Access Control for Adaptive Reservations on Multi-User Systems[*]

Tommaso Cucinotta
Scuola Superiore Sant'Anna

E-mail: `cucinotta@sssup.it`

Abstract

This paper tackles the problem of defining an appropriate access control model for multi-user systems providing adaptive resource reservations to unprivileged users. Security requirements that need to be met by the system are identified, and an access control model satisfying them is proposed that also does not degrade the flexibility available on such systems due to the adaptive reservations framework. Also, the implementation of the proposed model within the AQuoSA architecture for Linux is briefly discussed.

1. Introduction

Nowadays, a wide range of time-sensitive applications is gaining momentum, for which a few violations of the designed timing requirements may be acceptable, provided that their frequency and severity is kept in check. Typical examples are multimedia streaming applications, in which timing requirements are important, as their violation is immediately perceived by users as annoyances that degrade the quality of the offered service. Distributed applications for interactive cooperative work, eLearning or interaction through virtual reality, have also similar stringent timing requirements. Even computation-intensive batch activities may be enriched with timing requirements whenever the computational power is made available as a service by a provider, and such activities are embedded within workflows to be carried on within time frames of a few hours or days. In fact, the availability of higher and higher computational power on individual computing systems, along with the availability of higher and higher network bandwidth at cheaper rates, is leading to a growing interest in sharing the same physical system (or pool of interconnected systems) for a multitude of applications, even running remotely *on behalf of different users*, still retaining a minimum set of timing guarantees to individual activities. Typical example scenarios comprise compile factories, remote control of virtual machines, shared multi-purpose servers, web servers for monitoring and configuration of control plants, multimedia streaming (video / VoIP), Service-oriented Architectures (SoAs) with real-time enhancements.

While on typical embedded/dedicated systems the applications that may run are usually under control of the designer, on general-purpose ones this is not true anymore, and users may have the ability to dynamically install custom applications at will, e.g., through a general remote shell/terminal access. Though, whenever *multiple* users may be allocated portions of the same underlying physical machine with certain timing guarantees, issues arise on how to appropriately design access control (AC) models and implement corresponding security mechanisms so that individual users cannot disturb too much activities of other users, according to the security policy a system administrator may want to enforce. In such cases, temporal isolation among concurrently running applications is to be regarded as being as important as the data isolation feature traditionally provided by nowadays operating systems through memory-space isolation, access-control on multi-user filesystems and inter-process protection mechanisms.

After a quick discussion of existing approaches to embedding adaptive real-time technologies within a GPOS like Linux, Section 2 highlights what are, from a security and availability perspective, the critical issues in such approaches, then Section 3 identifies general security requirements that need to be satisfied in such systems, and presents an access-control model that may be implemented at the kernel-level for the purpose of fulfilling such requirements. Finally, Section 4 briefly discusses implementation of the proposed technique on a real Linux-based system.

1.1. State of the art

A quite common approach to adding real-time enhancements to a GPOS kernel is the one of the RTLinux [1] and RTAI [2] projects, where a real-time "executive" is interposed between the Linux kernel and the real hardware. This way it is possible to have a real-time scheduler that schedules real-time tasks, falling back to schedule the Linux kernel and tasks only when no real-time tasks are active. Also, hardware interrupts are managed in the first place by the real-time tasks (through a much lighter routing logics than

[*]This work has been supported by the FP6/2005/IST/5-034026 European Project named FRESCOR.

found in the Linux kernel), and only when no real-time tasks are interested, they are forwarded to the Linux kernel. This results in highly decreased scheduling and interrupt latencies. Main drawback of such approaches is the inability for the real-time tasks to exploit the services offered by the Linux OS and higher level software components (e.g., device drivers and architectures for multimedia).

An alternative approach is the one of the Linux/RK [3] project, a variant of Linux where real-time extensions are directly embedded into the Linux kernel (in terms of capabilities of the CPU and I/O schedulers), for the purpose of adding predictability on its timing behaviour across multiple tasks. This way, it is possible to give proper timing guarantees to standard Linux applications and software components, taking advantage of the full set of capabilities available on the system.

Concerning security issues arising from the availability of real-time extensions within a *multi-user* GPOS like Linux, the RTLinux and RTAI approaches basically neglect the issue, being focused on embedded applications. Infact, on such systems the real-time tasks are basically part of the kernel itself, or they require administration privileges in order to be run, what is perfectly coherent with the hard real-time focus of these projects. Instead, in [4], Rajkumar and Miyoshi explicitly address the problem of avoiding that unprivileged applications affect, intentionally or unintentionally, the timing guarantees granted to other applications by the OS, in a context where multiple resources may be allocated by means of reservation policies to a multitude of applications owned by different users. The authors propose a security mechanism based on a set of Resource Control Lists (RCLs) to be associated to such resources as CPU, network or disk, for the purpose of allowing for both specification of timing guarantees and constraints, and enforcement of system-wide security policies.

