The Capacity Exchange Protocol

Luís Nogueira, Luís Miguel Pinho
IPP Hurray Research Group
Polytechnic Institute of Porto, Portugal
{luis,lpinho}@dei.isep.ipp.pt

Abstract

This paper proposes a new strategy to integrate shared resources and precedence constraints among real-time tasks, assuming no precise information on critical sections and computation times is available. The concept of bandwidth inheritance is combined with a capacity sharing and stealing mechanism to efficiently exchange bandwidth among tasks to minimise the degree of deviation from the ideal system’s behaviour caused by inter-application blocking.

The proposed Capacity Exchange Protocol (CXP) is simpler than other proposed solutions for sharing resources in open real-time systems since it does not attempt to return the inherited capacity in the same exact amount to blocked servers. This loss of optimality is worth the reduced complexity as the protocol’s behaviour nevertheless tends to be fair and outperforms the previous solutions in highly dynamic scenarios as demonstrated by extensive simulations.

A formal analysis of CXP is presented and the conditions under which it is possible to guarantee hard real-time tasks are discussed.

1 Introduction

There is an increasing demand for highly dynamic real-time systems where several independent services with different timing requirements can coexist. Also, it is widely known that in many real-time applications the worst-case execution time (WCET) of some tasks is rare and much longer than the average case. As such, reserving resources based on a worst-case feasibility analysis will drastically reduce resource utilisation, causing a severe system’s performance degradation when compared to a soft guarantee based on average execution times. At the same time, it is increasingly difficult to compute WCET bounds in modern hardware without introducing excessive pessimism [11].

However, if resources are reserved based on average case estimations, it is necessary to prevent a task that needs more than its average execution time to introduce unbounded delays on other tasks’ execution, keeping the overload isolated to a particular task and not jeopardising the schedulability of the other tasks.

Abeni and Buttazo proposed the Constant Bandwidth Server (CBS) scheduler [1] to efficiently handle soft real-time requests with a variable or unknown execution behaviour under the EDF [18] scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task gets a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks.

Several other works [16, 6, 20, 7, 15] have extended this resource reservation approach with the ability to exploit tasks’ earlier completions and reclaim the resulting residual capacities to further increase resource usage and handle soft tasks’ overloads more efficiently.

These ideas are particularly interesting to open real-time systems where several independently developed services can coexist without any previous knowledge about their execution requirements and tasks’ inter-arrival times. Nevertheless, new highly dynamic open real-time systems introduce new requirements and opportunities that are not being completely handled.
Assume a QoS-aware framework that allows cooperation among neighbour nodes to address the increasing demands on resources and performance [21] where it is desirable to be able to process the framework’s management algorithms [23, 22] at a certain minimum rate while previously accepted services are being executed. Overloaded servers dealing with currently executing users’ services should be able to use capacities reserved for the framework’s management, giving them priority with respect to new service requests or QoS adaptations that eventually would bring more workload to the system.

The Capacity Sharing and Stealing (CSS) scheduler [24] was proposed to handle overloads by making additional capacity available from two sources: (i) reclaiming unused allocated capacity when jobs complete in less than their budgeted execution time; and (ii) stealing allocated capacities to non-isolated servers used to schedule sporadic best-effort jobs. CSS offers the flexibility to consider the coexistence of guaranteed and best-effort servers in the same system, reducing isolation in a controlled fashion in order to donate reserved, but still unused capacities to currently overloaded servers, achieving a lower mean tardiness of periodic guaranteed services.

However, tasks were assumed to be independent. A challenging problem in dynamic open real-time systems is how to schedule tasks that share access to some of the system’s resources and exhibit precedence constraints, without a complete knowledge of their behaviour. The purpose of this paper is to address both problems, enhancing CSS with the ability to work in more general real world scenarios. It proposes the Capacity Exchange Protocol (CXP), integrating the concept of bandwidth inheritance [14] with the efficient capacity sharing and stealing mechanism of CSS to mitigate the cost of blocking on soft real-time tasks.

CXP is particularly suitable to schedule soft real-time tasks with precedence constraints in highly dynamic open real-time systems without requiring any offline knowledge of how many services will need to be concurrently executed neither which resources will be accessed and by how long they will be held. The achieved results suggest that CXP effectively minimises the impact of bandwidth inheritance on blocked tasks, outperforming other currently available solutions for open systems.

Although the goal of this paper is to minimise the cost of blocking among soft real-time tasks, hard schedulability guarantees can still be provided even if hard and soft real-time tasks share resources at the expense of some pessimism on the computation of blocking times when tasks access (nested) critical sections.

The rest of the paper is structured as follows.

2 System model

This work focus on dynamic open real-time systems where all services execute on a single shared processor, the sum of the reserved capacities is no more than the maximum capacity of the processor, and the scheduler does not have a complete knowledge about the services’ execution requirements.

A service can be composed by a set of real-time and non-real-time tasks which can generate a virtually infinite sequence of jobs. The jth job of task τi arrives at time sij, is released to the ready queue at time rij, and starts to be executed at time zij with deadline dij = aij + Tij, with Tij being the period of τi. The arrival time of a particular job is only revealed during execution, and the exact execution requirements eij and which resources will be accessed and by how long will be held can only be determined by actually executing the job to completion until time fij. These times are characterised by the relations

\[ a_{ij} \leq r_{ij} \leq s_{ij} \leq f_{ij}. \]

Tasks may simultaneously need exclusive access to one or more of the system’s resources R, during part or all of their execution. If task τi is using resource Ri, it locks that resource. Since no other task can access Ri until it is released by τi, if τj tries to access Ri, it will be blocked by τi. Blocking can also be indirect (or transitive) if although two tasks do not share any resource, one of them may still be indirectly blocked by the other through a third task.

