the operation they were waiting for, and (f) suspends itself and all the P2P processes (by means of a SIGSTOP signal) for an amount of time T_{susp} computed as:

$$T_{susp} = \frac{1 - S(P2P, CPU)}{S(P2P, CPU)} * TSD \qquad (2)$$

were TSD denotes the duration of the time slice used by the SCHED_RR policy. T_{susp} is computed as indicated by Eq. 2 in order to enforce inequality (C_2). For instance, if $TSD = 100$ ms. and $S(P2P, CPU) = 0.2$, $T\_susp = (0.8/0.2) * 100 = 4 * 100 = 400$ 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). The *CPU Scheduler* imposes its scheduling decisions to the local OS scheduler by using a variant of the *Dual-Priority Assignment* scheme [9]. This scheme uses the SCHED_RR priority 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 higher 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 a P2P process 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, local OS schedules the P2P process (it is the ready-to-run process with the highest priority), that runs until the current time slice expires, or it issues a blocking I/O operation, whatever event occurs first. When the running P2P process is descheduled by the local OS, the CPU is given back to the *CPU Scheduler* (provided that no other higher priority thread becomes active), that decreases the priority of the P2P process by setting it to PRI_1, and places it either into the ready-to-run or into the I/O-blocked queue.

Let us discuss now the interactions of the *Disk* and *Network Scheduler* with the rest of the *Interceptor*'s threads. Both schedulers are based on the idea of delaying for a suitable amount of time the forwarding of the intercepted system calls to the operating system. Each time a new system call is forwarded by a *Catcher Thread*, the scheduler wakes up, preempts any other running process (it runs at the PRI_3 priority level) and places this call into a request queue sorted according to the particular scheduling policy in use. Both schedulers work by performing an endless loop in which they (a) select the system call at the head of the queue, (b) determine the time instant at which it can be forwarded to the local OS without violating Inequality (c2), (c) sleep until that time and then forward the selected system call. When one of these schedulers wakes up (step (c)), it has a priority higher than any other thread (with the exception of the *Creator*), so it is scheduled immediately by the local OS and can thus forward the selected system call at the proper time instant. Quantitative limitations on P2P network and disk resources usage are enforced as follows. Denoting with N_B and D_B the network and the disk bandwidth (measured in bytes/sec.), respectively, the total network and disk capacities $C_N(T)$ and $C_D(T)$ that can be allocated over an interval lasting T time units are given by:

$$C_N(T) = N_B \cdot T \tag{3}$$
$$C_D(T) = D_B \cdot T \tag{4}$$

Consequently, in an interval lasting T time units, the amount of bytes $C_D^{P2P}(T)$ that can be globally read/written to/from the disk is given by

$$C_D^{P2P}(T) = C_D(T) \cdot S(P2P, Disk), \tag{5}$$

while the amount of bytes $C_N^{P2P}(T)$ that can be sent/received to/from the network is given by:

$$C_N^{P2P}(T) = C_N(T) \cdot S(P2P, Net), \tag{6}$$

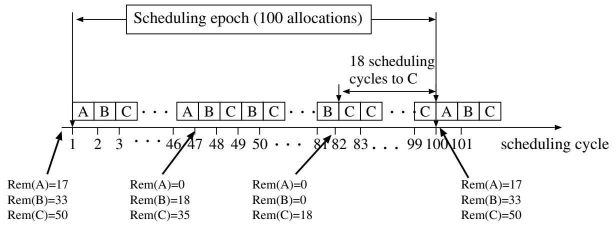
where $S(P2P, Disk)$ and $S(P2P, Net)$ denote the disk and the network share allocated to P2P applications, respectively. Both the *Disk* and the *Network* scheduler keep track of the disk and network capacities used by P2P applications by using discrete time intervals lasting T time units. In each interval the *Disk* (*Network*) scheduler keeps track of the number of bytes globally read/written to disk (sent/received over the network) by P2P applications from the beginning of that interval, and when this byte count exceeds $C_D^{P2P}(T)$ ($C_N^{P2P}(T)$), system call forwarding is delayed until the beginning of the next time interval.

Finally, let us consider 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 will be part of our future 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 with the above nominal share. 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., [19, 27] for the CPU, [25] for the disk, and [18, 28] for the network bandwidth), the ones used by the *Interceptor* have been designed in such a way that they fairly schedule processes whose execution may block for arbitrary amounts of time. These algorithms are based on a policy called *Proportional Number of Slices* (or *PNS* for brevity), that allocates resources in discrete time units, called *allocations*. The duration of each allocation can be either fixed (as done when scheduling the CPU) or variable (as done when scheduling the disk or network bandwidth). Without loss of generality, we will describe *PNS* by assuming allocations of fixed duration and by using CPU scheduling as example.

