STABILITY OR STABILIZABILITY? SEIDMAN'S FCFS EXAMPLE REVISITED

José A.A. Moreira*, **Carlos F.G. Bispo**†

* Agilent Technologies, Detschland GmbH,
Herrenberger Strasse 130, CCST-BUS, D-71034 Boeblingen, Germany
fax: +49-7031/464-4708
e-mail: jose_moreira@agilent.com

† Instituto de Sistemas e Robótica – Instituto Superior Técnico,
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
fax: +351-218 418 291
e-mail: cfb@isr.ist.utl.pt

Keywords: Queueing Networks, Distributed Scheduling, Stability.

Abstract

We address the issue of stability for multi-class, non-acyclic, and stochastic queueing networks. This has been an exciting problem, addressed by the research community in over a decade, where the nature of the *traffic intensity condition* as being sufficient for such networks to be stable has been debated and questioned. We argue that the concept of stability ought to be replaced by that of *stabilizability* and that this property is intrinsic to the network's topology. Under this more generic setting, and resorting to *idling policies*, we provide a distributed supervisory controller that is able to stabilize a large set of networks, provided that the traffic intensity condition holds. Seidman's example, [12], is used to demonstrate such property.

1 Introduction

The objective of this paper is to present an alternative perspective over the problem of multi-class queueing networks' stability. The need for such perspective has been motivated by a series of examples published over the last decade, where relatively simple networks operated by reasonable distributed scheduling policies are not stable, although the traffic intensity condition holds. By reasonable policies is usually understood the class of policies that keep each server busy as long as there is a queue of customers demanding its service – non-idling policies. The rational behind this is that doing the opposite would result in a waste of service capacity, contrary to the objective of getting the customers out of the system within a reasonable amount of time upon their arrival to the system.

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 in 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 [8, 12, 2] 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, [6, 1, 3].

One exception to this was [5], 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 [5] holds the key to solve the stability problem, although we do not agree entirely with the options made then. Essentially, their system would be run by a non-idling policy that would reduce the set-up frequency to a minimum and when a given run over a specific class would exceed a given amount of time, there would be a switch to a new class and all remaining customers of the first class would be sent to a special priority queue to be processed later on a first come first serve basis. We disagree with the fact that regular queues and the priority queue are run under possibly different policies, as the regular policy might not be the *FCFS*. Moreover, their result is only established for deterministic processing times.

What we believe to be the key to solve the stability problem is the fact that the resulting policy adds idle time in the presence of work, or put in another way, is the fact that the class/buffer switching frequency is increased at the expense of wasting capacity.

We argue that the networks' stability problem should be addressed as whether a given network is stabilizable and not whether the pair network-policy, or network-class of policies, is stable, as it has been the main focus in the past. We argue also that stabilizability is an intrinsic property of each network's topology. By network's topology we understand how the different classes of customers flow through it – their routes –, how many servers are present, what are the service distributions for each pair server-class of customers, and what are the arrival distributions for each different customer class. Under this setting, and by imposing reasonably mild constraints on the service and arrival processes, we claim that the traffic intensity condition is sufficient to ensure stabilizability.

To achieve this we introduce 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*, is presented. 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, queueing networks, which are unstable under their original scheduling policy, provided that the original policy is non-idling.

The paper presents results obtained for Seidman's example, [12], as an instance to demonstrate by simulation the technique's potential. It turns out also that, besides stabilizing unstable systems, we have been able to improve performance on systems which are already stable with the original policy. This last feature is of particular relevance, given that the fine tuning of our controller will allow the optimization of any given non-idling policy concerning the degree of Active Idleness that should be allowed to exist.

In the next section we briefly introduce the models addressed. In Section 3 we present the basics of the *TW Controller*. Then, in Section 4, we present a summary of results for Seidman's example, and we conclude, in Section 5, with a qualitative discussion of the concept proposed.