Tasks may also exhibit precedent constraints among them. A task τi is said to precede another task τk if τk cannot start until τi is finished. Such a precedence relation is formalised as τi \prec τk and guaranteed if f ij \leq s k,j. Precedence constraints are defined in the service’s description at admission time by a directed graph G, where each node represents a task and each directed arc represents a precedence constraint τi \prec τk between two tasks τi and τk. Given a partial order \preceq on the tasks, the release times and the deadlines are said to be consistent with the partial order if τi \prec τk \Rightarrow r_{ij} \leq r_{k,j} and d_{ij} < d_{k,j}. Each real-time task τi is associated to a CSS server Qi that is characterised by a pair (Qi, Ti), where Qi is the maximum reserved capacity and Ti is the server period. It is possible to perform an accurate analysis of hard real-time tasks and bound their execution times, minimum inter-arrival times, and the duration of the accessed critical sections and maximum blocking time, then it is possible to find an assignment of Qi and Ti to an isolated server Si that guarantees the schedulability of a hard real-time task, independently of the behaviour of other tasks. Please refer to Section 6 for a detailed analysis.
3 The Capacity Sharing and Stealing approach

The CSS scheduler [24] integrates and extends some of the best principles of previous approaches to efficiently handle soft-tasks’ overloads. It combines the ability to efficiently reclaim allocated capacities that were unused when jobs complete in less than their budgeted execution time with the ability to steal reserved capacities from inactive servers used to schedule sporadic best-effort jobs in overload situations. CSS not only offers the flexibility to consider the coexistence of guaranteed and best-effort servers in the same system, but also to give priority to the overload control of guaranteed services.

Guaranteed and best-effort services are scheduled using two different types of servers. For a guaranteed isolated server, a specific amount of a resource is ensured to be available every period. However, an inactive best-effort non-isolated server, can have some or all of its reserved capacity stolen by active overloaded servers. Non-isolated servers were motivated by the increasing use of imprecise computation models and anytime algorithms in dynamic real-time systems [4, 23, 22].

Both types of servers are characterised by a pair \((Q_i, T_i)\), where \(Q_i\) is the server’s reserved capacity and \(T_i\) its period. Then, the fraction of the CPU reserved to server \(S_i\) (the utilisation factor) is given by \(U_i = \frac{Q_i}{T_i}\).

Extending the traditional parameters of a CBS server [1], each CSS server keeps a specific recharging time \(r_i\) and a pointer to the server from which the reserved capacity is going to be decreased (accounted), eliminating the need of extra queues or additional servers’ states to dynamically manage all the available capacities. CSS’s dynamic budget accounting mechanism ensures that at time \(t\), the currently executing server \(S_i\) is using a residual capacity \(c_r\) originated by an early completion of another active server, its own reserved capacity \(c_i\), or is stealing capacity \(c_s\) from an inactive non-isolated server, using the following rules:

- **Rule A:** Whenever a server \(S_j\) completes its \(k^\text{th}\) job and there is no pending work, its remaining capacity \(c_j > 0\) is released as residual capacity \(c_r = c_j\) that can immediately be reclaimed by eligible active servers, until the currently assigned \(S_j\)'s deadline \(d_{j,k}\). \(S_j\) is kept active with its current deadline.

- **Rule B:** The next server \(S_{i'}\), scheduled for execution points to the earliest deadline server \(S_r\), from the set of eligible active servers with residual capacity \(c_r > 0\) and deadlines \(d_r \leq d_{i,k}\). \(S_i\) consumes the pointed residual capacity \(c_r\), running with the deadline \(d_r\) of the pointed server. Whenever \(c_r\) is exhausted and there is pending work, \(S_i\) disconnects from \(S_r\) and selects the next available server \(S'_{i'}\) (if any).

- **Rule C:** If all available residual capacities are exhausted and the current \(k^\text{th}\) job is not complete, the server consumes its own reserved capacity \(c_i\) either until job’s completion or \(c_i\)'s exhaustion. On a \(c_i\)'s exhaustion, \(S_i\) is kept active with its current deadline \(d_{i,k}\).

- **Rule D:** With pending work and no reserved capacity left, \(S_i\) connects to the earliest deadline server \(S_s\) from the set of eligible inactive non-isolated server with remaining capacity \(c_s > 0\) and deadlines \(d_s \leq d_{i,k}\). \(S_i\) steals the pointed inactive capacity \(c_s\), running with its current deadline \(d_{i,k}\). Whenever \(c_s\) is exhausted and the job has not been completed, the next non-isolated capacity \(c'_s\) is used (if any).

Note that at a particular time \(t\) there is only one server pointing to another server. Also note that, a CSS server suspends its capacity recharging and deadline update until a specific replenishment time \(r_i\), set to the current server’s deadline, implementing a hard reservation (refer to [26] for a description of hard vs soft reservations). This way, an active server which has already exhausted its own reserved capacity can keep its priority both when reclaiming any new spare capacity that may be released and when stealing inactive non-isolated capacities until its original deadline.

A server \(S_i\) is active at instant \(t\) if (i) the served task is ready to execute; (ii) is executing, consuming the pointed capacity; or (iii) the server is supplying its residual capacity to other servers until its deadline. \(S_i\) is inactive if (i) there are no pending jobs to serve; or (ii) the server has no residual capacity to supply to the other servers.

State transitions are determined by the (i) arrival of a new job, (ii) capacity exhaustion, or (iii) non-existence of pending jobs at replenishment time. An inactive server becomes active with the arrival of the new \(j^{\text{th}}\) job at time \(a_{i,j}\), if \(a_{i,j} \geq d_{i,j-1}\). If \(a_{i,j} < d_{i,j-1}\), the job is only released at the next \(S_i^{'s}\) replenishment instant \(r_i\). On the other hand, an active server becomes inactive if (i) all its reserved capacity is consumed and there are no pending jobs to serve (capacity exhaustion can occur while supplying its residual capacity to other servers or using its capacity to finish a job); or (ii) there are no pending jobs at replenishment time. At replenishment time \(r_i\), unconsumed capacities are discarded.

The increased computational complexity of fairly assign residual capacities to all active servers and the fact that fairly distributing residual capacities to a large number of servers can originate a situation where no enough excess capacity is provided to any one to avoid a deadline miss, lead us to assign all residual bandwidth to the currently executing overloaded
server (Rule B). Furthermore, since jobs’ execution requirements are not known beforehand, it makes sense to devote as much excess capacity as possible to the currently executing server, maximising its chances to complete the current job before its deadline. This policy has already been proved to minimise deadline postponements and the number of preemptions [16].

