Performance Improvement Through Active Idleness

José A.A. Moreira, Carlos F.G. Bispo

Keywords— **Queuing Networks, Distributed Scheduling, Stability.**

Abstract— **We illustrate by an example the potential of distributed scheduling with *Active Idleness* to improve the performance of multi-class queuing networks, which are originally controlled by non-idling, or work-conserving, policies. The queuing network we use in our simulation experiments is due to Dai, [1]. This particular network was shown to be unstable if FIFO is the scheduling policy. However, for the *Last Buffer First Serve* policy, [2], it is stable. Under this setting we show that forcing inactivity during some periods of time in the presence of customers may result in significant performance gains.**

I. Introduction

In an earlier work, [3], we proposed to resort to idling policies to show that the traffic intensity condition is a sufficient stability condition. Starting with a given network and a non-idling scheduling policy which induced instability, we constructed a controller that was able to stabilize the network by introducing a slight change in the way the servers execute the original policy.

In [4] we established the conditions that have to be observed for our method to work and demonstrated how to construct a provably stabilizing controller for a significantly large set of multi-class queuing networks.

However, given that the constructed controller is somewhat conservative, one could argue that, although the proposed method may be useful to stabilize a network, the principle of imposing idleness on the servers in the presence of customers does no make much sense when the purpose is to optimize performance.

In the present paper we intend to show that our controller offers more than just a stabilizing mechanism. It does possess the potential to be used in order to optimize performance.

There are many ways in which to define stability for networks. For our purposes, suffices to say that as long as the expected queue lengths remain bounded at each server, or as long as the expected time to flow through the system remains bounded for all customers, the network is stable.

In the past it was thought that the order by which customers are served would only affect the performance of the system, as long as the service capacity was above the load imposed by the arrival processes. However, as shown in [2], [5], [6] the order by which customers are served does affect the ability of the networks to remain stable, under non-idling policies. Confronted with these puzzling examples the community undertook the task of determining stronger conditions, besides the traffic intensity, that would suffice to ensure stability and kept focused on non-idling policies, [7], [8], [9].

One exception to this was [10], with the *Clear-a-Fraction Policies with Backoff*. There, in the context of multi-class networks with non-zero set-up times, the authors resorted to a supervisory mechanism that would essentially add a little more idling time, by increasing set-up frequency. This way, it was possible to establish a class of provably stable policies.

Curiously, such idea did not have much impact on the work developed afterwards, as the grand majority of the authors kept concentrated on non-idling policies. It is our opinion that [10] holds the key to solve the stability problem. The strong limitation of the method is the fact that the theoretical result is only established for situations where the processing times and the arrival processes are deterministic.

Our contribution in the above mentioned papers has been to explicitly address stochastic networks. There, we proposed to substitute the determination of whether a network is stable by the determination of whether it is stabilizable.

To achieve this we introduced the concept of *Active Idleness*. This concept consists on allowing a server to stay idle in the presence of waiting customers. To implement this concept in a real setting, a supervisory mechanism, denominated *Time Window (TW) Controller*, was developed. This mechanism consists on assigning to each class in the network a fraction of the available capacity. If a given class uses more than its share of capacity, it is blocked from being processed by a certain amount of time, which only depends on the system's evolution, that is, it is not calculated *a priori*. With the *TW Controller* it is possible to stabilize a significant amount of non-acyclic, multi-class, queuing networks, which are unstable under their original scheduling policy, provided that the original policy is non-idling.

In the next section we briefly introduce the models addressed. In Section III we present the basics of the *TW Controller*. Then, in Section IV, we present a summary of results for Dai's example, and we conclude in Section V.

II. Queuing Networks, Scheduling, and Stability

Our setting is that of open and non-acyclic networks, processing multiple classes of customers. First we consider there are different types of customers, where each type is characterized by its external arrival process, by its routing through the network, and by the service distributions at each server and visit number, if more than one visit is paid to a server. We assume the routing for each type to be deterministic. Furthermore, we consider a given type to be constituted of different classes, in the following sense:

J. Moreira is with Agilent Technologies, Detschland GmbH, Boeblingen, Germany. E-mail: jose_moreira@agilent.com. C. Bispo is with the Istituto de Sistemas e Robótica - Instituto Superior Técnico, Lisbon, Portugal. E-mail: cfb@isr.ist.utl.pt.

a customer is processed by a server and upon being sent to the next server it will change class. This allows the distinction of the same type of customers on different visits to the same server, if ever they occur.