2 Queueing Networks, Scheduling, and Stability

We consider that a queueing network is constituted by two components: the *queueing network topology* and the *control policy*. The *queueing network topology* contains the layout of the network in the form of the routing each class of customers must follow. The *topology* also includes the description for customer arrival and processing time distributions. The *control policy* performs two functions: controlling the admission of new customers into the network, which is usually referred as the *admission policy*, and deciding in which order each server processes cus-

tomers, known as the *scheduling policy*. Although all queueing networks must have some scheduling policy, it is not mandatory for them to possess an admission policy. Those which do not are called *open networks*, as opposed to the *closed networks* which have an admission policy.

As to the layout, queueing networks can be called *acyclic* or *non-acyclic*. The first group includes networks which do not have cycles in the flow of customers, and the second imposes no restriction on the flow of customers. This later group includes networks where a given class of customers may visit the same server more than once before leaving the system.

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 of each type to be deterministic. Furthermore, we consider a given type to be constituted of different classes, in the following sense: 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 FCFS 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 [4, 11] for surveys on distributed scheduling policies. For a very good example of non-local scheduling policies see [7] with their *Fluctuation Smoothing* policies, which suffer from the fact that they require complex implementation, as discussed in [10].

Under our setting the traffic intensity condition can be stated as

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

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

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

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.

3 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 nth 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 queueing 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

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

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

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

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 3.1 (Time Window) Consider a class k of a multi-class, non-acyclic, queueing 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. The use of a finite size window is not only due to memory and computational requirements, but is also essential to achieve the short/medium term objectives set to the *TW Controller*, as will be shown in the next section. Next, the definition of the *Processing History* associated to a class is introduced.

Definition 3.2 (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:

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

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 objective is to obtain a chronological description of the processing time used by each class from the corresponding server. The next definition presents the concept of the *Time Fraction* of a class k at a given time t.

Definition 3.3 (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:

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

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

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

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 $1/T_k$. The need to use an exponential function to smooth the *Processing History*

is to eliminate sudden changes on the *Time Fraction* value of a class. The objective is that when computing the *Time Fraction*, the more recent processing history has a larger weight to the computation than the processing history near the end of the *Time Window*. Since the *Time Window* size is finite, it was 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 3.4 (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 3.5 (TW Controller) Let ω be a multi-class, non-acyclic, queueing network, where each service station is controlled by the non-idling scheduling policy Λ. The *Time Window Controller* for this queueing 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 definition states that 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 – these decision points coincide with the end of service.

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 server is empty of customers, but because the *TW Controller* forbids the scheduling policy of 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, and those parameters can be set up-front.

3.1 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*.

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*.

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 [12] 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.

4 Simulation Results

Seidman, in [12], presented a queueing network topology that in connection with the *First In First Out (FIFO)* scheduling policy resulted in an unstable queueing network.

Figure 1 presents a diagram of the queueing network layout, which is constituted by four servers with twelve classes, corresponding to four different types of customers.

Table 1 presents a set of parameters for this queueing

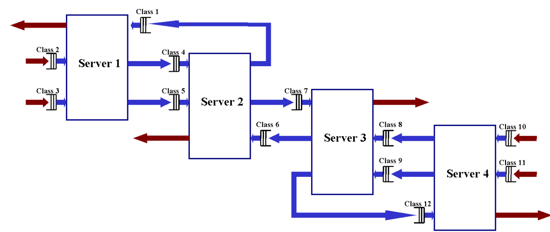

Figure 1: Seidman's queueing network topology.

network that in conjunction with the *FIFO* scheduling policy results in an unstable queueing network. Note that the parameters respect the *Traffic Intensity Condition*.

Table 1: Queueing network parameters.

Parameter	Value	Parameter	Value
λ_2	1.000	μ_5	0.001
λ_3	1.000	μ_6	0.900
λ_{10}	1.000	μ_7	0.900
λ_{11}	1.000	μ_8	0.001
μ_1	0.900	μ_9	0.001
μ_2	0.001	μ_{10}	0.001
μ_3	0.001	μ_{11}	0.001
μ_4	0.001	μ_{12}	0.900

Figures 2 and 3 present the results obtained for a simulation of the queueing network with the *FIFO* scheduling policy using the queueing network parameters presented in Table 1. The simulation was run for a total of 40,000 periods.