When the reserved capacity of server \( S_i \) is exhausted and there is still pending work, \( S_i \) is allowed to steal inactive non-isolated capacities to handle its overload (Rule D). However, capacity stealing is interrupted whenever \( S_i \) is preempted or a replenishment event occurs on the capacity being stolen. Also, since \( S_i \) keeps its current deadline \( d_{i,k} \geq d_s \) when stealing non-isolated capacities, capacity stealing is also interrupted when a new job for the inactive non-isolated server \( S_s \) arrives. When this happens, \( S_s \) becomes active with its current remaining capacity. Note that all active (isolated and non-isolated) servers can reclaim residual capacities and steal inactive non-isolated capacities.

### 3.1 Handling overloads with CSS

The next example details how CSS can handle soft tasks’ overloads without postponing deadlines, either by greedily reclaiming residual capacities or stealing inactive non-isolated servers’ capacities used to schedule sporadic best-effort services. It has been demonstrated [24] that this approach can reduce the mean tardiness of periodic guaranteed services in systems where guaranteed and best-effort services can coexist.

Consider the following periodic task set, described by average execution times and period: \( \tau_1 = (2, 5), \tau_2 = (4, 10), \tau_3 = (3, 15) \). \( \tau_1 \) is served by the non-isolated server \( S_1 \), while tasks \( \tau_2 \) and \( \tau_3 \) are served by the isolated servers \( S_2 \) and \( S_3 \), respectively. A possible CSS’s scheduling of this task set is detailed in Figure 1. When a server is using a residual or stolen capacity from another server a pointer indicates where the budget accounting is being performed.

![Figure 1. Handling overloads with CSS](image)

At time \( t = 3 \), \( \tau_2 \) has an early completion and a residual capacity \( c_r = 1 \) with deadline \( d_r = 10 \) is available. Server \( S_3 \) is scheduled for execution and connects to the earliest deadline residual capacity available of server \( S_2 \). \( \tau_3 \) consumes \( c_r = 1 \) before starting to consume its own capacity at time \( t = 4 \). At time \( t = 7 \), an overload of \( \tau_3 \) is handled by \( S_3 \), stealing capacity from the inactive non-isolated server \( S_1 \). A new deadline for the stolen capacity \( c_s \) is set to time \( t = 12 \).

Note that at time \( t = 9 \) a new job of \( \tau_2 \) arrives but the job is only released at time \( t = 10 \). Advancing execution times is against our purpose of executing periodic activities with stable frequencies.

At time \( t = 15 \), after \( S_2 \) completes its job by stealing some of the inactive non-isolated capacity of \( S_1 \), a new job for server \( S_1 \) arrives. \( S_1 \) reaches the active state, keeping its currently available capacity and corresponding deadline.

At time \( t = 16 \), server \( S_1 \) has no remaining capacity and stops executing. At time \( t = 19 \), a replenishment of \( S_1 \)’s capacity occurs and it continues to execute the pending job, releasing the residual capacity \( c_r = 1 \) with deadline \( d_r = 24 \) at time \( t = 20 \) when it completes its job’s execution. This residual capacity is used by server \( S_2 \) before consuming its own capacity at time \( t = 21 \).

At time \( t = 25 \), a new job of \( \tau_1 \) arrives and the non-isolated server \( S_1 \) becomes active. It first consumes the residual capacity \( c_r = 1 \) with deadline \( d_r = 30 \), generated at time \( t = 24 \) by an early completion of \( \tau_2 \), before consuming its own capacity.

At time \( t = 33 \) an overload of \( \tau_2 \) is first handled by stealing capacity of the inactive non-isolated server \( S_1 \) and then, at time \( t = 38 \), consuming the available residual capacity generated by an early completion of task \( \tau_3 \). Recall that a server remains active until its deadline, even if it has exhausted its capacity.
4 Sharing resources in open systems

A great amount of work has been addressed to minimise the adverse effects of blocking when considering shared resources among tasks. Resource sharing protocols such as the Priority Ceiling Protocol [28], Dynamic Priority Ceiling [9], and Stack Resource Policy [2] have been proposed to provide guarantees to hard real-time tasks accessing mutually exclusive resources. Solutions based on these protocols were already proposed [13, 8, 7, 3] but they all require a prior knowledge of the maximum resource usage for each task and cannot be directly applied to dynamic open real-time systems.

A resource sharing protocol that is independent from the actual tasks’ requirements was proposed in [14]. The Bandwidth Inheritance (BWI) protocol allows a shared access to resources without requiring any prior knowledge about the tasks’ structure and temporal behaviour and guarantees that tasks that do not access shared resources are not affected by the behaviour of other tasks. It extends the CBS scheduler to work in the presence of shared resources, adopting the Priority Inheritance Protocol (PIP) [28] to handle task blocking. Although the PIP was initially thought in the context of fixed priority scheduling, it can be applied to dynamic priority scheduling, holding its basic properties: it limits the worst-case blocking that must be endured by a job $j$ to the duration of at most $\min(n, m)$ critical sections where $n$ is the number of jobs with lower priority than $j$ and $m$ the number of different semaphores used by $j$.

However, the main drawback of BWI is its unfairness in bandwidth distribution. A blocking task can use most (or all) of the reserved capacity of one or more blocked tasks, without compensating the tasks it blocked. Blocked tasks may then lose deadlines that could otherwise be met. At the same time, servers keep postponing their deadlines and recharging their capacities on every capacity exhaustion, potentially severely delaying blocked tasks with earlier deadlines which will finish later than tasks with longer deadlines. It is known that allowing a task to use resources allocated to the next job of the same task may cause future jobs of that task to miss their deadlines by larger amounts [24, 15]. This violation of the original capacity distribution can have a huge negative impact in the overall system’s performance.