Given that the same server may have different processing distributions for different classes, these are not Jackson networks. Otherwise, a simple product form distribution could be computed for the number of customers in the system, if each server would process customers according the the FIFO scheduling policy. For those, the traffic condition is sufficient for the existence of such invariant distribution and, consequently, for stability to be ensured.

Given that the networks may have a significant dimension in terms of servers and classes, the state space is too large to allow a centralized scheduling policy to be computed. Therefore, usually these are controlled by distributed scheduling policies, which are solely based on the contents of each server's waiting queue. See [11], [12] for surveys on distributed scheduling policies. For a very good example of non-local scheduling policies see [13] with their *Fluctuation Smoothing* policies, which suffer from the fact that they require complex implementation, as discussed in [14].

Under our setting the traffic intensity condition can be stated as

$$\rho_i = \sum_{k=1}^{N_i} \mu_i^{c(i,k)} . \lambda_i^{c(i,k)} < 1, \tag{1}$$

for $i = 1, 2, \ldots, I$, where I is number of servers, N_i is the total number of different classes visiting server i, $\mu_i^{c(i,k)}$ is the first moment of the processing time distribution of class $c(i,k)$ on server i, and $\lambda_i^{c(i,k)}$) is the first moment of the arrival rate of customers of class $c(i,k)$ to server i. Furthermore, given that each class belongs to some type of customer it has to hold that

$$\lambda_i^{c(i,k_i)} = \lambda_j^{c(j,k_j)}, \tag{2}$$

for all classes and servers such that, if class $c(i,k_i)$ upon being served on server i visits next server j termed as class $c(j,k_j)$. That is, class $c(i,k_i)$ and class $c(j,k_j)$ belong to the same customer type.

The pair constituted by Equations 1 and 2 establishes that the load imposed by the external arrival processes on each server is below its capacity. Any scheduling policy that ensures the internal arrival processes to verify Equation 2, when Equation 1 holds, ensures stability of the overall network.

III. The Time Window Controller

To simplify the notation we assume that there is a global numbering of classes ranging from $k = 1, 2, \ldots, K$. We define $\{\epsilon_k(n)\}$ as the time between the external arrival of the n^{th} and the $(n-1)^{th}$ customer of class k. These are assumed to be independent and identically distributed and, for some k, $\epsilon_k(n) = \infty$ for all n, in which case the external arrival process to class k is null. This means that class k is generated from another class in the system when moving from a server to the next.

A customer of class k, after being served at a unique server j, written $j = s(k)$, or conversely $k \in c(j)$, becomes a customer of class $\mathcal{R}(k)$, where $\mathcal{R}$ is a bijective function representing the routing map for the queuing network.

Therefore, we can uniquely assign a service distribution to each class and denote by μ_k its first moment. Customers of different classes do not merge into a single class, nor does a single class split in more than one class. For each class k, define $F(k)$ as

$$F(k) = \begin{cases} k & \text{if class } k \text{ has a non null exogenous arrival rate.} \\ F(j) & \text{if } k \text{ has a null exogenous arrival rate, where } j \text{ is the class that directly feeds to class } k \text{ for which } F(j) \text{ has been defined.} \end{cases} \tag{3}$$

Note that, since there is no class split nor merge, $F(k)$ is an injective function. For each class k let

$$\lambda_k = \frac{1}{E[\epsilon_{F(k)}(1)]}. \tag{4}$$

One interprets λ_k as the effective mean arrival rate of class k.

The *TW Controller* is introduced with a series of definitions, the first of which is the *Time Window* associated to a class.

Definition III.1 (Time Window) Consider a class k of a multi-class, non-acyclic, queuing network. The *Time Window* associated to that class is defined as the finite time interval that starts at the current system time t_c and extends T_k time units into the past.

The *Time Window* of a class represents the amount of past time needed by the *TW Controller* for class k. Next, the definition of the *Processing History* associated to a class is introduced.

