Robust Multi-class Signaling Overload Control for Wireless Switches

J. Pinheiro
Novartis Pharmaceuticals
jose.pinheiro@pharma.novartis.com

C. Loader, T. LaPorta, S. Kasera, M. Karaul, A. Hari
Bell Labs Research
{cathl, tlp, kasera, karaul, hari}@lucent.com

Abstract

Mobile Switching Centers (MSCs) in wireless telecommunication networks must implement signaling overload controls to maintain acceptable performance during periods of high load especially due to fast changing hot spots resulting from user mobility. An MSC's signaling load comprises a variety of service requests, some more important than the others, and it must treat them differentially under overload conditions. We propose multi-class overload control algorithms that are highly reactive to sudden bursts of signaling load *and* maintain high performance under a wide range of overload conditions as well.

First, we introduce the concept of *equivalent system load measures* to convert the system load measures associated with different classes of traffic into a single measure with respect to a pre-defined *base class*. We use this concept to develop three multi-class overload detection and measurement algorithms. Next, we develop a new algorithm for partitioning the allowable equivalent system load across multiple traffic classes, using a strict priority scheme. Using simulations of call flows from mobile telecommunications standards, we compare different multi-class overload algorithms under a variety of overload conditions. Our simulation results indicate that our algorithm that measures system load using a combination of request acceptance rate and processor occupancy, provides highly reactive and robust overload control.

Last, for the purpose of making the overload control algorithms more robust, we propose a measurement-based simple regression technique to dynamically estimate key system parameters. We find that estimates derived in this manner converge rapidly to their true values.

I. Introduction

The heart of mobile telecommunication networks are the Mobile Switching Centers (MSCs). MSCs perform both control and transport functions. Control functions include routing calls to and from mobile users through the access radios, processing authentication of location update requests from mobile users, accessing user profi les so that advanced services may be invoked, and providing access to data services such as short messaging services. These control services are requested and invoked through the exchange of application level signaling protocols.

These switches are engineered to support a certain number of active calls and to process requests for calls and services at a certain rate. Occasionally, service requests may arrive at a switch at higher rate than the switch can process, even

S. Kasera is the correspondence author. This paper has been submitted for review.

if a switch has available bearer resources. This is possible under many conditions in a multi-service wireless network, for example a UMTS network, in which many requests to the network are for services other than connections, such as location updates. In this paper, we address processor overload controls due to signaling traffic. These controls execute inside MSC controllers to react to conditions of high signaling requests that overwhelm internal switch processing resources. They are required to maintain system integrity and maximize performance during overload. The lifetime of MSCs is many years, and processor upgrades can only be made within tight power budgets and floor-space constraints which are major cost factors in running a telecommunication network. Therefore, properly engineering an MSC for expected traffic, and relying on overload control algorithms to manage unexpected bursts of requests is critical to providing a cost effective service. These controls may also be used for load balancing and traffic differentiation.

Due to the ever growing demand for faster, ubiquitous, and newer services, mobile telephony poses new challenges in the design of signaling overload control algorithms for an MSC. First, user mobility causes fast changing hot spots resulting in bursty signaling traffic. Second, user mobility management and newer services result in a large number of signaling messages that require processing in an MSC. Third, some service requests might be more important than others, thereby requiring differential treatment under overload. Multi-class overload controls, as opposed to single class overload controls, are critical to multi-service wireless networks. The importance of service requests for basic users may vary from requests to establish calls, which generate immediate revenue, to registrations which may generate revenue in the future. Furthermore, these networks will provide multiple services, including voice, data, and messaging, each of which may generate different revenues for operators. By instituting multi-class overload controls, operators may maximize their revenue during overload periods.

Under overload, a switch may invoke *remote* overload control or congestion control by signaling its neighbors of its state. In practice, although recommended procedures exist to deal with inter-switch congestion control, many deployed switches do not implement these algorithms. Therefore, it is critical that a switch makes no assumptions about any remote overload control and *locally* protect its own processing resources by implementing appropriate local overload controls. This is usually done by throttling a fraction of the *new* service requests. In the multi-class case when certain service requests are more important than the others, the throttling must honor the class priority.

In this paper, we propose multi-class *local* overload control algorithms that are designed to be robust against different input traffic patterns (gradual, bursty) and system upgrades. Generally, overload control algorithms involve measurement of the system load and its comparison with a specified or measured system load threshold. When the system load threshold is reached or exceeded, overload control actions are triggered. In order to appropriately measure the system load when several classes of traffic are present, we first introduce the concept of *equivalent system load measure*, that converts the multiple system measures associated with different classes of traffic into a single measure with respect to a pre-defined *base class*. For example, arrival rates corresponding to calls, location updates, registrations, and short messages may be converted into a single, equivalent arrival rate. We use the concept of equivalent system load measure to develop three multi-class overload detection and measurement algorithms.