Figure 2 illustrates these problems with a simple example. Three servers $S_1 = (2, 5)$, $S_2 = (1, 3)$, and $S_3 = (1, 4)$ serve three tasks with execution times equal to their respective servers’ capacity. Tasks $\tau_1$ and $\tau_2$ share access to resource $R$ for the duration of their execution times, while $\tau_3$ is independent from the other two.

Note how an early arrival of the second job of task $\tau_1$ at time $t = 4$ allows $\tau_1$ to consume 3 units of execution in the interval $[0, 5]$, more than its initial reservation. The nonexistence of a compensation mechanism and the automatically deadline update are responsible for the deadline miss of the second job of task $\tau_2$.

To address this issue, BWE [31] and CFA [27] integrate bandwidth compensation mechanisms into BWI, trying to fairly compensate blocked servers in exactly the same amount of capacity that was consumed by a blocking task while executing in a blocked server. To achieve this, BWI maintains a global $n \times n$ matrix ($n$ is the number of servers in the system) in order to record the amount of budget that should be exchanged between servers, a budget list at each server to keep track of available budgets, and dynamically manages resource groups at each blocking and releasing of a shared resource. CFA requires each server to manage two task lists with different priorities and a counter that keeps track of the amount of borrowed capacity from a higher priority server, converting the inheritor into a debtor. Contracted debts are payed by blocking servers, until the blocked servers’ counters are successively decremented to zero.
The increased computational complexity of these attempts to fairly compensate borrowed capacities and the fact that CSS tends to fairly distribute residual capacities in the long run [24], lead us to propose an efficient capacity exchange protocol that merges the benefits of a smart greedy capacity reclaiming policy with the concepts of bandwidth inheritance and hard reservations. Adding to the lower complexity of our approach, achieved results detailed in Section 7 demonstrate that taking advantage of all of the available capacity instead of only exchanging capacities within the same resource group leads to a better system’s performance in dynamic open real-time systems.

4.1 The Capacity Exchange Protocol

With CXP, every server maintains a list of served tasks ordered by tasks’ deadlines. A task can then be added to more than one task list and be executed on more than its dedicated server. Initially, each server has only its dedicated task in its task list and, as long as no task is blocked, servers behave as in the original CSS scheduler. With blocking, the following rules are introduced:

- **Rule E**: When a high priority task \( \tau_i \) is blocked by a lower priority task \( \tau_j \), when accessing a resource \( R \), \( \tau_j \) is inherited by server \( S_i \). The execution time of \( \tau_j \) is now accounted to the currently pointed server by \( S_j \). If task \( \tau_j \) has not yet released the shared resource \( R \) when \( S_j \) exhausts all the capacity it can use, \( \tau_j \) continues to be executed by the earliest deadline server with available capacity that needs to access \( R \), until \( \tau_j \) releases \( R \).

- **Rule F**: If a blocking task \( \tau_j \) is inherited by a blocked server \( S_j \), delaying the execution of task \( \tau_j \), then \( \tau_i \) is also added to \( S_j \)’s task list. When task \( \tau_i \) is unblocked it is executed by the earliest deadline server which has \( \tau_i \) in its task list until it is finished or the server exhausts all the capacity it can use (whatever comes first).

- **Rule G**: If at time \( t \), no active server with pending jobs can continue to execute using some of the rules B, C, or D, and there is at least one active server \( S_r \) with residual capacity greater than zero, available residual capacities with deadlines greater than the one assigned to the current job \( j_{p,k} \) of the earliest deadline server \( S_p \) with pending work can be used to execute \( j_{p,k} \) through bandwidth inheritance.

Rule E describes the integration of the bandwidth inheritance mechanism in the dynamic budget accounting of CSS. The currently executing server always consumes the pointed capacity, either its own or another available and valid capacity in the system.

Rule F allows a blocked task \( \tau_i \) that has been delayed in its execution to be executed by the earliest deadline server with available capacity which has \( \tau_i \) in its task list, that may now be different from \( S_i \). Note that capacity exchange due to blocking is performed without the goal of a fair compensation, reducing the complexity and overhead of CXP.

In general, the hard reservation approach may cause the loss of more deadlines since once a server’s capacity is depleted capacity recharging is suspended until the server’s next activation. To minimise its drawbacks and take advantage of a more constant rate in tasks’ execution Rule G allows the use of bandwidth inheritance to execute unfinished tasks, including those from servers that do not directly or indirectly share any resource with the selected server, if at a particular time no active server in the system is able to reclaim new residual capacities or steal inactive non-isolated capacities to continue executing its pending work after a capacity exhaustion.

Since the queue of active servers is ordered by deadlines, CXP easily keeps track of the earliest deadline server with pending work and no capacity left \( S_p \) as well as the earliest deadline server with available residual capacity \( S_r \) when traversing the queue to select the next running server. If the end of the queue of active servers is reached without finding a server with pending work and available capacity, server \( S_r \) is selected as the running server and inherits the first task of \( S_p \)’s list and executes it consuming its own residual capacity. Since a server always starts to consume the earliest residual capacity available, no modification to the budget accounting mechanism is needed to correctly account for the consumed capacity.

Note that Rules A and B of the original CSS scheduler ensure that residual capacities originated by earlier completions can be reclaimed by any active eligible server. Blocked servers can then take advantage of any residual capacity, even if it is released by a server that does not share any resource with the reclaiming server. Furthermore, this residual capacity tends to be reclaimed in a fair manner among needed servers across the time line [16, 24].

Figure 3 illustrates how CXP can minimise the cost of blocking and exchange capacity between servers, scheduling the same set of tasks used to analyse the BWI’s drawbacks in Figure 2.

At time \( t = 1 \), task \( \tau_2 \) is added to server \( S_1 \)’s task list (Rule F). At time \( t = 2 \), task \( \tau_2 \) is unblocked and is executed by server \( S_1 \), since it is the earliest deadline server with remaining capacity with \( \tau_2 \) in its task list (the same happens at time
Figure 3. Sharing resources with CXP

Note that despite the earlier arrival of task τ₁’s second job at time \( t = 4 \), \( S_1 \)'s deadline is not set to \( d_{1,2} = 9 \) and the job is only released at time \( t = 5 \), implementing a hard reservation (see Section 3 for details). Also note that capacities are exchanged between all the system’s servers and not only within a specific resource group, maximising the use of extra capacities to handle overloads and still meet deadlines. An overload of the independent task \( τ₃ \) was handled by reclaiming the residual capacity originated by an earlier completion of task \( τ₁ \) at time \( t = 12 \).