PNS is based on the idea that in each period lasting C allocations (henceforth referred to as a *scheduling epoch*), a process P_i whose nominal share of 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 $S(P_i, CPU) = 0.5$ then process P_i must receive at least 50 allocations provided that it is always ready-to-run. To precisely allocate to each process its nominal share, a scheduling algorithm could work 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, by decrementing it by one unit each time P_i receives an allocation, by sorting the resource request queue in non-increasing order w.r.t. the $Rem()$ values, and by always running the process at the head of the queue. An example is shown in Fig. 3, where the CPU is

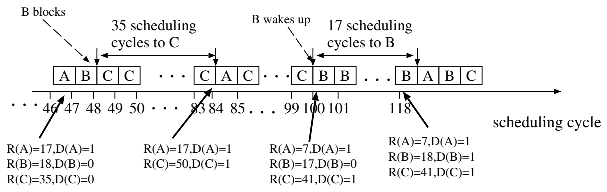


Scheduling epoch (100 allocations)
18 scheduling cycles to C
A B C · · · A B C B C · · · B C C · · · C A B C
1 2 3 · · · 46 47 48 49 50 · · · 81 82 83 . . . 99 100 101
scheduling cycle
Rem(A)=17 Rem(B)=33 Rem(C)=50
Rem(A)=0 Rem(B)=18 Rem(C)=35
Rem(A)=0 Rem(B)=0 Rem(C)=18
Rem(A)=17 Rem(B)=33 Rem(C)=50

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

multiplexed in a round robin way among the three processes until A completes its $Rem(A, CPU)$ allocations (at time 46). From time 47 to 82 the CPU is multiplexed between B and C only, until also B terminates its $Rem(B, CPU)$ 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 $Rem(B, CPU)$ reset to $Av(B, CPU)$ 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 w.r.t. the scheduling epochs completed by the processes issuing the requests (requests of processes with the same $Done()$ values are sorted in non-decreasing order w.r.t. their $Rem()$ values). To exemplify, let us consider again the three processes in Fig. 3. As show in Fig. 4,



B blocks
35 scheduling cycles to C
B wakes up
17 scheduling cycles to B
A B C C · · · C A C · · · C B B . . . B A B C
· · · 46 47 48 49 50 · · · 83 84 85 . . . 99 100 101 118
scheduling cycle
R(A)=17,D(A)=1 R(B)=18,D(B)=0 R(C)=35,D(C)=0
R(A)=17,D(A)=1 R(C)=50,D(C)=1
R(A)=7,D(A)=1 R(B)=17,D(B)=0 R(C)=41,D(C)=1
R(A)=7,D(A)=1 R(B)=18,D(B)=1 R(C)=41,D(C)=1

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

after B blocks (time 48) C gets 35 consecutive allocations, since it has precedence over A (being $Done(C) = 0$) and B is blocked. After C completes its current scheduling epoch (time 84), the CPU is multiplexed between A and C (they both have completed 1 epoch) until time 100, when B wakes up. B receives then the 18 next allocations (i.e., until it complets its first epoch at time 118), and only after that time the CPU is multiplexed again among the three processes.

A similar approach is used to allocate the network and disk bandwidth to P2P process. The most notable difference with the CPU scheduling algorithm lies in the fact that processes that have completed the same number of scheduling epochs are sorted in increasing order w.r.t. the number of bytes read/written to/from disk, or sent/received over the network.

4 Experimental evaluation

In order to assess the viability and effectiveness of *Interceptor*, we have developed a proof-of-concept implementation that runs on top of the Linux 2.6.x kernel, and uses the high-resolution timers provided by the PAPI library [2]. At the moment of this writing, we have completed the implementation of the various threads making up *Interceptor*, but only the PNS-based CPU scheduling policy has been incorporated. Therefore, we report only results concerning the CPU Scheduler.

To assess whether *Interceptor* is able to achieve application segregation and, at the same time, to enable P2P applications achieve satisfactory performance, we performed a thorough experimentation, in which we ran experiments considering a rather large set of workloads, comprising different numbers of simultaneously-running applications of various type. We considered workloads where P2P appli-