Definition III.2 (Processing History) For each customer i of class k, define $t_{k,i}^{start}$ and $t_{k,i}^{end}$ as the start and finish time instant for the processing of that customer, respectively. The *Processing History* of class k is defined as a function $H_k(t)$ given by:

$$\underset{0 \le t \le t_c}{H_k(t)} = \begin{cases} 1 & \text{if } \mathrm{t}_{(\mathrm{k,i})}^{\mathrm{start}} \le \mathrm{t} \le \mathrm{t}_{(\mathrm{k,i})}^{\mathrm{end}} \ \exists \mathrm{i} \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

The *Processing History* associated to a class represents a function that describes the amount of time used by the server to process customers of that class. The next definition presents the concept of the *Time Fraction* of a class k at a given time t.

Definition III.3 (Time Fraction) The *Time Fraction* of class k with a *Time Window* of size T_k at time t is defined as $f_k(t)$ and is computed by the following expression:

$$f_k(t) = \beta_k \int_{t-T_k}^{t} e^{\alpha_k.(\tau-t)} H_k(\tau) d\tau \qquad (6)$$

where $\alpha_k \in [0, \infty[$, is a Smoothing Parameter and β_k is a normalization parameter, given by:

$$\beta_k = \frac{\alpha_k}{1 - e^{-\alpha_k.T_k}} \qquad (7)$$

The *Time Fraction* of a class is an estimate of the fraction of the total time contained in its *Time Window* during which the server was processing customers of that class. It clearly represents a measure of the amount of server capacity assigned to that class. If $\alpha_k = 0$, it measures the exact time fraction allocated to class k over the *Time Window* span, given that β_k becomes $1/T_k$. Since the *Time Window* size is finite, it is necessary to include the normalization factor β_k. This guarantees that, if during the entire *Time Window* the server is always processing customers of a given class, the computed value for the *Time Fraction* of that class will be 1. The last concept necessary is that of a *Blocked* class, which has the following definition.

Definition III.4 (Blocked class) A class k is said to be *Blocked* at time t with parameter f_k^{max}, if $f_k(t) > f_k^{max}$, where f_k^{max} is the *Maximum Time Fraction* allowed for class k.

A *Blocked* class is simply a class that has exceeded the f_k^{max} awarded to it. Since the $f_k(t)$, is a measure of the server capacity used by class k in its T_k, f_k^{max} represents the maximum level of capacity that class k can use during T_k without becoming *Blocked*. Finally, using the previous definitions the definition of the *Time Window Controller* is presented.

Definition III.5 (TW Controller) Let ω be a multiclass, non-acyclic, queuing network, where each service station is controlled by the non-idling scheduling policy Λ. The *Time Window Controller* for this queuing network consists on assigning to each class k an f_k^{max} and a T_k, for which it is possible to compute $H_k(t)$, with a *Smoothing Parameter*, α_k. Each service station performs its scheduling decisions using policy Λ, with the exception that all classes that are *Blocked* should be considered empty of customers.

The *TW Controller* is described by a set of parameters $(\alpha_k, T_k, f_k^{max})$ with $k = 1, \ldots, K$. The functioning of the *TW Controller* is very simple. Each time a server has to make a scheduling decision, the *TW Controller* calculates the f_k of all classes in that server. If any class has an f_k above f_k^{max}, then the *TW Controller* blocks that class from the set of classes from which the server can remove customers to process. Note that there is no interruption nor preempting of ongoing services. The decision points coincide with the conclusion of a service, the arrival of a new customer, or the time at which a class stops being blocked. The two later decision points are solely valid when the server is idle.

At certain times the scheduling policy may not be able to choose a customer to be processed because all customers are in classes that are *Blocked*. In this case the server becomes idle, not because the queues are empty of customers, but because the *TW Controller* forbids the scheduling policy from using the available customers.

For this reason this type of idleness is termed as *Active Idleness*. In the present context, idleness incurred for actual lack of customers, would be considered as *Passive Idleness*.

After being *blocked*, a class will see its corresponding f_k decrease with time, guaranteeing that at some point in the future it will cease to be *Blocked*. Note that adding the *TW Controller* keeps the overall scheduling policy distributed. Each server, in essence, has a *TW Controller* with the $(\alpha_k, T_k, f_k^{max})$ parameters corresponding to the classes it processes.