While preserving the isolation principles of independent tasks and inheritance properties of critical sections of BWI, CXP introduces significant improvements in the system’s performance by efficiently exchanging capacities between hard reservation servers. Since the execution and inter-arrival times of jobs are not known in advance it is important to minimise the impact of misbehaved tasks that exceed their expected execution times or have a shorter inter-arrival time of jobs.

5 Handling precedence constraints in open systems

Additional constraints on real-time systems arise when the execution of the data’s producer must precede the execution of the consumer of that data. Such precedence constraints may affect the system’s schedulability, and in more complex scenarios, both shared resources and precedence constraints can be present among the tasks.

It is well known that precedence constraints can be guaranteed in real-time scheduling by priority assignment. In fact, with dynamic scheduling, any task will always precede any other task with a later deadline. This suggests that precedence constraints that are consistent with the tasks’ deadlines do not affect the schedulability of the task set. In fact, the idea behind the consistency with the partial order is to enforce a precedence constraint by using an earlier deadline.

Formal work exists, showing how to modify deadlines in a consistent manner so that EDF can be used without violating the precedence constraints. Garey et al. [12] show that the consistency of release times and deadlines can be used to integrate precedence constraints in the task model. Spuri and Stankovic [30] introduce the concept of quasi-normality to give more freedom to the scheduler so that it can also obey shared resource constraints, and provide sufficient conditions for schedules to obey a given precedence graph. The authors prove that with deadline modification and some type of inheritance it is possible to integrate precedence constraints and shared resources. Mangeruca et al. [19] consider situations where the precedence constraints are not all consistent with the tasks’ deadlines and show how schedulability can be recovered by considering a constrained scheduling problem based on a more general class of precedence constraint.

However, all these works base their modifications of deadlines on a previous knowledge of the tasks’ execution times. To make use of these previous results in open real-time systems, the consistency of release times and deadlines with the partial order must be enforced considering estimated execution times when applying some known technique [12, 29, 19, 5, 10] at admission time.

Such an approach immediately raises two questions: (i) what happens if a precedent task requires more capacity than declared? (ii) how can a task know if its predecessors have already finished? CXP provides answers for both questions and can be used to handle blocking due to precedence violations in the same way as for a critical section blocking, minimising the impact of misbehaved tasks on the overall system’s performance. We base our approach on the idea that if task \( τ_j \prec τ_i \),
has not yet finished at time $s_{i,k}$, when the $k^{th}$ instance of $\tau_i$ is selected to execute, it is blocking its successor.

Given a partial order $\prec$ on the tasks, described by a directed graph $G$, servers’ state changes in CXP allow an easy verification of the current condition of a precedent task $\tau_j$. Recall that a server that has completed its job is only kept active until its deadline if it is supplying some residual capacity originated by an earlier completion of its previous job.

- **Rule H:** If a precedent server $S_j$ is active at time $s_{i,k}$, $S_i$ checks the current value of $S_j$’s residual capacity. If its equal to zero, then $\tau_j$ has not yet been completed and must be added to $S_i$’s task list.

Note that a server that is scheduled for execution already checks the current state of the residual capacity of earlier deadline servers as it tries to consume them before its own reserved capacity and precedence constraints are handled as an access to a shared resource.

The next example illustrates how CXP allows an easy integration of precedence constraints among tasks without requiring any previous knowledge of their exact computation times. Figure 4 shows a possible scheduling of three servers $S_1 = (2, 8)$, $S_2 = (4, 10)$, and $S_3 = (3, 15)$ used to serve three tasks, based on their estimated average execution times and periods, that exhibit the precedence constraints $\tau_1 \prec \tau_2 \prec \tau_3$.

Figure 4. Handling tasks’ precedences with CSS

Time $t = 3$ illustrates the situation where the successor server knows it has to complete its predecessor’s task. Since $S_1$ is still active and its residual capacity is set to zero, task $\tau_1$ has not yet been completed and must continue to be executed in $S_2$ prior to $\tau_2$’s execution.

On the other hand, at time $t = 6$ and $t = 10$, servers can start executing their dedicated tasks. At time $t = 6$, $S_2$ becomes inactive by completing $\tau_2$ and exhausting its capacity. Its inactive state clearly indicates that task $\tau_2$ has been completed and $S_2$ is not able to supply any residual capacity to other servers. At time $t = 10$, however, the predecessor server $S_1$ is active but with residual capacity available. This is only possible when a server has completed its current task using less that its budgeted capacity.

6 Theoretical validation

Demanding that soft real-time (SRT) tasks declare the maximum duration of the critical sections on each accessed resource at admission time is against the basic purpose of an open system itself. However, hard real-time (HRT) tasks still need to be guaranteed based on the knowledge of their worst-case behaviour.

One way to achieve this in an open system is to implement the critical sections as library functions whose WCET can be determined. Of course, this comes at the cost of some pessimism but a server for a HRT tasks must always have a reserved capacity based on the worst case.

If nested critical sections are allowed, the system’s libraries must also impose a totally ordered access to resources, since for a deadlock to be possible, a blocking chain must exist in which there is a circular relationship. Furthermore, deadlocks can be detected and exceptions raised if a misbehaving task attempts to acquire resources in an improper order, by following the chain of accessed resources and detecting a resource that is already in the list.

On the other hand, when considering precedence constraints, guarantees to HRT tasks can only be provided in sets of HRT tasks, since the WCET of SRT tasks cannot be determined offline.
In the remaining of this section, we assume that resources are orderly accessed through shared libraries and discuss how to assign the maximum capacity $Q_i$ and period $T_i$ to an isolated server which has to serve a hard real-time task in an open system with $n$ hard reservation servers with a total utilisation of $\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$. We start by proving the correctness of the proposed capacity exchange mechanism of CXP.