The first algorithm (denoted "Occupancy") uses processor occupancy to measure system load. The second algorithm (denoted "ARO") uses acceptance rate in conjunction with processor occupancy to measure the system load. Acceptance rate is defined to be the number of service requests accepted into the system in a given time interval. The third algorithm (denoted "SRED") measures the system load using queue lengths. Each of the three algorithms determine how much equivalent load should be allowed into the system at any given time. We develop a new procedure for partitioning the allowable equivalent load across traffic classes, using a strict priority scheme that assigns preferential treatment to higher priority classes. Our partitioning procedure is common to all the three algorithms.

Using simulations of call flows from mobile telecommunication standards, we investigate the performance of the three algorithms. We find that all three algorithms are capable of sustaining good throughput and delay performances under a wide range of steady overload conditions. Interestingly, under sudden load ramp up, ARO and SRED dramatically reduce the reaction time and the maximum average delay in comparison to the Occupancy algorithm. In comparison to SRED, ARO has a slightly higher reaction time and maximum average delay under sudden load ramp up, but provides more stable feedback under heavy overload, leading to a throttling pattern that is more consistent with the desired priority scheme.

Our algorithms require the relative processing cost of each class of traffic. Knowledge of switch behavior may allow values to be specified, but it is important to know how robust the algorithms are when the relative processing costs are mis-specified. We find that mis-specification can cause some instability in the algorithms, particularly if the cost of a high priority event is under-specified. For the purpose of making the algorithms more robust, we propose a scheme that does not require prior specification, but instead uses simple regression techniques to obtain estimates of relative processing costs using measured data. We find that estimates of relative costs derived in this manner converge rapidly to their true values as additional data accumulates.

The rest of the paper is structured as follows. Section II describes related work in multi-class overload control algorithms. A high level description of a wireless access switch and the system model used is presented in Section III. The multi-class overload algorithms considered in the paper are described in Section IV and compared via simulations in Section V. We discuss implementation issues in Section VII. We conclude in Section VIII and also provide several directions for future extensions to our work.

II. Related Work

Overload and congestion control has been actively studied in both telecommunication networks and the Internet. In telecommunication networks, single class overload controls have been proposed and evaluated in [12], [1], [5], [6], [15], [18], [17], [19]. One common overload control technique standardized and used in telecommunications networks is call gapping in which all service requests are rejected during certain time gaps. The period and duration of the gap are tuned based on overload conditions. To date, no analysis of multi-class overload controls using call gapping exist. Another common technique used is call percentage blocked in which a certain percentage of service requests are blocked during overload periods, but the remainder are serviced. The percentage blocked is tuned based on the level of overload. We have designed our algorithms in the spirit of percentage blocked algorithms, but they can be applied to call gapping algorithms as well. In the remainder of this section, we consider work related to multi-class overload controls only.

In [9] the authors developed analytical approximations for the queue length distributions in the buffers of a multi-queue, multi-priority traffic. In addition to the difference due to using multiple queues, this work differs from ours in its use of tail drop mechanism under overload. In a tail drop mechanism new arrivals are dropped when the queue length exceeds a certain threshold. The tail drop mechanism suffers from synchronization effects in which remote switches might reduce, or increase, sending traffic at the same time, causing degraded performance. Even though we do not study the synchronization effects in this paper, our dropping mechanism (first proposed in [11]) prevents them.

In [2] the authors examined a multi-class token-based scheme where a newly arrived call of a given class is accepted when there is free token of that class or when there is free token in a common pool of unused tokens. One concern with this scheme is that there is no priority mechanism in using the tokens of the common pool and under overload, calls belonging to lower priority classes might exhaust these tokens. Here too, the drop mechanism is tail drop. In [16] Rumsewicz used different queue size thresholds, one for each class, for discarding call messages when the thresholds are exceeded in a tail drop manner. He also observed that call completion throughput rather than message throughput is a better measure of performance. We also use call completion throughput as one of our performance measures.

In the Internet, overload control has manifested itself in active queue management of router queues. Most of the proposed multi-class active queue management approaches including RIO [4], WRED [20] and [3] are multi-class extensions of the *Random Early Discard* (RED) algorithm [10]. In these approaches, different average queue size thresholds are set for different classes and packet drop probabilities are computed based on how the actual average queue length(s) compare with the thresholds. It is assumed that the link, rather than the processor, is the bottleneck. In order to study the applicability of RED-like approaches in the context of controlling processor overload, we studied a modified version of the RED of RED, called signaling RED, or SRED[1] in [12]. In this paper we extend SRED for

[1] The acronym SRED has also been used in [14] for a different scheme called Stabilized RED.

handling multiple classes of signaling traffic. The performance of RED, RIO, WRED has been mainly evaluated in the presence of TCP traffic with TCP's remote overload control (end-to-end congestion control). In evaluating our algorithms, we do not assume any remote overload controls.

Our algorithms and their evaluation extend and build upon the earlier work on single class signaling overload control [12]. The main enhancements and differences from [12] are as follows. First, we consider multiple classes of traffic and introduce the concept of *equivalent system load measures* to design multi-class overload control algorithms. We introduce a new algorithm for partitioning load across multiple classes of traffic. Second, we evaluate our algorithms for a wireless scenario. Last, we propose a simple scheme that uses regression techniques to estimate relative processing cost of calls belonging to each traffic class.