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 of approximately 18,000 close to the end of simulation.

Clearly, these results show that the queueing network is unstable. The original example was shown to be unstable with deterministic processing times.

We have used Poisson arrivals and exponential processing times for our simulations. The behavior we present is qualitatively similar to that presented by Seidman in his article.

Figures 4 and 5 present a comparison of the server inventory evolution of the *FIFO* scheduling policy with the same scheduling policy supervised by the *TW Controller*.

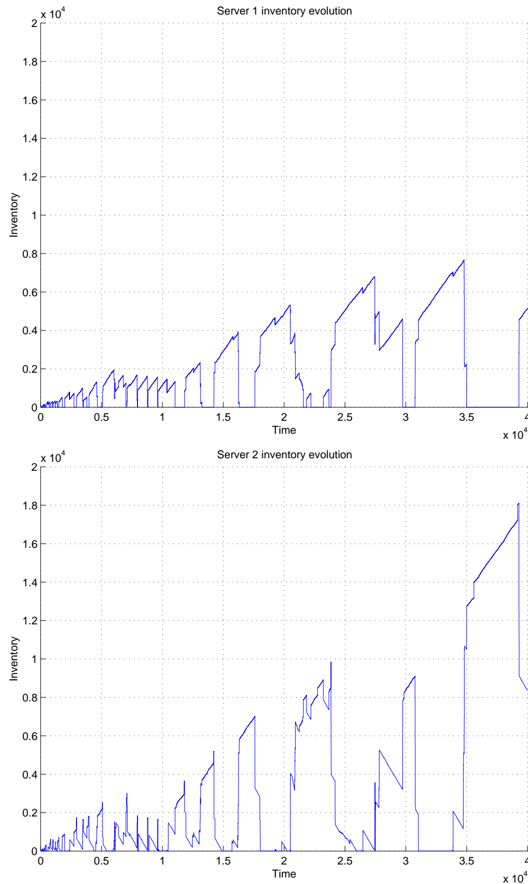

Figure 2: Server inventory evolution for the *FIFO* scheduling policy (server 1 and 2).

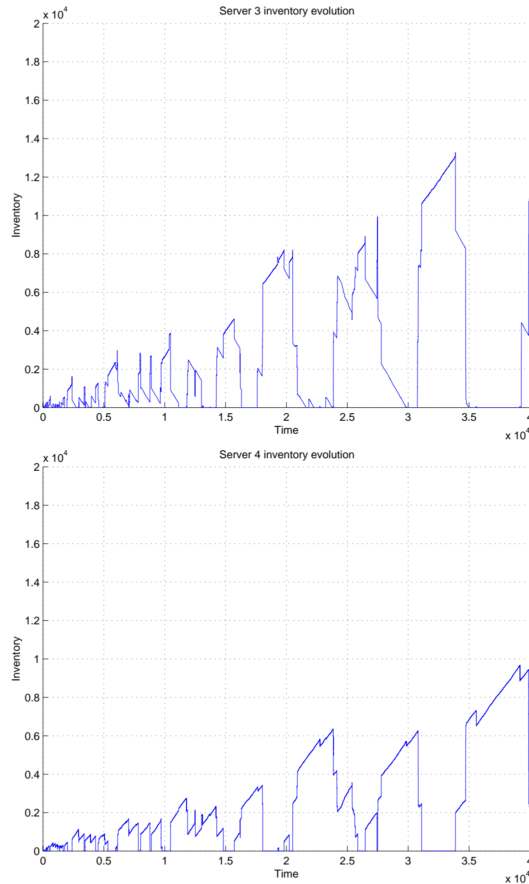

Figure 3: Server inventory evolution for the *FIFO* scheduling policy (server 3 and 4).

Through simulation we demonstrate that the *TW Controller* is able to stabilize this queueing network. The choice of parameters for the *TW Controller* was made by allocating to each class a *Maximum Time Fraction*, f_k^{max}, proportional to $\lambda_k \mu_k$. The surplus capacity was equally divided among all classes. Table 2 presents the choice of parameters for the *TW Controller*.