**Definition 1** At a particular time instant $t$, the total amount of available system’s execution capacity $C_a$ is the sum of the remaining reserved capacities greater than zero that can be used to execute a task (either the remaining execution or residual capacities of active servers or the remaining execution capacities of inactive non-isolated servers whose capacity can be stolen by active servers).

**Lemma 1** Just after a task $\tau_i$ releases the shared resource $R$, the total amount of available system’s execution capacity $C_a$ is the same as in the non-resource sharing case.

**Proof**
While task $\tau_i$ is accessing the shared resource $R$ during $t$ units of time, it can block some other task. It follows from the bandwidth inheritance protocol that when a task blocks another one it inherits the latter’s server. CXP’s dynamic budget accounting mechanism guarantees that while a server is executing any task from its task list, it is consuming the capacity of the currently pointed server. This may be a residual capacity originated by an early completion of some other task, its own reserved capacity, or a stolen inactive non-isolated capacity.

Hence, the total amount of available system’s execution capacity $C_a$ when task $\tau_i$ releases the shared resource $R$ is independent of whether the task was executed only by its dedicated server $S_i$ or not. In the worst case, the longest time a server can be connected to another server is bounded by the currently pointed server’s capacity and deadline.

□

**Lemma 2** No capacity is exchanged after its deadline

**Proof**
Let $a_{i,k}$ denote the instant at which the $k^{th}$ instance of task $\tau_i$ arrives and its dedicated server $S_i$ is inactive. At $a_{i,k}$, a new execution capacity $c_i = Q_i$ is generated. If $S_i$ is a non-isolated server and some amount $c_s$ of its reserved capacity was stolen while it was inactive, the server becomes active with the remaining execution capacity $c_i = Q_i - c_s$.

Let $\forall i,k d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$ be the deadline $\forall i,k r_i = d_{i,k}$ be the replenishment time associated with capacity $c_i$.

Let $L$ be the task list of server $S_i$. $L$ is composed at least by jobs of task $\tau_i$, but can also contain, due to blocking, inherited tasks (Rule E) and tasks that were delayed by the execution of $\tau_i$ in high priority servers (Rule F).

Let $[t, t + \Delta_t]$ denote a time interval during which server $S_i$ is executing the earliest unblocked task of $L$, consuming its own reserved capacity $c_i$. Consequently, $S_i$ has used an amount equal to $c_i' = c_i - \Delta_t \geq 0$ of its own capacity during this period. As such, $c_i$ must be decreased to $c_i'$, until its value is equal to zero.

Let $f_{i,k}$ denote the time instant when server $S_i$ completes the last job of $L$. The remaining execution capacity $c_i > 0$ is released as residual capacity $c_r = c_i$ and $c_i$ is set to zero.

At instant $f_{i,k}$, the next active server $S_j$ with pending work and remaining execution capacity is scheduled for execution, according to the EDF policy. If the inequality $d_{i,k} \leq d_{j,l}$ holds, server $S_j$ can use the released residual capacity $c_r$ until its deadline $d_{i,k}$ or $c_r = 0$.

Let $[t, t + \Delta_t]$ denote a time interval during which server $S_j$ is executing, consuming the residual capacity $c_r$. Consequently, $S_j$ has used an amount equal to $c_r' = c_r - \Delta_t \geq 0$ of $S_i$’s residual capacity during this period. As such, $c_r$ must be decreased to $c_r'$, until its value is equal to zero.

If at some instant $t$ all active servers have exhausted the amount of execution capacities they can use and there are unfinished jobs, the job of the earliest deadline unfinished task $\tau_o$ is added to the task list of the earliest deadline active server $S_i$ with residual capacity $c_r > 0$. Assume that $S_i$ is the selected server.

Let $[t, t + \Delta_t]$ denote a time interval during which server $S_i$ is executing, consuming its own residual capacity $c_r$. Consequently, $S_i$ has used an amount equal to $c_r' = c_r - \Delta_t \geq 0$ of its residual capacity during this period. As such, $c_r$ must be decreased to $c_r'$, until its value is equal to zero.

At replenishment time $t = r_i$, any remaining unused residual capacity $c_r$ of server $S_i$ is discarded and $c_r$ is set to zero.
Theorem 1 Given a system with $n$ servers with utilisation $U = \sum_{i=1}^{n} \frac{Q_i}{T_i}$ which uses CXP for accessing shared resources, it can be guaranteed that, at any time, the system’s utilisation $U$ is no more than the case when the served tasks do not share access to some resources.

Proof

Without resource sharing, CXP ensures that no server consumes more than its reserved capacity $Q_i$ every period $T_i$ and the amount of capacity that can be reclaimed or stolen is limited in the worst case by the reserved capacity and deadlines of the pointed servers. By directly applying the results of Lemma 1 and Lemma 2, the same properties hold in CXP when tasks share access to resources.

Theorem 2 A blocked task scheduled by CXP never has less available time to complete its execution than under the basic BWI protocol

Proof

From Rule F, CXP guarantees that a blocked task $\tau_i$ resumes its execution in the earliest deadline server which has $\tau_i$ in its task list, which may be different from its dedicated server $S_i$. With BWI, however, the blocked task $\tau_i$ is only able to resume its execution when its dedicated server $S_i$ has no more blocking tasks in its task list and is the earliest deadline among all active servers.

As a consequence, the time that is available for a blocked task $\tau_i$ to complete its execution may be increased with CXP but never reduced when compared against BWI.

After proving the correctness of the capacity exchange mechanism of CXP, we now discuss how to provide guarantees to hard real-time tasks starting with some important definitions that help to clarify the following analysis.

Definition 2 Two tasks are in the same resource group $G$ if they directly or indirectly share some resource.

Definition 3 Given a task $\tau_i$ served by server $S_i$, the blocking time $B_i$ is defined as the maximum time that all other tasks can be executed by $S_i$, for each job of $\tau_i$.