We end this section by noting that several papers on handling multi-class traffic have been written in the context of admission control where new calls or requests are admitted or dropped depending upon their requirements and the available bearer resources. This differs from our work which focuses on controlling processor overload due to the signaling messages themselves.

III. System Description and Model

In a wireless network, service requests are processed through the cooperation of several network elements, each of which performs a fixed set of functions. For example, MSCs provide local processing for call requests, location updates, registrations, messaging services, etc. They provide these services by interacting with other network elements, such as Home Location Registers (HLRs) which store user profiles, track user mobility, and provide authentication services. Each element in a wireless network requires its own local overload controls since they receive service requests from many different sources. For example, MSCs receive requests from mobile devices as well as all HLRs in the network. For this reason, we evaluate overload control algorithms within a single network element. We choose an MSC because it handles many types of service requests, and because it services users based on location, it is particularly susceptible to transient overload periods.

Figure 1 shows a mobile network with an MSC. The MSC receives both control as well as bearer traffic from base stations on one side and the Public Services Telephone Network (PSTN) on the other side. The functional components of an MSC [13] are shown in Figure 2.

Typically, the processing in a switch is distributed between line cards and processing cards [17]. The signaling component, which terminates the protocol interfaces with the network, is implemented on line cards. These line cards exchange internal control messages with the processing cards which perform the call processing and Visitor Location Register (VLR) functions. The call processing functions include call routing, invoking services, and maintaining call state.

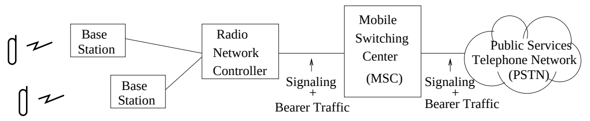

Base Station
Base Station
Radio Network Controller
Signaling + Bearer Traffic
Mobile Switching Center (MSC)
Signaling + Bearer Traffic
Public Services Telephone Network (PSTN)

Fig. 1. A Simple Mobile Network

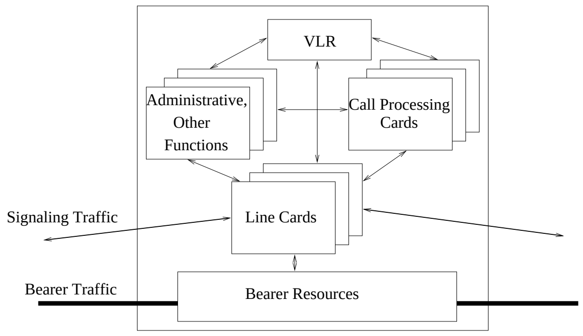


Fig. 2. A Typical Mobile Switching Center

We are concerned with signaling overload control at the line and the call processing cards. Signaling traffic arrives in the form of new requests, messages associated with established requests, or those associated with requests that are in the process of getting established. It is a standard practice to throttle new requests under overload rather than throttling a message associated with an established request or that associated with a request in the process of getting established. This allows elimination of future signaling messages associated with the throttled requests. When a signaling message arrives at a line card, the line card must first perform lower layer protocol processing and any other processing to determine if the signaling message is a new call request. Hence even if a new call is throttled due to overload in the line card, it imposes some processing cost on the line card. This processing cost is called the throttling cost. A call processing card may avoid this cost by communicating its load (or the rate at which it would accept calls) to the line card(s) which identifies and throttles new calls appropriately. The division of functionality across cards varies in different switch implementations.

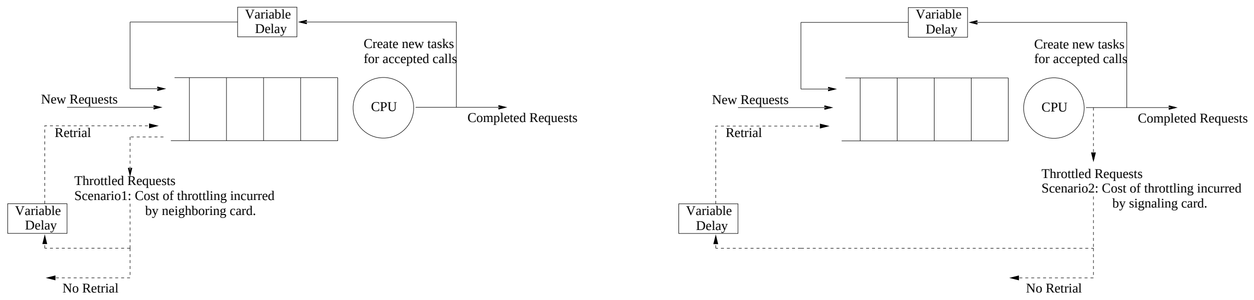


Fig. 3. Single processor queue model for a card.