A. Qualitative discussion on properties

As it has been defined, the *TW Controller* can be adjusted in order not to influence the original policy. If all smoothing parameters are set to zero and if all maximum time fractions are set to one, no matter the window size for each class, no class will ever be blocked. This way, the system will be ran with the original policy. As some maximum time fractions are decreased to a number less than one, the *TW Controller* will progressively increase its influence over the original policy, as those classes will start getting blocked with increased frequency.

Naturally, if a given class receives a maximum time fraction below its long term needs in terms of arrival rate and processing time, instability will occur.

Therefore, each class should be awarded a maximum time fraction slightly above its individual needs. Note that these fractions are short term fractions. They need to be above the long term needs to allow each class to have access to its necessary long term share of capacity.

One possible and simple way to define these fractions is to split each server's capacity in such a way that all of them are above the long term needs of their classes, but their sum does not exceed one. This is ensured to be feasible when the traffic intensity holds, because each server will have some surplus capacity.

When the choice of fractions is such that their sum is equal to one, we are in some sense decoupling the network, given that it behaves as if each server is split into smaller servers dedicated to each class. This particular choice makes the *TW Controller* similar to the *Generalized Processor Sharing – GPS*. See [4] for details on how to construct the stabilizing version of the controller.

However, we may allow the fractions on a given server to add up to more than one, meaning in this case that some degree of coupling and interference is allowed between different classes visiting the same server. This particular feature makes our controller drastically different from those based on the *GPS* concept.

The higher the degree of interference, the higher is the potential to better use a given server, but also the higher is the potential for instability. The example of [1], which is unstable under FIFO, is a case where, under our setting, each maximum time fraction is set to one for all classes. We argue that this maximum interference causes short term losses of capacity that will not be recovered in the long run, causing the observed instability.

IV. Simulation Results

Dai, in [1], presented a queuing network topology that in connection with the *First In First Out (FIFO)* scheduling policy resulted in an unstable network. Figure 1 presents a diagram of the queuing network layout, which is constituted by two servers with six classes, corresponding to a single type of customer.

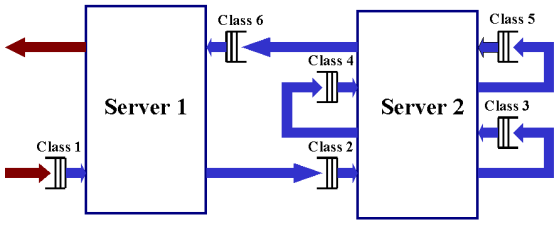

Fig. 1. Dai's queueing network topology.

Table I presents a set of parameters for this queuing network that in conjunction with the *FIFO* scheduling policy results in an unstable network. Note that the parameters respect the *Traffic Intensity Condition*. Note also, according to the discussion on Section II, that there is only one type of customer. Therefore, there is only one external arrival process.

TABLE I
Queuing network parameters.

Parameter	Value
λ	1.000
μ_1	0.001
μ_2	0.897
μ_3	0.001
μ_4	0.001
μ_5	0.001
μ_6	0.899

The simulation results obtained by Dai demonstrated that this network topology under the *FIFO* policy presents an unstable behaviour. This implies that, for this network, the *Traffic Intensity Condition* is not a sufficient stability condition, understanding here that the queuing network is composed by the topology and the scheduling policy. To replicate those results for this paper, the network was simulated in a stochastic setting, where the arrival of customers to the system is modeled by Poisson processes and the processing times at the service stations are modeled by exponential distributions. Their respective mean values are those displayed in Table I. Figure 2 presents our simulation results in the form of the server's inventory, or queue length, evolution as a function of time.