While addressing basically similar issues as the ones that motivate the present paper, the RCL approach focuses on the seamless integration of timing guarantees within existing applications at deployment time, in a way that does not need any change to the application code. We believe that such an approach, while being of great interest in the context of system administration and legacy applications, does not manage to exploit to the full extent the benefits of embedding real-time scheduling strategies within the kernel. This may be done, instead, with a tight interaction between the application code, the scheduler and a feedback-based QoS control loop, as done in our prior work [5, 6], based on the AQuoSA QoS management architecture [7]. More importantly, the present paper focuses deeply on the possible menaces arising from exposing to unprivileged applications resource reservation OS capabilities in an untrusted multi-user environment, making an attempt for a systematic enumeration of the security requirements, and it presents one possible access-control model fulfilling them.

Another interesting paper related to the work here presented, is the one recently presented by Tsafrir, Etsion and Feitelson [8], where it is shown that, in most GPOSes, a malicious underprivileged application can utilize nearly 100% of the CPU, regardless of any fairness guarantees supposed to be provided by the scheduler. Also of interest is the security viewpoint of the paper, according to which such "breach" in the fairness property of the scheduler ends up with opening a set of possibilities of attack to a system, where undesired services may be run and steal most of the CPU cycles without this being reported by standard monitoring tools like ps or top. The technique is based on a good knowledge of the time granularity and mechanisms based on which the system scheduler enforces the fairness property, and the system monitoring tools report statistics, and on the availability to the application of a much finer time measurement mechanism such as the CPU cycle counter. Such technique is interestingly related to the presented work as it constitutes a warning for scheduler implementers to not use "poor" time measurement mechanisms, as possible attackers might try to exploit them in order to break the timeliness guarantees and properties that the new scheduler is supposed to achieve (see also Section 3.2).

2. Criticalities in Adaptive Reservations

This section recalls basic properties of the resource reservation scheduling policies, as well as a few technicalities arising in the context of adaptive reservations, that are of relevance for the purpose of ensuring appropriate robustness/security properties for the system. For the sake of brevity, the technical description of the mechanisms is omitted, while the discussion is focused on providing the necessary background for a complete understanding of the security requirements stated in Section 3. The interested reader may refer to [9, 10] for details.

Resource reservation schedulers may allow to associate a single or multiple threads of execution to each reservation, thus they schedule abstract entities usually called *servers*, whereas a scheduler local to a server is responsible for scheduling the various threads within the time frames assigned to the server. The typical guarantee provided by an OS featuring resource reservation scheduling to a server $S^{(i)}$, is that the task(s) associated to the server will be scheduled for *at least* $Q^{(i)}$ time units (whenever in need) within every time window of duration $P^{(i)}$. $Q^{(i)}$ is known as the *maximum budget*, and $P^{(i)}$ as the *period* of the server. This kind of guarantee is also referred to as a *soft* reservation, as opposed to a *hard* reservation, in which a server is instead scheduled for exactly $Q^{(i)}$ time units (whenever in need) and not more, within every time window of duration $P^{(i)}$. For soft reservations, the ratio $\frac{Q^{(i)}}{P^{(i)}}$ is the minimum percentage of resource utilisation guaranteed to $S^{(i)}$, while for hard reservations it constitutes both a minimum guarantee and a maximum constraint on it. Both mechanisms may be implemented on a GPOS, as discussed in Section 1.1, and may allow for coexistence with non-real-time tasks, that are scheduled in background with respect to reserved ones. Schedulers distinguish between a *minimum* budget $Q^{(i)}_{min}$, negotiated at the time of creation of a reservation (used for admission control), that is always granted for each server period (unless all tasks therein are suspended), and an *actual* budget $Q^{(i)} \geq Q^{(i)}_{min}$. This may be granted by the scheduler either statically by distributing the spare resource capacity, or dynamically based on *requested* budgets $Q^{(i)}_{req}$ that may be changed at arbitrary points in time by applications (useful in the context of adaptive reservations).

Note that, for the sake of simplicity, the discussion that follows refers to situations in which admission control may be done through a simple resource utilisation test. This may be as simple as checking that: $\sum_i \frac{Q^{(i)}_{min}}{P^{(i)}} \leq U^{lub}$, where the value of U^{lub} depends on the scheduling algorithm. For example, such test is accurate when servers are scheduled by an underlying EDF scheduler according to their deadlines (with $U^{lub} = 1$), but it may be excessively conservative when servers are scheduled by an underlying FP/RM scheduler (with $U^{lub} = n(2^{1/n} - 1)$, where n is the number of servers), where methods based on response-time analysis may be more accurate.

The server period is a critical parameter, as it constitutes the basic time frame over which the requested utilisation is granted by the operating system. This is usually dictated by the typical period of time-sensitive applications (e.g., 10 or 20 ms for VoIP trans-coding, or 40 ms for video decoding). However, in a context where hard reservations are used, it may be convenient to use a granularity that is a (usually integer) sub-multiple of the application period, for reasons related to jitter control, as shown in [6, 11]. Generally speaking, setting the granularity of the resource allocation to lower values leads to lower latencies and higher interactivity levels. Unfortunately, decreasing more and more the value of even a single server period, the *entire* system undergoes higher and higher scheduling overheads, up to unacceptable values, as shown in [7]. Therefore, for robustness purposes, applications should not be allowed to use reservations with arbitrarily *small* server periods.