We model a card, also referred to as *system*, by a single processor queue as shown in Figure 3. In this model, new requests belonging to different priority classes are assumed to arrive at the queue. For example, in the case of the MSC we could potentially have five different priority classes of requests - mobile calls, power-up registrations, location updates, paging requests, and SMS (short messaging service) requests. Depending on the current measure of load of the system, the class priority, and the overload control algorithm, a new request could be accepted or throttled. If the new request is accepted, it is processed by the CPU and, depending on the nature of the request, an additional task is generated and fed back into the system after a variable delay. This additional task, when processed, may generate more tasks which are fed back into the system. The processor schedules tasks using a first-in-first-out policy. Eventually, when all the tasks associated with an accepted request are executed, the request is considered completed and removed from the system. The generation of new tasks models the multiple tasks and messages associated with a request. The variable delay applied to a task, before it is fed back into the system, models the delay between arrival of different tasks associated with a request. Two throttling scenarios are shown in Figure 3. In the first scenario, a new call is throttled in a neighboring card and hence is shown to be throttled even before it is queued. In the second scenario a new call is throttled after incurring some processing cost. Due to user retrials a throttled request could arrive again at the system after a variable amount of delay. In this paper, we will only examine the first throttling scenario and assume that the call retrials are part of the new calls. The correlation between throttled calls and the retrials is not considered. Retrials cause overload to persist for a longer period of time. We do consider this persistence in our evaluation in Section V.

IV. Multi-class Overload Algorithms

In this section we develop overload control algorithms for multiple classes of signaling traffic representing tasks of multiple types of service requests. Under normal conditions, when the processor is not in overload, all the requests are allowed into the system. When the processor is in overload, some requests are throttled. In this paper we consider the scenario where service requests are arranged in *strict priority* order and under overload lower priority service requests are throttled first. An alternate approach would be to allow some kind of *fair access* to different service requests but we do not consider that approach in this paper.

As mentioned earlier, the process of throttling tasks could be done intelligently by throttling only those tasks that initiate service requests. It is assumed that ongoing events cannot be terminated to reduce overload.

All multi-class overload control algorithms considered in this paper are comprised of three steps.

1. A *Time-driven* measurement and detection mechanism in which system load is measured at fixed *probe times*, with overload being assessed at every k^{th} probe interval, denoted the *assessment time*. The detection mechanism produces a total fraction of the incoming service requests that must be allowed into the system.
2. An allocation mechanism, in which the total fraction to be allowed is converted into fractions for individual classes, in a manner that respects class priority.
3. A throttling scheme, which converts the fractions allowed into accept/reject decisions for individual events in each class.

The algorithms presented below differ only with respect to the first step where different measures are used for detecting overload; the allocation and throttling mechanisms are common to all algorithms.

In the rest of this section we introduce the concept of *equivalent measures* used in three multi-class overload detection and measurement algorithms (Step 1) described below. This is followed by a description of the allocation procedure (Step 2) and the throttling scheme (Step 3).

A. Equivalent System Load Measure

Equivalent system load measure allows us to convert the multiple system load measures associated with different classes of traffi c into a single load measure with respect to a pre-defi ned base class as described below.

It is assumed that there are M types of *origination* events that start a new thread of tasks (*e.g.*, new calls, location updates, registrations), which are assumed to be the only events that can be throttled. These origination events arrive at rates $\lambda_i,\ i = 1, \ldots, M$ events/sec and are processed by a single card at rates $\mu_i,\ i = 1, \ldots, M$. The corresponding average processing times μ_i^{-1} comprise the complete processing of the events, including all subtasks they generate, up to their termination.

It is also assumed that a *strict throttling priority* $p_i,\ i = 1, \ldots, M$ has been established for each event class, such that, if $p_i < p_j$, then events of class i should be throttled before events of class j. A *processing* priority may also be established for the event classes, but, for simplicity of implementation, we consider that all events have the same processing priority and are processed according to a FIFO scheme.

Let c_i represent a generic system measure for class i based on *counts*. Examples of such count measures are arrival rates, acceptance rates (*i.e.*, the number of events accepted into the system per unit time), and queue lengths (*i.e.*, the number of events waiting for processing).

One possible way of combining the different class count measures would be to simply add the c_i, but this is clearly not reasonable for certain load measures, as events in different classes generally have very different processing times. For example, if a location update takes, on average, 10% of the average processing time of a call, then accepting ten new location updates into the system is equivalent, in terms of processing effort, to accepting one new call. Therefore, a weighted average of the c_i, with the weights proportional to the processing times of each class, is used to combine the different class count measures into a single, *equivalent* count measure. For that, we choose a class, say the M^{th}, to serve as a base class and defi ne the *relative cost* of processing an event of class i with respect to an event in the base class as $\pi_i = \mu_i^{-1}/\mu_M^{-1}$. The equivalent count measure is then defi ned as

$$c^{\mathrm{eq}} = \sum_{i=1}^{M} \pi_i c_i. \tag{1}$$

This measure can be interpreted as the equivalent number of counts (in terms of processing cost) if the system were receiving only base class events. In order to determine π_i's we require the relative processing cost of each class of traffi c. A measurement-based scheme for estimating these costs is derived in Section VI-B.