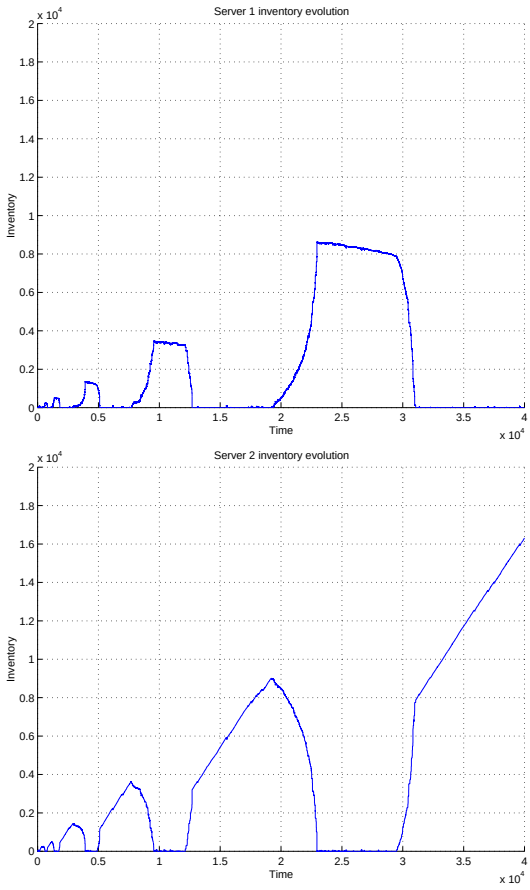

Fig. 2. Server inventory evolution for the *FIFO* scheduling policy.

The figures show the sum of customers on each server as a function of time. Basically one can observe that there are periods during which the servers are available to work but their queues are empty – starvation. These starvation periods are intertwined with busy periods which grow in size and length almost linearly as time progresses. The same happens with the idle periods. For instance, server 2 reaches a total of customers a little over to 16,000 close to the end of simulation.

The results demonstrate that the unstable behavior observed for the *FIFO* scheduling policy is due to the inability of the servers to use the available resources to process the customers, since the scheduling policy creates a starvation phenomena between the servers. In [4] we provide an instance of our controller which stabilizes this network, while using *FIFO* at all servers.

The question that this paper addresses is if the *TW Controller* presents any advantages for pairs topology/scheduling policy which are stable. Dai's network is stable under the *Last Buffer First Serve (LBFS)* policy, [2]. It is also shown in [2] that the *LBFS* policy is able to achieve good performances in comparison with other distributed scheduling policies. Taking into account these facts, a pos-

sible demonstration of the performance enhancement properties of the *TW Controller* would be to use Dai's network with the *LBFS* policy.

The *TW Controller* was set with the parameters presented in Table II, where the value of f_6^{max} will be changed from unity, corresponding to the original *LBFS* scheduling policy, to smaller values.

TABLE II
TW Controller PARAMETERS.

Parameter	Value
T_k	100
α_k	0.01
f_1^{max}	1.0
f_2^{max}	1.0
f_3^{max}	1.0
f_4^{max}	1.0
f_5^{max}	1.0
f_2^{max}	—

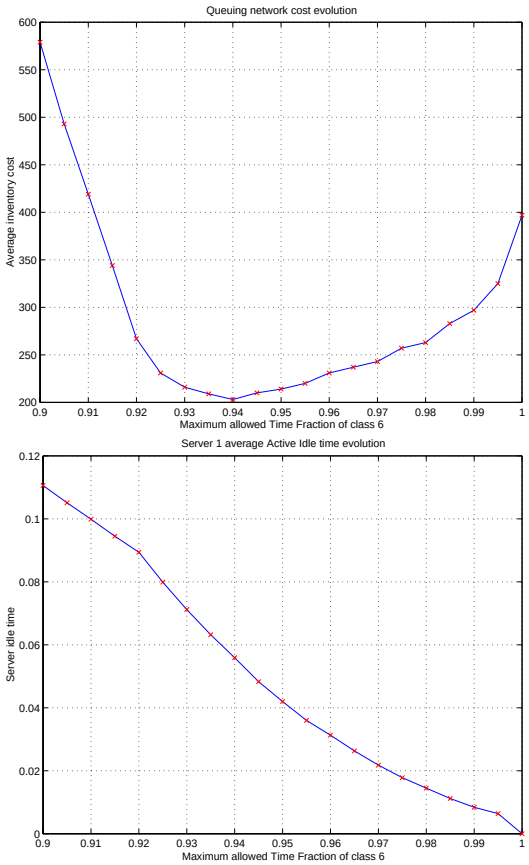

Fig. 3. Evolution of the queuing network cost (left) and average *Active Idle* time of server 1 (right) for the *LBFS* + *TW Controller* scheduling policy with the f_6^{max} parameter.

Figure 3 presents the results of several simulations performed with different values of f_6^{max}, displaying the average cost and *Active Idle* time at server 1, as functions of f_6^{max}. The cost function used is a linear combination of each buffer's contents, with different weights given to each buffer.