In the context of adaptive reservations, each application is allowed to request a percentage of CPU utilisation independently of one another, according to its continuously evolving needs. This allows temporary *overload* conditions, during which the sum of all requests $\sum_i \frac{Q^{(i)}_{req}}{P^{(i)}}$ made asynchronously by all applications overcomes the resource capacity U^{lub}. In such situations, a resource supervisor is allowed to reduce the actual budgets within the limits $Q^{(i)} \in \left[Q^{(i)}_{min}, Q^{(i)}_{req}\right]$. Once the minimum guarantees have been conceded to the servers, various policies are possible for the partitioning of the available capacity to the servers, e.g., a (weighted) fair or priority-based partitioning.

Note that a change of a server scheduling parameters cannot be immediate, but it is realized within a time-frame whose length depends on the adopted scheduling algorithm. However, if the request is for a budget increase that results in a budget decrease for one or more of the other servers, the maximum delay may depend on the *maximum period across all servers* within the system. Furthermore, large server periods usually allow for large budget values as well. This constitutes a potential problem for the responsiveness of the background activities running in the system without guarantees. Therefore, applications should not be allowed to use reservations with arbitrarily *large* server periods.

3. Security Requirements and Access Control

This section first introduces and motivates security requirements in the context of a GPOS with support for adaptive resource reservations, when the system may host applications running on behalf of different users. Then, an access control model is proposed that fulfills the identified requirements, while the next section will briefly discuss how the proposed model has been implemented on Linux.

3.1. Security Requirements

The following is a list of requirements that are needed in order to ensure security, availability and robustness of an adaptive reservation framework that exposes its capabilities to multiple unprivileged, potentially untrusted, users. They will be explained in greater detail in Section 3.2:

1. the possibility for *normal (non-privileged) users* to exploit timeliness guarantees provided by the OS scheduler:
   - for individual threads or processes/applications;
   - for the *entire user session*;
2. system robustness and timeliness guarantees should not be maliciously or non-voluntarily compromisable and should not depend on single applications behaviours;
3. the possibility to run entire applications or user sessions with timeliness guarantees, *transparently* to the user and the applications he/she uses;
4. the availability of appropriate *supervision policies*:
   (a) that do not allow for single users abuses/;
   (b) that allow for a partitioning of the available bandwidth, beyond the total guarantees level, based on *priorities* and *weights* assignable to single users or groups;

5. the possibility for the system administrator(s) to specify appropriate *utilisation bounds* for the reserved applications:
   (a) limit to the total utilisation due to single servers or to the aggregate of a set of servers (quotas);
   (b) the possibility to confine entire sessions within utilisation bounds, transparently to the user and the applications he/she uses;
   (c) the possibility to still run soft real-time applications with timeliness guarantees, even when launching them from within a confined session (*private* server);
   (d) the possibility for the system administrator(s) to *forbid* the use of reservations;
   (e) the possibility for the system administrator(s) to provide such constraints on a per-user or per-group basis;
   (f) the impossibility for user applications to overcome the bounds/limits configured by the administrator(s) through non-conventional or unforeseen patterns of use of the soft real-time functionalities;
6. the ability to create *soft reservation servers*: this allows applications to utilise the system for more than specified by the run-time requested reservation, therefore, depending on the context, a system administrator should be able to selectively allow or deny such possibility for individual users or user groups;
7. servers created by a user should not be manageable by other users (except the privileged one), relatively to the operations of:
   (a) attach or detach of tasks;
   (b) change of the scheduling parameters;
   (c) destruction of the server;
8. privileged users should be able to monitor the entire situation and do any server-related operation they want;
9. the non-real-time tasks should have at least some kind of guarantees, not only in terms of bandwidth, but also in terms of maximum latency ;
10. configurations leading to excessive scheduling overheads should not be allowed;
11. configurations leading to excessive latencies in the dynamic change of reservation parameters should not be allowed;
12. the system should be robust with respect to accidental (or voluntary) crashes of applications ;
13. administration of the AC model configuration itself should be allowed only to privileged users;
14. the overhead due to the enforcement of the access control model should be negligible.

3.2. Access Control (AC) Model

In this section we identify the rules of a security mechanism suitable for satisfying the requirements sketched above. First, the set of objects that may be manipulated by users are identified, along with the set of operations allowed on them. Then, a discussion of the conditions under which each operation is allowed or denied is made.

Discussion is focused on servers that allow to host zero, one or multiple threads of execution, that will be called tasks. Access control is traditionally enforced on GPOSes on the basis of a set of *rules* that are configured both by the system administrator(s) and by users themselves, so also restrictions on the timing requirements that may or may not be granted to users and applications are expected to be specified in terms of rules. Therefore, the objects that may interact raising issues related to access control are: resource reservation servers, tasks, users, user groups and AC rules.

Furthermore, the operations that may be performed on such objects, and that are relevant for the purposes of this paper, are: *management of AC rules, create a server, destroy a server, attach a task to a server, detach a task from a server, change a server parameters, retrieve a server parameters, list tasks attached to a server, list existing servers*.

Management of AC rules For the sake of simplicity, operations related to the management of the AC rules themselves may be thought of as being reserved to the system administrator(s), who have the ability to set-up a static configuration for the system-wide AC rules.

Read-only operations Scheduling parameters are supposed to not carry information that is required to be kept private for users. Therefore, read-only operations (the last three ones enumerated above) are assumed to always be allowed to all users.