B. Overload Measurement and Detection

In this section, we describe three algorithms that use equivalent measures of system load and compare it with a corresponding equivalent system load threshold to detect and measure overload. The goal is to produce an equivalent fraction of the input load that can be allowed into the system. The three algorithms we describe are multi-class extensions of the single class algorithms described in [12]. The fi rst algorithm is based on measurement of processor occupancy, the second is based on a combination of processor occupancy and acceptance rate measurements, and the third is based on measurement of queue lengths.

B.1 Occupancy Algorithm

Processor occupancy, ρ, is defi ned as the percentage of time, within a given probe interval, that the processor is busy processing tasks. Processor occupancy is a dimensionless quantity, which makes it relatively system independent. The measurement of processor occupancy is clearly independent of the number of event classes present in the system. This is because the total processor occupancy is the sum of individual processor occupancies. Therefore, the measurement and detection part of the Occupancy algorithm are same in the single-class and multi-class cases.

In the Occupancy algorithm, the estimated processor occupancy at assessment interval n, denoted by $\hat{\rho}_n$ and given by the average of the previous k probed processor occupancies, is compared to an occupancy threshold ρ_{thresh}. When the measured $\hat{\rho}_n$ is greater than ρ_{thresh}, the system is considered to be in overload, and throttling is forced. When $\hat{\rho}_n$ is less than ρ_{thresh}, no throttling is required. The specifi c feedback function for the equivalent fraction allowed (similar to the single class feedback function proposed in [5] and later examined in [12]), is

$$f^{\mathrm{eq}}_{n+1} = \mathrm{restrict}\left(f_{\min}, \frac{\rho_{thresh}}{\hat{\rho}_n} f^{\mathrm{eq}}_n, 1\right) \tag{2}$$

where $\mathrm{restrict}(a, x, b) = \max(a, \min(x, b))$ bounds x to the interval $[a, b]$. A minimum fraction allowed $f_{\min}$ is used to prevent the system from throttling all incoming events. In the single-class case the fraction allowed is applied directly to the individual events; in the multi-class case, it must be split into separate fractions allowed for the individual classes depending on their priorities. The mechanism for achieving this is described in Section IV-C.

B.2 ARO Algorithm

One of the problems with using an occupancy-based algorithm is that processor occupancy provides only a measure of the processed load. The offered load in a given time interval might be much higher than the processed load especially when there is a sudden burst of service requests. The acceptance rate, defi ned to be the number of new service requests accepted into the system in a given time interval, is a better measure of offered load. An algorithm based on the acceptance rate (AR) uses a feedback function similar to (2), but here the average number of accepted service requests over the past k probe intervals replaces the processor occupancy. In our multi-class scenario, the equivalent acceptance rate is obtained using (1). That is, letting $\hat{\alpha}_{n,i}$ represent the estimated acceptance rate for events in class i at assessment time n, the equivalent acceptance rate is $\hat{\alpha}_n^{\text{eq}} = \sum_{i=1}^{M} \pi_i \hat{\alpha}_{n,i}$. It estimates the equivalent number of class M events being accepted into the system. Letting $\alpha_{thresh}^{\text{eq}}$ denote the equivalent acceptance rate threshold (whose determination is discussed later in this section), the multi-class AR feedback function is

$$f_{n+1}^{\text{eq}} = \text{restrict}(f_{\min}, \frac{\alpha_{thresh}^{\text{eq}}}{\hat{\alpha}_n^{\text{eq}}} f_n^{\text{eq}}, 1). \tag{3}$$

The equivalent acceptance rate threshold $\alpha_{thresh}^{\text{eq}}$ is the most crucial parameter in the multi-class AR feedback function (3). It may either be determined based on the engineered capacity of the system, or it may be estimated dynamically, by determining the equivalent maximum system throughput $\mu_{\max}^{\text{eq}} = \hat{\alpha}^{\text{eq}}/\widehat{\rho}$, where $\hat{\alpha}^{\text{eq}}$ is the current estimate of the equivalent acceptance rate and $\widehat{\rho}$ is the current estimate of the processor occupancy. We then set $\alpha_{thresh}^{\text{eq}} = \rho_{thresh}\mu_{\max}^{\text{eq}}$. For robustness against sudden traffi c bursts, $\alpha_{thresh}^{\text{eq}}$ is updated at every $K \gg k$ probe intervals, according to an exponentially weighted moving average (EWMA) scheme with small updating weight w.

The problem with using a detection mechanism based only on acceptance rate is that this measure does not indicate overload induced by internal changes in the system. For example, an increase in the service rate for certain classes of events or consumption of processing resources by background tasks can not be captured by acceptance rate. Therefore, it is necessary to combine acceptance rate with another system measure that represents the system's processed load and not just offered load. This leads us to consider the combined acceptance rate and occupancy (ARO) algorithm,

$$f_{n+1}^{\text{eq}} = \text{restrict}\left[f_{\min}, \min\left(\frac{\alpha_{thresh}^{\text{eq}}}{\hat{\alpha}_n^{\text{eq}}}, \frac{\rho_{thresh}}{\hat{\rho}_n}\right) f_n^{\text{eq}}, 1\right].$$

B.3 SRED Algorithm

The SRED algorithm is based on the average queue length of new events. We use (1) to obtain the equivalent queue length $q^{\text{eq}} = \sum_{i=1}^{M} \pi_i q_i$, where q_i represents the length of the queue of originating events of class i[2].