Lemma 3 Given a task $\tau_i$ served by server $S_i$, only tasks in the same resource group $G$ can be added to $S_i$’s task list and contribute to $B_i$, for each instance of $\tau_i$.

Proof

Initially, each active server has exactly one task in its task list. It follows from the bandwidth inheritance protocol that if a task $\tau_i$ is blocked by task $\tau_j$ when accessing a resource $R$, then $\tau_j$ is added to the task list of server $S_i$. If $\tau_j$ is also blocked on another resource, the chain of blocking is followed and all the blocked tasks are added to $S_i$ until a non-blocked task is reached. The task list of all other servers remains unchanged. Hence, the number of tasks that can contribute to $B_i$ is restricted to those tasks that belong to the same resource group $G$.

Theorem 3 If a HRT task $\tau_i$ is served by an isolated server $S_i$ with parameters $(Q_i, T_i)$, where the reserved capacity $Q_i = C_i + B_i$ is determined by adding the WCET $C_i$ of $\tau_i$ to the maximum blocking $B_i$ that can be experienced by an instance of $\tau_i$, and $T_i$ is the minimum inter-arrival time of $\tau_i$’s jobs, then $\tau_i$ will meet its deadline, regardless of the behaviour of the other tasks in the system.

Proof
From Theorem 1 it follows that each isolated server \( S_i \) always receives \( Q_i \) units of execution capacity every \( T_i \) units of time. Lemma 3 assures that the set of tasks that can be executed by \( S_i \) is restricted to those tasks in the same resource group \( G \). Hence, if a HRT task \( \tau_i \) does not access any shared resource it is not affected by the behaviour of other tasks. Therefore, if each instance of \( \tau_i \) consumes up to \( C_i \leq Q_i \) units of execution capacity and instances are separated at least by \( T_i \), is is guaranteed that task \( \tau_i \) finishes no later than \( S_i \)'s capacity exhaustion and it will meet all its deadlines.

If a HRT task \( \tau_i \) accesses some shared resources during its execution, we have to consider the maximum time that other tasks can be executed by \( S_i \) through bandwidth inheritance. It follows from Lemma 3 that whether task \( \tau_i \) meets its deadline depends only on the timing requirements \( C_i \) of task \( \tau_i \) and on the maximum blocking time \( B_i \) that can be experienced by each instance of task \( \tau_i \). Hence, in order not to miss any deadline of a HRT task \( \tau_i \) it is sufficient to assign a capacity of \( Q_i = C_i + B_i \) to the isolated server \( S_i \).

From Theorem 3 it is possible to derive sufficient conditions for the schedulability of HRT tasks scheduled by CXP. HRT tasks which do not access any shared resource can be guaranteed exactly like in the original CSS algorithm by assigning them to isolated servers with capacities \( Q_i = C_i \), where \( C_i \) is the WCET of task \( \tau_i \), and periods \( T_i \) equal to the minimum inter-arrival times of \( \tau_i \)'s jobs. A HRT task \( \tau_i \) which accesses shared resources during its execution can be guaranteed if it is assigned to an isolated server \( S_i \) whose capacity \( Q_i = C_i + B_i \) also accounts for the maximum blocking time \( B_i \) that can be experienced by each instance of \( \tau_i \).

6.1 Blocking time computation

An exact computation of the worst-case blocking time \( B_i \) for a HRT task \( \tau_i \) is a complex problem in open systems where the unpredictable behaviour of SRT tasks may cause the associated servers to exhaust their capacities while inside the critical sections, causing many possible situations in which a SRT task can block a HRT task. Without a complete knowledge of the number, type, and behaviour of tasks that may, directly or indirectly, interact through shared resources with a HRT task \( \tau_i \) it is impossible to perform an accurate offline analysis and compute the worst case blocking \( B_i \) that can be experienced by \( \tau_i \) without imposing some pessimism.

The dynamic properties of an open real-time systems only allows us to assume that the WCET of the critical sections that may be accessed by any task through the system’s libraries can be indirectly computed through an offline analysis of those shared libraries. With nested critical sections, the WCET must consider the worst possible path in the blocking chain. The reader may refer to [32] for an extensive survey of the current methods and tools to compute WCETs.

This may be considered too pessimistic since, to guarantee a set of \( n \) HRT tasks, the blocking times must all be summed together at admission time, but the dynamic nature of an open system and lack of information impose such a pessimism. It is impossible to completely identify the conditions under which any task that is dynamically admitted in the system can interfere with a HRT task. Of course, this comes at the cost of a lower system’s utilisation to guarantee HRT tasks. However, with CXP, SRT tasks can benefit from the unused reserved capacities of HRT tasks, minimising this restriction.

If a resource group \( G \) is guaranteed to be composed only by HRT tasks, it is possible to explore all possible blocking situations and compute a more accurate and less pessimistic value for \( B_i \), using, for example, an algorithm similar to the one presented in [17].

7 Evaluation

Extensive simulations were conducted in order to evaluate how the flexible management of reserved capacities performed by CXP can minimise the cost of blocking of soft real-time tasks in the presence of shared resources and conditions of overload. Multiple and independent runs with initial conditions and parameters but different seeds for the random values were used to drive the simulations [25] using a discrete uniform distribution.

Resource sharing protocols that require a prior knowledge of the maximum resource usage time for each task such as the Priority Ceiling Protocol, the Dynamic Priority Ceiling, or the Stack Resource Policy were not considered since they cannot be directly applied to open real-time systems.

The first study compared the cumulative capacity that was consumed by the shortest period (SP) and longest period (LP) tasks of a randomly generated task set when tasks share resources to the amount of capacity that would be consumed if the same set of tasks did not share any resources.
Different sets of 5 tasks were randomly generated, with varied execution requirements ranging from 20 to 60 units, and period distributions ranging from 100 to 300 time units, always ensuring a system’s utilisation \(U \leq 1\). An isolated server was assigned to each task, with a reserved capacity \(Q_i\) equal to the task’s execution requirements and period \(T_i\) equal to the task’s period. The execution requirements of each job were always equal to the reserved capacity of its dedicated server and all jobs accessed the shared resource \(R\) during all their executions, with a new job being released immediately after a task has completed its current job.