Creation of servers Creation of servers gives users the ability to launch applications with scheduling guarantees, that are run prioritarily by the system with respect to non-guaranteed applications. Therefore, it is desirable for system administrators to have the ability to limit this possibility to special users or to special groups of users (Requirement 5d and 5e). Moreover, in order to fulfill Requirement 5a and 5e, the system should allow administrators to specify limits on the maximum CPU utilisation that the system may grant as guaranteed for single servers, for the total set of servers created by a given user, as well as created by all users belonging to a given group.

Therefore, it is envisioned that the AC model allows for the specification of:

- user-level AC rules, where each rule R_k refers to a specific user U_k and whose *scope* S_k is defined as the set of servers that are created by user U_k;
- group-level AC rules, where each rule R_k refers to a specific user group G_k and whose *scope* S_k is defined as the set of servers that are created by any user within the group G_k.

Each rule R_k should include specification of:

- a *maximum minimum guaranteed utilisation* U_k^{maxmin}, that limits the minimum guaranteed utilisation that may be requested for a single server: $\frac{Q_{min}^{(i)}}{P^{(i)}} \leq U_k^{maxmin}$ for all $i \in S_k$; a request for creating a server with a greater minimum utilisation should be rejected;
- a *maximum aggregate minimum guaranteed utilisation* U_k^{aggmin}, that limits the total sum of the minimum guaranteed utilisations granted for all servers in $S_k : \sum_{i \in S_k} \frac{Q_{min}^{(i)}}{P^{(i)}} \leq U_k^{aggmin}$; a request for creating a server that overcomes such condition should be rejected;
- a *maximum aggregate utilisation* U_k^{agg}, that all servers within the scope of the rule are allowed to *occupy* in total: $\sum_{i \in S_k} \frac{Q^{(i)}}{P^{(i)}} \leq U_k^{agg}$; this condition must be enforced by the supervision policy (Requirement 5a) while computing the granted budget values $\{Q^{(i)}\}$ from the actual requests $\{(Q_{req}^{(i)}, P^{(i)})\}$, e.g., through a simple rescaling of all the granted values, occurring whenever $\sum_{j \in S_k} \frac{Q_{req}^{(j)}}{P^{(i)}} > U_j^{agg}$ (for the sake of simplicity, we assume $Q_{req}^{(i)} \geq Q_{min}^{(i)}\, \forall i) : Q^{(i)} =$
$$Q_{min}^{(i)} + \frac{Q_{req}^{(i)} - Q_{min}^{(i)}}{\sum_{j \in S_k}(Q_{req}^{(j)} - Q_{min}^{(j)})}(U_j^{agg} - \sum_{j \in S_k} \frac{Q_{min}^{(j)}}{P^{(j)}})P^{(i)}; \quad (1)$$
- a *maximum aggregate requested utilisation* U_j^{aggreq}, that all servers within the scope of the rule may *request* in total: $\sum_{i \in S_j} \frac{Q_{req}^{(i)}}{P^{(i)}} \leq U_j^{aggreq}$: this is useful in order to avoid that, by maliciously requesting an arbitrarily large request, a server might achieve to "prevaricate" other servers legitimate requests exploiting knowledge of the rescaling supervision algorithm (Requirement 4a); in fact, for example, in Equation 1, we have that $\lim_{Q_{req}^{(i)} \to \infty} \frac{Q^{(i)}}{P^{(i)}} = U_j^{agg} - \sum_{j \in S_k} Q_{min}^{(j)}$, thus a sufficiently large single request may easily occupy the entire utilisation available after the minimum guaranteed values have been assigned.

Note that a zero value for all of the above limits may equally be used, in the rule specification, to selectively forbid creation of servers (Requirement 5d). Also, whenever a server is in the scope of multiple rules, the system should apply all of the constraints, not just one of them. As a last remark, it is noteworthy that, contrary to the commonly seeked behaviour for real-time systems of having soft reservations, in the context of the present paper, enforcement of the mentioned utilisation bounds would require a hard reservation policy. Alternatively, it should be possible to specify, within an AC rule, a flag that allows or forbids the servers within the rule scope to participate to the soft reservation reclamation process.

Finally, if overload situations should be managed through a prioritary and/or weighted partitioning of the granted utilisations exceeding the minimum guaranteed values (Requirement 4b), then the shown formulas should be accordingly changed. Their full description is omitted for the sake of brevity.

In addition to the restrictions on the requested and assigned reservation parameters, applying for classes of servers as described so far, the system should also enforce system-wide restrictions useful to keep below reasonable bounds both the scheduling overhead and the overall responsiveness of the non-reserved activities, as well as the maximum latencies experimented by servers between the request of a change and the actual application of it. Such issues, as addressed by Requirements 10 and 11, are solved by supporting at least the possibility to define minimum and maximum allowed values for the server periods.

Server parameters change and server destruction A user should not be allowed to change parameters of, or to destroy, servers created by other users (Requirement 7b). This is achieved by introducing a concept of *server owner*, i.e., the user who created a server in the first place. Such information needs to be kept by the system at run-time, so to be able to compare the UID of the process requesting a server parameters change with the one of the server owner. On Unix-like systems, for consistency with the in-place AC model, it is recommended that the *effective* user ID (EUID) is used for this purpose. Therefore, whenever a server is created, the system associates the EUID of the process that made the operation along with the other server-related information. Later, whenever a change of parameters or destruction is requested by another process, the operation is only allowed if the EUID of the requesting process is the same as the server owner, or if it corresponds to the system administrator (Requirement 8).