The average equivalent queue length at time $n+1$ is estimated according to an EWMA based on the measured equivalent queue length at time n, q_n^{eq}, $Q_{n+1}^{\text{eq}} = (1-w)Q_n^{\text{eq}} + wq_n^{\text{eq}}$, where the updating weight w needs to be specifi ed.

[2]We consider the equivalent queue of origination events instead of equivalent queue of all events for the following reasons.

1. There is no simple defi nition of an equivalent non-originating event as the relative costs used refer to the complete processing of the event and the events that are processed by it.
2. In our scheme, only new service requests (originating events) can be throttled, so there is a direct relationship between the queue length and the amount of throttling, which would be indirect if all events were considered.
3. The SRED algorithm is more reactive when only originating events are used in the queue because lower threshold can be used, and more immediate changes in the queue length are produced.

The equivalent fraction allowed for the multi-class SRED algorithm is then defined as

$$f_{n+1}^{\text{eq}} = \begin{cases} f_{\min}^{\text{eq}}, & Q_n^{\text{eq}} \geq Q_{\max}^{\text{eq}} \\ 1, & Q_n^{\text{eq}} \leq Q_{\min}^{\text{eq}} \\ \max\left[f_{\min}^{\text{eq}}, \frac{Q_{\max}^{\text{eq}} - Q_n^{\text{eq}}}{Q_{\max}^{\text{eq}} - Q_{\min}^{\text{eq}}}\right], & \text{otherwise} \end{cases},$$

where $Q_{\min}^{\text{eq}}$ and $Q_{\max}^{\text{eq}}$ are lower and upper thresholds on the equivalent queue length, which need to be specified or could also be obtained from specifications of minimum and maximum queue length thresholds of each class using (1). Note that unlike the Occupancy and ARO algorithms, SRED does not use an explicit closed feedback loop for updating f_n^{eq}.

C. Allocation

The detection step of the overload algorithms, discussed in Section IV-B, determines the equivalent fraction of *class* M events that should be allowed into the system. Let $\tau^{\text{eq}} = 1 - f^{\text{eq}}$ be the the equivalent fraction of *class* M events that should be throttled. In the multi-class case, this throttling fraction needs to be split among the various event classes, respecting the throttling priority specified for them. We now describe a simple algorithm for calculating throttling fractions $\tau_1, \ldots, \tau_N$ for each priority, so that the equivalent throttling fraction is τ^{eq}.

Recall the arrival rate for class i events is λ_i, and the relative processing cost with respect to the base class M is π_i. The total requested load, per unit time, for class i events is therefore $\lambda_i \pi_i$, and the total requested load for the entire system is $\Lambda_M = \sum_{i=1}^{M} \lambda_i \pi_i$.

Since the required equivalent throttling fraction is τ^{eq}, the load that must be throttled is $\tau^{\text{eq}} \Lambda_M$. The throttling scheme then assigns individual throttling fractions to each class, so that the low priority classes are throttled first, and to maintain the desired overall throttling fraction. Formally, define $\Lambda_0 = 0$ and $\Lambda_k = \sum_{j=1}^{k} \lambda_i \pi_i$; note that Λ_k is the cumulative requested load for classes 1 to k. The throttling fraction τ_j for class j is

$$\tau_j = \begin{cases} 1, & \Lambda_j \leq \tau^{\text{eq}} \Lambda_M \\ \left(\tau^{\text{eq}} \Lambda_M - \Lambda_{j-1}\right) / \lambda_j \pi_j, & \Lambda_{j-1} \leq \tau^{\text{eq}} \Lambda_M < \Lambda_j \\ 0, & \Lambda_{j-1} \geq \tau^{\text{eq}} \Lambda_M \end{cases} \tag{4}$$

The allocation algorithm described in (4) requires that some estimates of the class arrival rates, λ_i, be available. We use separate EWMAs in each measured class arrival rate, which, for robustness, are updated at each $K_a \gg k$ probe intervals, using a small updating weight w_a.

D. Throttling

Given the throttling fraction $\tau_{n,i}$ to be used for class i during the nth interval, the throttling scheme determines which new events to drop. We adopt a deterministic throttling scheme first proposed by [11]. In this algorithm, a variable r is first initialized to 0, then the accept/reject decision procedure described below is used

```
$r := r + (1 - \tau_{n,i})$.
If $r \geq 1$
    $r := r - 1$
    accept request
else reject request,
```

[12] compares this deterministic throttling algorithm to alternative schemes based on pseudo-random numbers and concludes that the deterministic algorithm has the best overall performance in terms of reducing the variability in the fraction allowed. The randomness in arrivals from different sources ensures that no particular source is able to misuse the deterministic throttling to its advantage.