This choice of parameters is such that the sum of fractions at each server is exactly equal to one. Therefore, this is a choice made with the intent of reducing to a minimum the degree of coupling between different classes. Also, we made the time windows and smoothing parameters equal for all classes.

Table 2: *TW Controller* parameters.

Parameter	Value	Parameter	Value
T_k	100	f_6^{max}	0.996
α_k	0.010	f_7^{max}	0.996
f_1^{max}	0.996	f_8^{max}	0.002
f_2^{max}	0.002	f_9^{max}	0.002
f_3^{max}	0.002	f_{10}^{max}	0.002
f_4^{max}	0.002	f_{11}^{max}	0.002
f_5^{max}	0.002	f_{12}^{max}	0.996

We had to switch the scale of the plots in order to better see the buffer sizes at each server when compared with the original values. Now, server 2 has a max-

imum of less than 600 customers along the 40,000 periods of the simulation. Besides the reduction of the maximum value, the total number of customers as a function of time exhibits a stationary behavior.

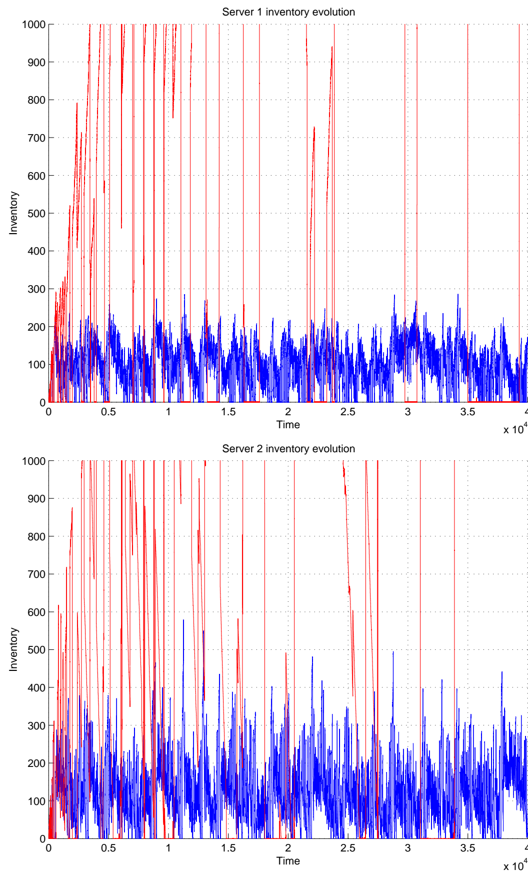

Figure 4: Comparison of the server inventory evolution for the *FIFO* (light grey) and *FIFO + TW Controller* (dark grey) scheduling policies (server 1 and 2).

The results show that the unstable behavior observed for the *FIFO* scheduling policy is due to the inability of the servers to use enough resources to process the customers, since the scheduling policy creates a starvation phenomena between the servers.

The *TW Controller* is able to stabilize the system, adding in the process some *Active Idleness*, as presented in Table 3. Naturally, the total idle time is higher on the original system. But the original system only has *Passive Idle Time*.

The amount of *Active Idle Time* introduced by our controller is relatively small – the maximum occurs for server 3 and amounts to less than 2.4% of the total simulation length –, but produces a huge difference in the overall performance.