Whenever changing a server parameters, the bounds within which parameters are allowed to be changed, and the relationship between the requested values and the actually granted ones by the supervisor, may follow the same rules as for the server creation. Whenever destroying a server, after destruction tasks previously served by the server, if any, should be returned to the default OS scheduling policy.

Utilisation limit overcome attacks Whenever servers are dynamically created and destroyed by applications, particular attention needs to be paid in order to avoid that a malicious user, by using unexpected patterns of use of the available API, manages to overcome the utilisation bounds configured by the system administrator. At least a couple of scenarios of this kind may be identified: one making use of unexpected patterns of destruction and (re)creation of servers, and another one making use of unexpected patterns of block and unblock of (all of the threads within) a server.

The first potential attack may be detailed as follows:

- an application creates a server and attaches to it one of the running threads;
- when the budget is about to be exhausted, the application destroys the server, returning the attached thread to the default system scheduling policy;
- then, the application immediately creates another server, attaching again the thread to it, and keeps repeating the loop over and over;
- the actions of creating and destroying the server may also be carried on by a thread served by another long-established server with a very small utilisation, just sufficient to keep repeating these few operations.

Unless appropriate countermeasures are taken, a too simple implementation of server destruction might erroneously allow the application to overcome configured utilisation limits. In fact, the key point in such a scenario is that, whenever a server is destroyed, the system cannot (yet) completely forget about its existence, but it must continue to consider its utilisation as not available until the next server period expired[1].

The second potential attack is possible whenever the resource reservation scheduler is implemented relying on an inappropriately large time measurement granularity within the kernel (for example, this has been the case for a few of the very first prototype releases of AQuoSA itself, that were relying on the Linux kernel jiffies for the purpose of accounting budget consumptions):

- an application creates a server and attaches to it one of the running threads;
- when the time granularity is about to expire, the application blocks the thread;
- the application wakes up the thread again;
- block/unblock cycles are repeated over and over again.

The key point in this kind of attack is that a malicious application, letting one of its threads to run for time frames that are below the time granularity of the system, might manage to make the system believe that the thread executed for exactly 0 time units at every block, thus not decreasing the server budget of the time the thread has actual run. Repeating this process multiple times, the application might manage to overcome the utilisation limits that the scheduler should apply to the server.

Such an attack is easily avoidable by recurring to appropriately precise (or at least over-estimated) time measurements and timer setting mechanisms, usually available on modern operating systems (e.g., CPU internal cycle counters, CPU-local or system-wide timer devices, and the High Resolution Timers on Linux).

Attaching and detaching tasks For the purpose of understanding when the operation of attaching or detaching a task to a server should be authorised or not, we need to distinguish among a few cases. In the simplest and probably most common scenario, an application, after having successfully created a server, asks the system to "bind" (or "attach") one or more of its threads to the server. Later, the same application may need to request the system to "unbind" (or "detach") one or more of those threads from the server, either temporarily or definitively. As these operations are made by either the same process/thread, or by processes/threads that are all owned by the same user (EUID, actually), then no particular security issues arise.

On the other hand, whenever an unprivileged user attempts to attach to one of its own servers a process that is owned by another user (or by the system administrator), such action should be prohibited, as it would allow for Denial of Service (DoS) attacks leading to the confinement of threads/processes of other users within the utilisation bounds that have been defined with a different scope. Similarly, any attempt made by processes owned by an unprivileged user to attach threads to, or to detach threads from, a server not owned by the same user, should be prohibited, otherwise such actions would equally allow for DoS attacks. In fact, by attaching threads to a server owned by another user, a malicious user might exploit timing guarantees configured for the other user, possibly overcoming the limits defined in the AC rules for himself/herself, but

[1]Actually, it is sufficient to wait until an earlier time related to the concept of server virtual time, that we skip here for the sake of brevity.

he/she could also disrupt the timing guarantees provided to the other threads already running into that server, that would share the same timing guarantees with more threads than thought of by the application designer. On the other hand, detaching threads of another user from the server they are being served by, a user might simply perform plain DoS attacks versus that other user. Moreover, by detaching own threads from a server owned by another user (e.g., the system administrator), a malicious user might simply try to remove the timing confinement that a system administrator might have configured for the user session, thus it should not be allowed either.

So, apparently, it would be sufficient to allow an attach or detach operation of a thread to/from a server only if the EUID of the requesting process is the same as the server owner, or it corresponds to the system administrator (Requirement 8).

Concerning possible attempts of attachment of threads already bound to another server, it could be sufficient to handle the situation as a request of detach from the original server, followed by the request to attach to the new server.

Also, a desirable feature is the ability to easily "wrap" an entire application or user session within a server (Requirement 3). The key point of such a feature is allowing applications that are bound to a server to automatically attach any child processes or threads to the same server. This way, an entire application may be easily given timing guarantees, or may easily be confined within specific utilisation bounds, by simply attaching the initial thread that launches the application, without any need to deal explicitly with all of the threads or processes, or even external programs, that the application spawns during its execution, because they are bound automatically to the same server.