V. Performance of Algorithms

We now evaluate the performance of the multi-class overload control algorithms described in Section IV by simulating the system model described in Section III. We make the simplifying assumption that the process of detecting overload is *free*. The simulator used to obtain the results presented in this section is custom written.

For simplicity, only two classes of events are considered in the simulation: *mobile originated calls* and *location updates*. As described in Section IV, strict priority is used in throttling events from the two classes. The task and event structures for calls and location updates, derived from call flows from mobile telecommunication standards, are presented in Figure 4. The call model treats each request as consisting of four task segments: initial request, call setup, handover, and call termination. The model used for location updates consists of a single task segment: the initial request. The initial request, call setup and call termination tasks comprise several subtasks which are generated after a random delay. For the purpose of the simulation, subtasks occurring with negligible delay of each other are combined into one subtask. This results in the initial request being subdivided into one or three subtasks, the call setup being subdivided into four subtasks and the call termination being composed of three subtasks. Only 10% of the initial request events require three subtasks; the other 90% require a single subtask[3]. In our simulations, 70% of the calls experience handover and the ratio of number of calls to the number of location updates has been assumed to be 1:10.

The system represented in the simulation is designed to operate under approximately 95% processor occupancy under a load of 592,000 busy hour calls attempts (BHCA), corresponding to an average of about 164 call attempts per second, together with ten times more location updates. The probability distributions used for the delays until the next subtask

[3]The choice of one or three subtasks depends upon whether the record of the user is available on the local VLR or if it must be obtained from a remote VLR or even the user's Home Location Register (HLR). Here, we assume that 90% of the time the user's record is found in the local VLR.

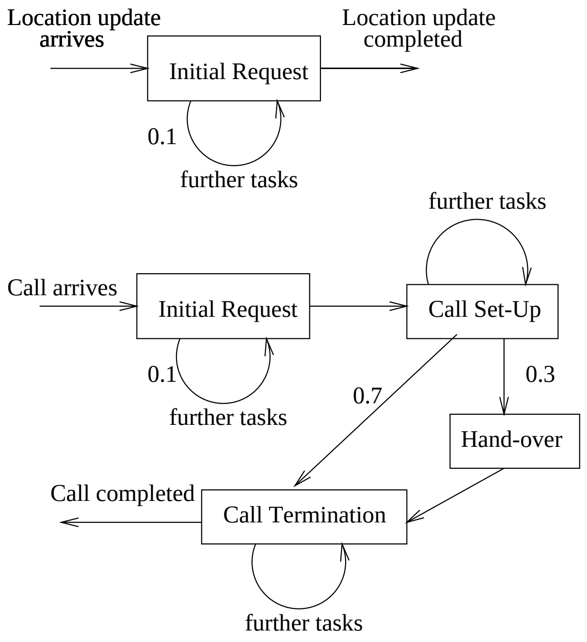


Fig. 4. Task and event structures for calls and location updates. Numbers represent branching probabilities.

and the subtask processing times are listed in Table I, with $\exp(\lambda)$ denoting the exponential distribution with parameter λ and $\Gamma(\alpha, \beta)$ denoting the Gamma distribution with parameters α and β.

The choice of distributions and parameter values in Table I, as well as the task and event structures in Figure 4, are based on telecommunications traffi c engineering recommendations and measurements on prototype implementations. Under these assumptions, the total average processing time is 2.89 ms per call and 0.29 ms per location update. The average relative cost of a location update with respect to a call is about 10%. The holding time for a call, that is, the time between the end of the call setup and the start of the call termination, is assumed to be exponentially distributed with mean 90 seconds.

All three overload algorithms considered in the simulation use the same probe intervals and assessment intervals of 100 ms, and the same minimum fraction allowed, $f_{\min} = 0.005$. Experience with evaluation of switches suggests that a timer value of a few hundred milliseconds is good for a variety of input traffi c patterns and system settings. Smaller timer values would result in faster response during sudden overload but also cause more oscillations during steady state leading to reduced performance. Table II lists the parameter values used in the simulations for the different algorithms. The parameter values for the SRED algorithm were chosen to produce an average processor occupancy of about 95% under mild to moderate overload conditions (call rates between 160 and 350 calls/s). As discussed in Section V-A, SRED becomes unstable under higher overload (call rates above 500 calls/s) and it is not possible to choose parameter values that give steady 95% occupancy for the range of call rates considered in the simulation.

The performance metrics used to compare the algorithms are *task delay* (time in queue until start of processing),

TABLE I

PROBABILITY DISTRIBUTIONS FOR DELAY UNTIL THE NEXT SUBTASK AND SUBTASK PROCESSING TIME.