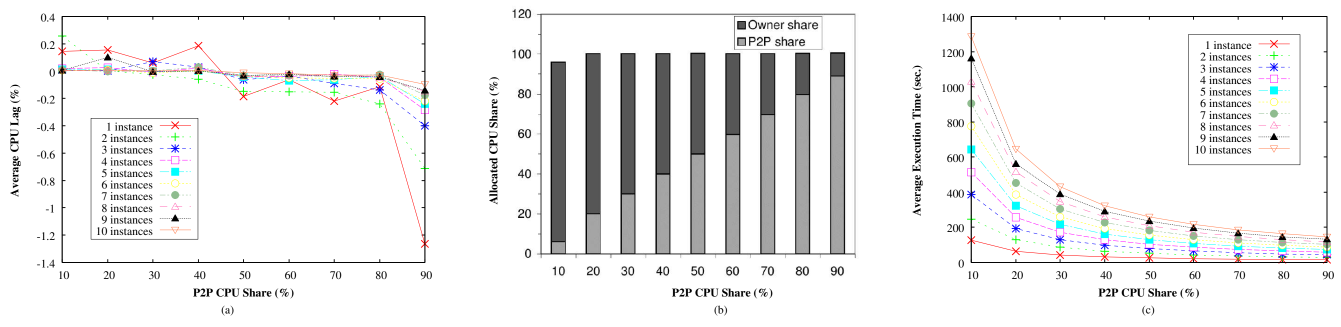


Figure 5. Workload 1 results: (a) Average allocation lag; (b) Share allocated to P2P and owner's applications; (c) Average execution time of EMAP instances.

cations were CPU-bound only, interactive only, and of both types with varying percentages of each type. Unfortunately, space constraints limit us to report only the results concerning one of tha above workloads, namely that comprising CPU-bound P2P applications only. The interested reader may find the results for the other workloads in [4]. The experiments carried out with this workload were aimed at verifying if *Interceptor* is able to (a) allocate to each P2P process *exactly* its specified share, (b) schedule these applications efficiently, and (c) strictly enforce quantitative limitations on CPU usage. We used a P2P workload comprised a set of identical instances of *EMAP* [1], a CPU-bound ElectroMagnetic Analysis Program. Furthermore, to measure the share allocated to owner's processes, we ran a synthetic application that used only the CPU (i.e., it did not perform any I/O operation). We performed a set of experiments in which we progressively increased both the number N of *EMAP* instances from 1 to 10, and the CPU share $S(P2P, CPU)$ globally allocated them from 0.1 to 0.9. In all experiments the nominal share of each *EMAP* instance was set to $S(P_i, CPU) = S(P2P, CPU)/N$. This was done to ensure that the share actually received by any instance is not larger than its nominal share, as consequence of the fact that a set of identical instances receiving the same share will terminate their executions at the same time, thus maintaining a constant CPU demand. In all the experiments we measured (a) the average *allocation lag* (the difference between the share received by an application and its nominal share), (b) the share globally received by P2P and owner's applications, and (c) the average execution time of the *EMAP* instances. The results of our experiments, reported in Fig. 5, show that *Interceptor*'s CPU Scheduler is very fair, as the average lag (Fig. 5(a)) is always very close to 0. Moreover, its application segregation capabilities are also very good, as indicated by Fig. 5(b), that shows that both P2P and owner's applications received their nominal shares (for the sake of brevity only the results measured with 10 *EMAP* instances have been shown, as the ones observed with different instances numbers are practically identical). Finally, as shown in Fig. 5(c), that reports the average execution 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 is directly proportional to the share globally allocated (for instance, with 10 instances the average execution time with a share of 10% (1286.1 sec.) is about 9 times larger than the execution time with a share of 90% (145.7 sec.)), that means that the scheduling overhead per application instance does not depend on the number of instances simultaneously executed on the node.

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. The experiments we carried out with a proof-of-concept implementation running on Linux show that *Interceptor* is able, at least for the CPU (the only resource for which, at the time of this writing, we had a working implementation), (a) to enforce these limitations, thus providing application segregation for P2P applications, (b) to fairly schedule P2P applications (i.e. each P2P application is guaranteed to receive at least its nominal share), and (c) to efficiently schedule P2P applications, that are able to achieve satisfactory performance even in face of the quantitative limitations set by the node owner.

Future work includes the study of memory management mechanisms, available at the user level, in order to develop a better memory scheduler, and the completion of the implementation with the inclusion of the disk and network scheduling policies. Furthermore, in order to reduce the overhead due to the basic system call interception mechanisms adopted in the current implementation, alternative interception mechanisms will be investigated. 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.

Acknowledgements

The author thanks the anonymous referees for their very helpful comments on the first version of this paper.

References

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

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

[3] D. 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.

[4] C. Anglano. Interceptor: Middleware-level Application Segregation and Scheduling for P2P Systems. Technical report, Department of Computer Science, University of Piemonte Orientale, October 2005.

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

[6] S. Balfe, A. Lakhani, and K. 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.

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. 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.

[15] 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.

[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] 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.

[18] K. Jeffay, F. 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.

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.