Default server As system/background activities run usually "in background" with respect to served applications, they are at risk of being "starved" for arbitrarily large periods of time, until all of the threads associated to all reservations are suspended. In order to ensure a minimum level of interactivity, responsiveness or throughput to such activities (Requirement 9), the mechanism that seems most appropriate is the creation of a *default server* within which all applications that do not explicitly request to be scheduled by a resource reservation policy reside by default.

The default server may simply exist after system boot, or it may need to be created by an appropriate privileged system process. In both cases, it is important that the server owner be set as the system administrator (e.g., EUID of 0 on Unix-like systems). This allows only the system administrator to change the default server parameters and, moreover, avoids that an unprivileged user, by manipulating the default server parameters, allows to overcome the utilisation limits that its own servers would be subject to (due to an explicit AC rule) if he/she attempted to create his/her own servers.

Whenever a system is configured with a server to which all tasks are bound by default when not explicitly attached to other servers, AC rules for allowing attach and detach of threads get more involved and the simple two rules stated above are not sufficient anymore. In fact, all attach and detach operations become operations that move threads, respectively, from the default server to the one explicitly referred to by the requesting process, and vice-versa. Therefore, as the default server is owned by the system administrator, no user would ever be allowed to detach any thread from it for attaching them to his/her own server. The problem is fixed by:

- allowing an unprivileged process to detach a thread from the default server *only* if the process is allowed to attach it to the new server, and *only* as a consequence of an attach operation request;
- allowing an unprivileged process to attach a thread to the default server *only* if the process is allowed to detach it from the old server, and *only* as a consequence of a detach operation request;
- forbidding any explicit attempts made by processes owned by unprivileged users to attach threads to the default server, or to detach threads from it.

Private default servers Whenever a system administrator wants to enforce a security policy that assigns specific utilisation bounds to entire user sessions (i.e., not only for reservations explicitly created by a user, but also for all of the processes that may ever run on behalf of the user, as coming from Requirement 5c), a mechanism that seems appropriate is the creation of a *private default server*, i.e., a server dedicated to a user to which all tasks owned by that user are attached by default if not explicitly bound to other servers.

Confinement of an entire user session that does not explicitly make use of resource reservations is straightforward. At log-in time, it is sufficient that a privileged system process creates the private server for the user (if it does not already exists due to an already running session), and starts the user session from a thread initially bound to such server. Note that the server is thus owned by the administrator, not the user, so that the user cannot detach its own threads from it.

Whenever there is a need for allowing users to create their own servers from within such a session, the AC rules for allowing attach and detach of threads, as stated above, are again insufficient. In fact, a user turns to be unable to detach any thread from its private server, for the purpose of attaching it to a new created server. The situation may be

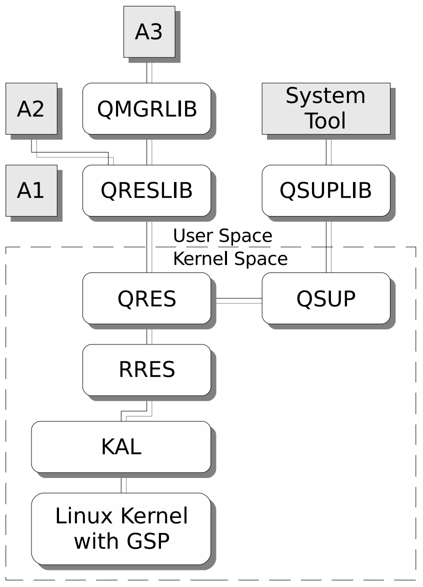

Figure 1. AQuoSA Architecture: applications with static (A2), dynamic (A3) and without (A1) reservations coexist.

fixed similarly to what done for the default server. One last last remark on such issues is due to the dynamic change of a process EUID, as allowed by the standard UNIX seteuid() and related system calls. Detailed discussion of these topics is omitted for the sake of brevity.

4. The AQuoSA supervisor implementation

We implemented the above proposed access control model within the open-source project "*Adaptive Quality of Service Architecture"* (AQuoSA) for the Linux kernel [7, 5]. The project was born for the purpose of improving and maintaining open-source components that had been developed for the Linux kernel in the context of the OCERA EU-funded project. Now, it is being extended and improved in the context of the FRESCOR EU-funded project, with multi-resource, power-aware, QoS management and *security* features. Briefly, the architecture is composed of the following components (see Figure 1):

GSP The Generic Scheduler Patch is a minimally invasive modification to the kernel allowing external dynamically loadable modules to add scheduling policies.

KAL The Kernel Abstraction Layer aims at providing an interface towards the kernel scheduling-related services that are needed in order to write a scheduler: such services comprise the ability to measure time, set timers, efficiently associate external data to tasks, etc.

RRES The Resource RESservation layer implements the basic resource reservation scheduler, along with the supported variants, and makes their services available to further kernel modules through a well-designed and complete kernel-level programming interface (KPI). The RRES layer does not address such issues as admission control or security.

QRES The QRES layer deals with interfacing the RRES functionality with the applications, mediating all the requests through the supervisor component (QSUP).

QSUP This component mediates all the requests made by applications to the kernel so to enforce the security policies configured by the administrator. The QSUP component is responsible for ensuring the correct partitioning of the CPU allocation during overload conditions, so to respect the minimum guarantees promised to applications according to the minimum guaranteed budget values specified at server creation time.