The results show that the *TW Controller* is able to improve performance over the *LBFS* policy. The improvement displayed is close to 50% for f_6^{max} close to 0.94. Connected with this improvement is the inclusion of *Active Idleness* only on server 1, since all classes in server 2 have their *Maximum Time Fraction* set to unity. Note that as f_6^{max} decreases, there is a point after which the performance degrades significantly. By inspecting Table I it clear that such degradation is due to the fact that the fraction assigned to class 6 approaches its stability bound. Thus, the excessive reduction of fraction for class 6 will, naturally, induce instability.

Figure 4 presents the evolution of the average inventory in the system. Average inventory is another instance of the type of cost function used on Figure 3. It is also possible to observe a significant improvement for some instances, which implies a reduction in the customers' lead time due to Little's Law. Note that, for this alternative cost function, the optimal value of f_6^{max} is between 0.95 and 0.97, and the cost reduction relatively to $f_6^{max} = 1$ is a little over 25%. The average inventory for $f_6^{max} = 1$ is 80 and the average inventory for $f_6^{max} = 0.955$ is 57.

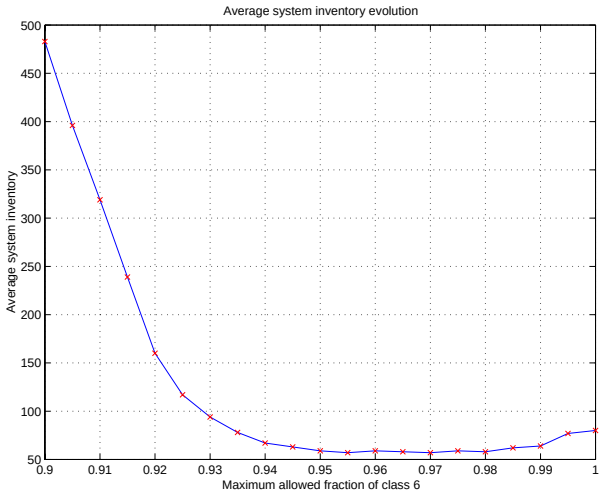

Fig. 4. Evolution of the average system inventory with the f_6^{max} parameter.

The *TW Controller* allows the short term fractions to add up to more than one. This particular feature corresponds to a generalization of the *GPS*-based policies, for which the instantaneous control fractions are constrained to sum up to one.

The *TW Controller* is able to stabilize the system, adding in the process some *Active Idleness*, as presented in [4]. Naturally, the total idle time is higher on the original system. But the original system only has *Passive Idle Time*.

It should be emphasized that the reduction in cost displayed was accomplished just by changing one of the parameters. One can expect even more dramatic performance improvements by determining the six optimal fractions. Also, the time window and the smoothing parameter in-

fluence the achievable performance. These are also susceptible of being set to their optimal values.

A. Discussion

The question to be addressed here is to provide some intuition on as to why do we get such tremendous performance change at the expense of forcing some customers to wait when the server is available. When we only have a single server it does not make any sense to keep customers waiting while the server is available, because there is no gain for anyone. However, in a network what seems to be good for each individual server may – and in fact does – hurt the network as whole.

Given there is no control over the input of customers into the system, all burstiness the arrival processes contain will be transferred to the network and allowed to propagate freely. Burstiness in the arrival process is directly connected to variance. It is well known the impact that variance has on queuing networks.

The *TW Controller*, with its short/medium term bounds on capacity utilization by each class of customers, acts as a burstiness filter, that is, contributes to reduce the variance on all the internal arrival processes.

Another way of looking at this controller is the following. Most local scheduling policies implement local feedback but the overall network is operated in open loop. The proposed controller adds some measure of global feedback to the local decisions. Moreover, that global feedback only relies on knowing the external arrival rates of customer types, but its implementation preserves the locality of the decision making process.

V. Conclusion

We showed that our supervisory controller, originally designed to stabilize networks, is also a tool to optimize the performance of queuing networks. By adjusting the fractions of each class one can reach the optimal balance for sharing capacity among classes. While non-idling policies make sense in the context of single server, they may induce significant amounts of idle time for multiple servers. Therefore, they may hurt stability of the networks and ultimately may hurt performance. By forcing some idleness at points well defined in time, one can reduce the amount of total idle time and, not only stabilize otherwise unstable networks, but also improve performance over non-idling policies for networks which are already stable.