Each simulation ran until \(t = 250000\), producing a large variety of inheritance and preemption situations among tasks, and was repeated several times to ensure that stable results were obtained. The cumulated capacities consumed by the SP and LP tasks were recorded every 200 time ticks and the mean values of all generated samples plotted in Figures 5 and 6, respectively.

![Figure 5. Capacity consumed by the SP task](image1)

![Figure 6. Capacity consumed by the LP task](image2)

The achieved results show that with BWI, and due to blocking, while higher priority tasks can consume less than their initial allocations, tasks with longer deadlines can consume more than their reserved capacities since BWI is affected by the absence of a compensation mechanism. In contrast, the efficient capacity exchange mechanism of CXP ensures that both tasks are able to get their allocated capacities even when accessing shared resources thus providing a better fairness than BWI and sustaining the conclusions drawn from the examples in Section 4.

A second study compared the efficiency of the studied protocols BWI, BWE, CFA and CXP in lowering the mean tardiness of a set of periodic jobs with variable execution times in highly dynamic scenarios. At each simulation run, a random number of servers with a system’s utilisation up to 70% contended for the system’s resources with a dynamic traffic that demanded up to 30% of the system’s capacity.

All servers were generated with varied reserved capacities \(Q_i\) ranging from 15 to 50 units of execution and period distributions ranging from 50 to 500 time units, creating different types of load, from short to long deadlines and capacities. Tasks arrived at randomly generated times and remained in the system for a variable period of time with each job having an execution time in the range \([0.8Q_i, 1.2Q_i]\) of its dedicated server’s reserved capacity, originating both overloads and residual capacities due to early completions. There were 6 resources, whose access and duration of use was randomly distributed by the servers, creating direct and transitive blocking situations and distinct resource groups.

For a fair performance comparison against the other algorithms only isolated servers were used in CXP, disabling its ability to steal non-isolated capacities on overloads. The significant improvement on the system’s performance achieved by
allowing active overloaded servers to steal inactive non-isolated capacities, particularly in the presence of a large variation in jobs’ computation times is detailed in [24].

Figure 7 illustrates the performance of the evaluated protocols as a function of the system’s load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs’ overload. The mean tardiness was determined by \( \sum_{i=1}^{n} trd_i / n \), where \( trd_i \) is the tardiness of task \( \tau_i \), and \( n \) the number of evaluated tasks.

Figure 7. Performance in dynamic scenarios

As expected, the achieved results clearly justify the use of a capacity exchange mechanism to minimise the impact of blocking on the system’s performance. Without any compensation for the extra work on blocked servers, BWI obtains the poorest result. Recall that with BWI, a blocked task is only able to use the remaining capacity of its dedicated server, if any.

BWE and CFA achieve similar performances when handling tasks with variable execution times. Both algorithms are unable to reclaim residual capacities originated by early completions, wasting available resources to handle overloads and minimise the number of deadline misses. Also, both algorithms immediately recharge a server’s capacity and extend its deadline at every capacity exhaustion, allowing a task to use resources allocated to a future job, contributing for future jobs of that task to miss their deadlines by larger amounts.

On the other hand, by reclaiming as much extra capacity as possible, CXP outperforms BWE and CFA in lowering the mean tardiness of periodic tasks in highly dynamic scenarios. CXP not only exchanges capacities between all active servers, not restricting capacity exchange to the same resource group, but it also reclaims all the available residual capacity to handle overloads of soft real-time tasks.

Furthermore, these better results in highly dynamic scenarios were achieved with a less complex approach to exchange reserved capacities among servers. Figure 8 illustrates the average overhead introduced by the optimisations of BWE, CFA, and CXP in terms of the needed scheduling time and memory consumption during the previous study, using the base BWI protocol as a reference.

Figure 8. Overhead using BWI as reference

As expected, the optimisations performed by BWE, CFA, and CXP introduce some overhead when compared against BWI in terms of needed time and memory. Although all the three algorithms need only slightly more time than BWI to determine which capacity is going to be accounted by the currently executing server, they substantially differ in terms of
storage information demands. BWE requires a global $n \times n$ matrix to record the amount of capacity that must be exchanged between servers and an extra list at each server to keep track of available capacities. CFA enhances BWI by adding a new task queue to each server and one extra variable for each contracted debt between servers $S_i$ and $S_j$. On the other hand, CXP focuses on minimising the cost of blocking by exchanging reserved capacities as early, and not necessarily as fairly, as possible. As such, it does not account the amount of borrowed capacity on each server neither manages individual resource groups.

The fourth study compared the time and memory needed by CXP to schedule the same task set with and without precedence constraints among its tasks. 10000 tasks sets were randomly generated, with different system’s utilisation in the range $[0.6, 1.0]$. For each task set, a random set of precedence constraints consistent with the tasks deadlines was determined. Each job had random execution requirements in the range $[0.7Q_i, 1.3Q_i]$ of its dedicated server’s reserved capacity. Achieved results allow us to conclude that CXP is able to handle precedence constraints among tasks whose exact behaviour is not known beforehand without any noticeable overhead.

8 Conclusions

This paper presents the Capacity Exchange Protocol (CXP), combining the concept of bandwidth inheritance with an efficient greedy capacity sharing and stealing mechanism to efficiently handle shared resources and precedence constraints among tasks in dynamic open real-time systems. CXP efficiently minimises the cost of blocking by allowing a server to reclaim residual capacities allocated but unused when jobs complete in less than their budgeted execution time, to steal capacity from inactive non-isolated servers used to schedule best-effort jobs, and to exchange capacity between servers that suffered bandwidth inheritance.

Achieved results justify the use of a capacity exchange mechanism that reclaims as much capacity as possible and does not restrict itself to exchange capacities only within a resource sharing group.

The approach is particularly interesting in highly dynamic open real-time systems where is not possible to have a precise knowledge of how many services will need to be concurrently executed neither which resources will be accessed and by how long they will be held. Furthermore, CXP is able to provide hard schedulability guarantees at the expense of a lower system’s utilisation.

References