QRESLIB This is the application-level library that makes all the resource reservation functionality available to applications through a well-defined API.

QSUPLIB This is the application-level library that makes all the supervisor configuration functionality available to a special privileged system tool that needs to be run after the activation of the resource reservation modules, so to inject into the system the appropriate security policies configured by the system administrator.

QMGRLIB This library allows applications to use *adaptive* reservations, providing a set of simple QoS controllers that are of general use for applications, and allows for the definition of custom controllers.

With respect to the mechanisms for resource reservations introduced in Section 2, the core AQuoSA scheduler, namely the RRES dynamically loadable kernel module, implements hard resource reservations with multi-tasking capabilities. Also, it embeds a simple soft reservation mechanism that works as follows: if the application enables a specific flag at server creation time (QOS_F_SOFT), then the threads attached to the server are scheduled not only while the server is scheduled by AQuoSA, but also together with the other non-served threads running into the system in the background (when no servers are scheduled by AQuoSA). As compared to traditional mechanisms for soft reservations such as CBS [12], GRUB [9] or IRIS [13], this mechanism allows to mix very well soft real-time applications with background activities, as the latter ones are not completely starved by soft real-time tasks that never suspend.

The supervisor component, composed of the QSUP kernel module and the QSUPLIB library, implements the core functionality realising (partially) the AC model proposed in this paper. Essentially, with respect to the model detailed in Section 3.2, the AQuoSA supervisor lacks only the functionality related to the private default servers, implementing all of the other security mechanisms that have been described. Concerning configuration of the AC rules, the QSUPLIB library allows a privileged system tool (*qsup-admin*) to inject into the kernel AC rules that are statically configured by the system administrator through a configuration file */etc/qossup.conf*. The program *qsup-admin*, to be launched as a privileged process, parses this file, performing the corresponding API calls to the QSUPLIB library. This process is automatically handled by the script that starts the AQuoSA framework (*/etc/init.d/aquosa-qosres*). The same script may be used to restart the AQuoSA service whenever the configuration has been updated (unfortunately this destroys all existing servers, if any).

The availability of a system-wide default server may be either included or excluded through a compile-time configure switch. Also, the default server, whenever available, must be explicitly created through a call to the server creation function, enabling the QOS_F_DEFAULT flag. Such operation is only allowed to be done by a privileged process and is automated, in the init.d script that starts AQuoSA, through an invocation of the *qres* command-line program that exports to the command-line the entire set of available application-level API.

For robustness purposes, AQuoSA embeds a mechanism for automatic destruction of a server as soon as the last task has been detached from it. This allows to free automatically resources reserved by applications that, accidentally or voluntarily, crash, what is a very useful feature while debugging soft real-time applications (Requirement 12). For the particular cases in which such behavior is not desirable, it is possible to request to the system to create servers as *persistent*, setting the QOS_F_PERSISTENT flag in the parameters supplied at server creation time. For security purposes, such possibility may be forbidden for single users or classes of users by the system administrator by setting the same flag in the *forbidden flags mask* that may be specified within an AC rule specification. However, it is planned to slightly modify the mechanism so to allow for a maximum time a server is allowed to exist without attached threads, after which it is destroyed (unless it is persistent). This would be useful to protect against "garbage" servers also with respect to crashes occurring during an application start-up, i.e., between the time an application creates a new server and the time it attaches the first thread.

The same AC rule flags mask may be used to forbid creation of soft reservations, by setting the QOS_F_SOFT flag (Requirement 6).

Concerning the issue of rescaling utilisation requests above the minimum guaranteed values, that overcome the total available capacity, the AQuoSA supervisor configuration allows for specification of multiple *priority levels*. The general idea is allowing for the co-existence of higher priority applications, to be served before lower priority ones in case of overloads. Each AC rule specifies, in addition to the parameters detailed in Section 3.2, a level identifier, with the meaning that the user or entire group of users being addressed by the rule are considered in the scope of that level. Each level may also be associated a total utilisation bound. Basically, starting from a single server, while traversing the AC rules configured for the user that owns the server, the groups the user belongs to, and the level the server owner or owner group belongs to, more and more checks of the total utilisation bounds are performed, along with aggregation of requests made by other servers within the same scope. This gives a system administrator the flexibility to define a prioritary and hierarchical partitioning of the utilisation that is dynamically assigned to servers, covering a wide set of requirements scenarios.

Despite the described mechanism might seem quite complex at a first glance, in order to ensure appropriate scalability with respect to the number of servers, an algorithm has been developed that avoids linear-time re-computation of granted utilisations, what would not have been very appropriate for the so called (and so much advertised) "$O(1)$" scheduler of the $2.6.x$ series of the Linux kernel (note that, in order to implement the EDF low level scheduler the AQuoSA RRES module relies upon, the scheduler has already a logarithmic complexity in the number of servers). This algorithm allows for an $O(L)$ re computation of the utilisation values, with L being the number of actually used AC levels, and is based on the following concept. An accumulator is kept within the supervisor stating, for each aggregation point (user, user group, level), what is the total utilisation (minimum and above minimum) that is currently being assigned to it. These accumulators are incrementally managed in $O(1)$ time, by:

- each time a user creates a new server, adding to them the utilisation increment;
- each time a server is destroyed, deleting from them the utilisation decrement;
- each time a server is changed its utilisation parameters, summing to them the algebraic utilisation difference;

For each of these situations, the change is made by traversing the accumulators starting from the server itself, up to the user, group and level aggregation points (a fixed number of 3 items are traversed for each change). For each one of these accumulators, if the aggregate request of the servers in the current scope exceeds the allowed maximum aggregate utilisation, then a rescaling of the requests is performed. Such rescaling is not done by iterating all of the servers within the current scope, but simply by setting a *rescaling factor* associated to the aggregation point in the hierarchy. This process needs actually to proceed iteratively from the level of the server causing the change, down to the other lower levels (higher levels are of course not affected by the change), what leads to the final $O(L)$ theoretical complexity.

Now, whenever a server needs a budget refill, the actual budget is instantaneously computed by multiplying the server request by the rescaling factors that apply for the aggregation points of that server (a fixed number of operations, carried on in fixed-point precision for speeding up operations and avoiding to use the floating-point unit within the kernel context, as it is highly recommendable). Therefore, the trick is that a rescaling of utilisations due to an overload, that needs the change of budgets of a potentially high number of servers, is reduced to an $O(L)$ operation for the server causing the conditions change, plus an $O(1)$ operation by all of the affected servers at each budget refill time. A few preliminary measurements made on a Pentium M at 1.73 GHz with two levels and a few servers configured showed an overhead for changing the required bandwidth below 0.13 microseconds.

Note also that, due to rounding occurring with management of server utilisations, the supervisor needs a periodic update of the entire set of accumulators and multiply factors, an operation that is not necessarily needed to be done synchronously for all servers. Within AQuoSA, this is going to be handled by a periodic kernel thread, but the mechanism is still under development.

5. Conclusions

This paper addressed various security issues of concern for deploying in a secure, robust and safe manner, adaptive resource reservations in the context of multi-user general-purpose operating systems, where applications taking advantage of soft real-time capabilities of the kernel may be dynamically installed and launched by unprivileged users. After a description of the security requirements of interest, an access control model has been proposed for satisfying them in the context of an adaptive reservation scheduler. Also, it has been described how such issues have been faced with in the implementation of the supervisor component within the AQuoSA architecture for adaptive QoS management on Linux. Most of the requirements identified in this paper showed up as practical issues that needed to be solved during implementation of the AQuoSA components, in order to enhance with appropriate robustness and security properties the provided services.

Probably far from being exhaustive, hopefully the overview made in these few pages may shed some light on how security may be appropriately taken into account while enhancing GPOSes with adaptive reservation technologies, so to build the secure and QoS-aware systems of tomorrow.

References

[1] V. Esteve, I. Ripoll, and A. Crespo, “Stand-alone rtlinux-gpl.” Fifth Real-Time Linux Workshop, November 2003. Department of Computer Engineering, Universidad Politecnica de Valencia, Valencia, Spain.

[2] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, “Rtai: Real time application interface,” *Linux Journal*, vol. 72, April 2000.

[3] S. Oikawa and R. Rajkumar, “Portable RK: A portable resource kernel for guaranteed and enforced timing behavior,” in *Proceedings of the* 5^{th} *IEEE Real Time Technology and Applications Symposium*, pp. 111–120, 1999.

[4] A. Miyoshi and R. Rajkumar, “Protecting resources with resource control lists,” in *Proceedings of the seventh IEEE Real-Time Technology and Applications Symposium*, (Taipei, Taiwan), June 2001.

[5] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, “QoS management through adaptive reservations,” *Real-Time Systems Journal*, vol. 29, March 2005.

[6] L. Palopoli, T. Cucinotta, and A. Bicchi, “Quality of service control in soft real-time applications,” in *Proc. of the IEEE 2003 conference on decision and control (CDC03)*, (Maui, Hawai, USA), December 2003.

[7] T. Cucinotta, L. Palopoli, L. Marzario, and G. Lipari, “AQuoSA – adaptive quality of service architecture,” *to appear in Software – Practice and Experience*, 2008.

[8] D. Tsafrir, Y. Etsion, and D. Feitelson, “Secretly monopolizing the CPU without superuser privileges,” in *Proceedings of the 16th USENIX Security Symposium*, (Boston, MA), August 2007.

[9] G.Lipari and S. Baruah, “Greedy reclaimation of unused bandwidth in constant bandwidth servers,” in *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, (Stokholm, Sweden), June 2000.

[10] L. Abeni and G. Buttazzo, “Hierarchical QoS management for time sensitive applications,” in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, (Taipei, Taiwan), May 2001.

[11] L. Palopoli and T. Cucinotta, “Feedback scheduling for pipelines of tasks,” in *Proc. of 10th conference on Hybrid Systems Computation and Control 2007 (HSCC07)*, (Pisa, Italy), Springer Verlag, Lecture notes in computer science, April 2007.

[12] L. Abeni and G. Buttazzo, “Constant bandwidth vs. proportional share resource allocation,” in *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, (Florence, Italy), June 1999.

[13] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, “IRIS: a new reclaiming algorithm for server-based real-time systems,” in *Real-Time Application Symposium (RTAS 04)*, (Toronto (Canada)), May 2004.