The paper presents one example to illustrate the claim. Although we do not provide any formal proofs here, the approach provably stabilizes queuing networks, as long as the processing times of each class possess an upper bound. This constraint means, for instance, we cannot ensure to stabilize Markovian networks. While this can be seen as a theoretical limitation of our controller, it is not a serious limitation from the practical point of view, given that any real life network does have some sort of upper bound on the processing times of customers. There is always such as thing as the longest service ever.

The essence of the stability proof, which can be found in [4], is as follows. The smoothing parameter is set to zero. The fractions are set to add up to one in a way that each class has a fraction slightly above its long range needs. The window size is set equal to all classes and a function of the upper bound on the processing times of all classes: the longer the longest service the wider the window. This choice of parameters is one possible instance of all values they can take. With this particular choice of values it is possible to show that each class will have an average availability of its server above its long range needs and that the longest service will not take capacity away from other classes. Given that we can, by construction, provide a provably stable instance, it follows that the optimal choice of these parameters cannot lead to instability, provided all queues are observable on the performance measure being used.

The most relevant consequence of this result is that the *Traffic Intensity Condition* is sufficient to ensure stabilizability on a very wide class of networks.

VI. Acknowledgements

The work described in this paper was partially funded by Fundação para a Ciência e Tecnologia under references SRI/34646/99-00 and Praxis XXI/BM/21090/99.

References

[1] Jim G. Dai, "On positive Harris recurrence of multiclass queueing networks: A unified approach via fluid limit models," *The Annals of Applied Probability*, vol. 5, no. 1, 1995.

[2] Steve H. Lu and P. R. Kumar, "Distributed scheduling policies based on due dates and buffer priorities," *IEEE Transactions on Automatic Control*, vol. 36, no. 12, Dec. 1991.

[3] José A. A. Moreira and Carlos F. G. Bispo, "Stability or stabilizability? Seidman's FCFS example revisited," in *Proceedings of 10th Mediterranean Conference on Control and Automation*, Lisbon, Portugal, 2002.

[4] José A. A. Moreira and Carlos F. G. Bispo, "Distributed scheduling aith active idleness: a key to to the stabilization of multiclass queuing networks," *Submitted to the IEEE Trans. on Aut. Control*, 2002.

[5] Thomas I. Seidman, "'first come, first served' can be unstable!," *IEEE Transactions on Automatic Control*, vol. 39, no. 10, Oct. 1994.

[6] Maury Bramson, "Instability of FIFO queueing networks," *The Annals of Applied Probability*, vol. 4, no. 2, 1994.

[7] Sunil Kumar and P. R. Kumar, "Performance bounds for queuing networks and scheduling policies," *IEEE Transactions on Automatic Control*, vol. 39, no. 8, pp. 1600–1611, Aug. 1994.

[8] Dimitris Bertsimas, David Gamarnik, and John N. Tsitsiklis, "Stability conditions for multiclass fluid queueing networks," *IEEE Transactions on Automatic Control*, vol. 41, no. 11, pp. 1618–1631, Nov. 1996.

[9] Hong Chen and Hanqin Zhang, "Stability of multiclass queueing networks under priority service disciplines," *Operations Research*, vol. 48, no. 1, pp. 26–37, 2000.

[10] P. R. Kumar and Thomas I. Seidman, "Dynamic instabilities and stabilization methods in distributed real-time scheduling of manufacturing systems," *IEEE Transactions on Automatic Control*, vol. 35, no. 3, Mar. 1990.

[11] Stephen C. Graves, "A review of production scheduling," *Operations Research*, vol. 29, 1981, July-August.

[12] S. S. Panwalkar and Wafik Iskander, "A survey of scheduling rules," *Operations Research*, vol. 25, no. 1, 1977.

[13] Steve C. H. Lu, Deepa Ramaswamy, and P. R. Kumar, "Efficient scheduling policies to reduce mean and variance of cycle-time in semiconductor manufacturing plants," *IEEE Transactions on Semiconductor Manufacturing*, vol. 7, no. 3, Aug. 1994.

[14] Tomohito Nakata, Koichi Matsui, Yasuhisa Miyake, and Kyusaku Nishioka, "Dynamic bottleneck control in wide variety production factory," *IEEE Transactions on Semiconductor Manufacturing*, vol. 12, no. 3, Aug. 1999.