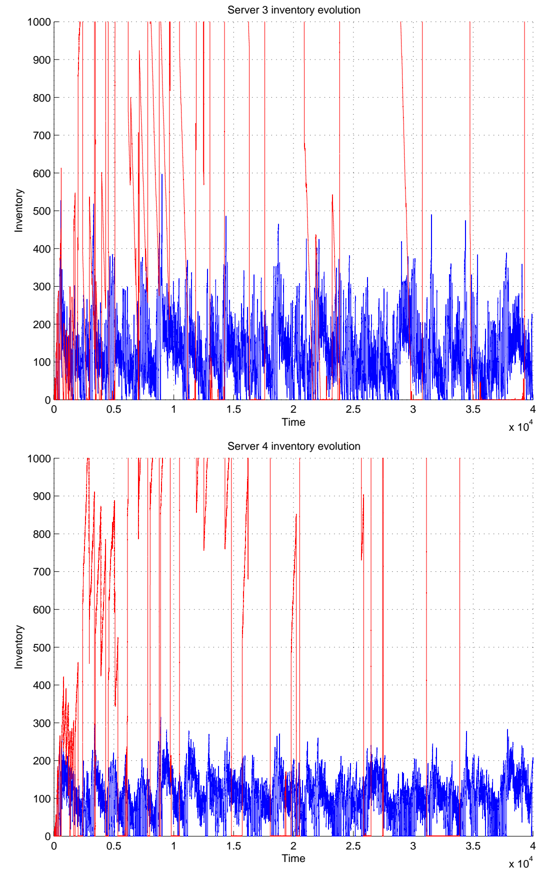

Figure 5: Comparison of the server inventory evolution for the *FIFO* (light grey) and *FIFO + TW Controller* (dark grey) scheduling policies (server 3 and 4).

Table 3: Comparison of the *Active Idle Time* used in each server.

Statistic	Original	*TW*
Server 1	0	0.0128
Server 2	0	0.0218
Server 3	0	0.0234
Server 4	0	0.0123

It would be possible to further improve the performance by choosing different maximum time fractions. This could allow some degree of interference between the classes. This is an issue of performance optimization, which is outside the scope of

the present paper, but constitutes the challenge we face now.

4.1 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 queueing 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.

5 Conclusion

We proposed a supervisory controller that resorts to *Active Idleness* to ensure the stabilization of multiclass, stochastic queueing networks operated with non-idling scheduling policies and are unstable, despite the fact that the *Traffic Intensity Condition* holds.

The paper presents one example to illustrate the claim. Although we do not provide any formal proofs here, the approach provably stabilizes queueing 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.

Nevertheless, the example presented is a Markovian network. On the other hand, in reality no queueing network is such that there is no upper bound on the processing times of individual customers. Therefore, the range of application of the controller is very significant.

The essence of the stability proof, which can be found in [9], 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. 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 by induction 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.

We have also been able to obtain performance improvements over non-idling policies which do not generate instability. This second feature is also particularly interesting because, provided an optimization procedure is in place, the adequate measure of *Active Idleness* can be determined through the adequate choice of maximum time fractions for each class.

6 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] Dimitris Bertsimas, David Gamarnik, and John N. Tsitsiklis. Stability conditions for multiclass fluid queueing networks. *IEEE Transactions on Automatic Control*, 41(11):1618–1631, November 1996.

[2] Maury Bramson. Instability of FIFO queueing networks. *The Annals of Applied Probability*, 4(2), 1994.

[3] Hong Chen and Hanqin Zhang. Stability of multiclass queueing networks under priority service disciplines. *Operations Research*, 48(1):26–37, 2000.

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

[5] 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*, 35(3), March 1990.

[6] Sunil Kumar and P. R. Kumar. Performance bounds for queuing networks and scheduling policies. *IEEE Transactions on Automatic Control*, 39(8):1600–1611, August 1994.

[7] 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*, 7(3), August 1994.

[8] Steve H. Lu and P. R. Kumar. Distributed scheduling policies based on due dates and buffer priorities. *IEEE Transactions on Automatic Control*, 36(12), December 1991.

[9] José A. A. Moreira. *Distributed Scheduling with Active Idleness: A key to the stabilization of multiclass queuing networks*. MSc Thesis, Instituto Superior Técnico, Technical University of Lisbon, 2001.

[10] Tomohito Nakata, Koichi Matsui, Yasuhisa Miyake, and Kyusaku Nishioka. Dynamic bottleneck control in wide variety production factory. *IEEE Transactions on Semiconductor Manufacturing*, 12(3), August 1999.

[11] S. S. Panwalkar and Wafik Iskander. A survey of scheduling rules. *Operations Research*, 25(1), 1977.

[12] Thomas I. Seidman. 'first come, first served' can be unstable! *IEEE Transactions on Automatic Control*, 39(10), October 1994.