	Subtasks			
	Subt.	Delay	Process.	Mean
Task	No.	Dist.	Dist.	(ms)
Init. Req.	1	$\exp(250)$	$\Gamma(2.5, 10)$	.25
	2	$\exp(250)$	$\Gamma(2, 10)$	.2
	3	$\exp(250)$	$\Gamma(2, 10)$	.2
Setup	1	$\exp(250)$	$\Gamma(3, 3)$	1
	2	$\exp(250)$	$\Gamma(2, 10)$	.2
	3	$\exp(7500)$	$\Gamma(2, 10)$	.2
(no HO)	4	$\exp(90000)$	$\Gamma(2, 10)$	.2
(with HO)	4	$\exp(45000)$		
HO	1	$\exp(45000)$	$\Gamma(3, 3)$	1
Term.	1	$\exp(250)$	$\Gamma(3, 10)$	.3
	2	$\exp(250)$	$\Gamma(2, 10)$	.2
	3	—	$\Gamma(2, 10)$	.2

TABLE II

PARAMETER VALUES USED IN THE SIMULATIONS.

	Algorithm		
Parameter	Occupancy	SRED	ARO
ρ_{thresh}	0.95	–	0.95
$\phi_{\max}$	20	–	20
k	3	1	3
w	–	0.05	0.02
K	–	–	300
$Q_{\min}$	–	1	–
$Q_{\max}$	–	4.5	–

throughput (number of service requests or calls completed per second), and *allowed fraction* of service requests (call origination or location update events) into the system.

A. Performance Under Steady Load

We investigate the performance of the overload control algorithms when call arrivals are Poisson[4], under steady mean call attempt rates varying between 450 thousand BHCA (125 calls/s) and 7.2 million BHCA (2000 calls/s), covering the range from no overload to severe overload. The location update mean arrival (also a Poisson process) rate was set to ten times the mean call rate. For each attempt rate, calls and location updates were simulated over a 15-minute period, with the performance metrics averaged over the whole period.

The sample size associated with the simulation depends on the performance metric under consideration: for *task delay* the sample size is determined by the number of simulated calls during the 15-minute period (which ranges from around 112,500 to 1,800,000, depending on the call load); for *throughput* and *allowed fraction* the sample size is determined by the number of probe intervals in the 15-minute period, 9000.

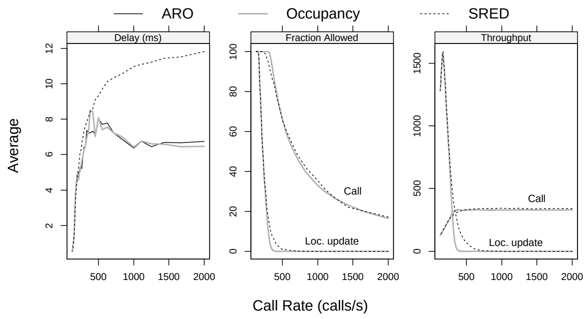


Fig. 5. Average performance metrics versus call attempt rate by overload control algorithm.

Figure 5 shows the averages of the performance metrics versus call attempt rate, for each overload control algorithm[5]. The sample sizes, mentioned above, give the following *maximum* simulation errors for the average performance metrics in this fi gure: ± 0.1ms for task delay, ± 1% for allowed fraction, and ± 3 calls (or location updates)/s for throughput[6]. All three algorithms show good, similar performance under multi-class overload conditions. The average delay stays within reasonable limits (below 12 ms), though SRED gives consistently higher delays for heavier overload conditions. The fraction of allowed events behave consistently with the strict priority scheme adopted: the fraction of allowed location updates drops faster than that of calls, staying close to zero for call rates $\geq$ 500 calls/s. Consistently, the average throughput curves indicate that no location updates are processed under heavier overload. The system is capable of

[4]Other arrival processes can be easily used in our simulator

[5]Because of their close proximity, the ARO curves have been overwritten by the Occupancy curves in some cases.

[6]In order to obtain the simulation errors, we had to make some assumptions about the distributions of these performance metrics. The details have been skipped for brevity.

sustaining a constant call throughput, under all three overload algorithms.

The fraction allowed and throughput plots in Figure 5 also indicate that SRED throttles location updates less aggressively than the other two algorithms, under moderate overload conditions (close to 500 calls/s). The reason for this is the greater instability observed for the feedback mechanism of this algorithm, which translates into a highly variable fraction allowed of calls. Figure 6 presents the inter-quartile ranges (*i.e.*, the difference between the third and the first quartiles, which provides a measure of variation for the variable under consideration) of the fractions of calls and location updates allowed, observed over time during the simulation, for the different call attempt rates. The SRED fraction allowed variation is about three times larger for location updates and ten times larger for calls. To further explore this issue we study the behavior of the algorithms over time.

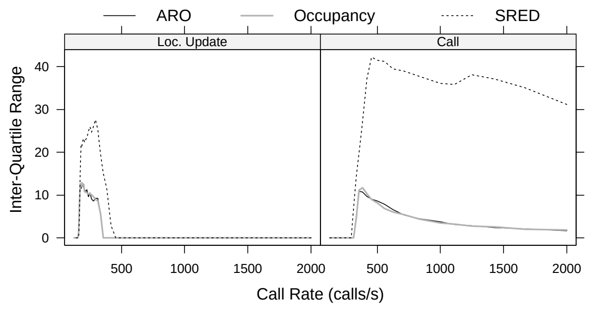
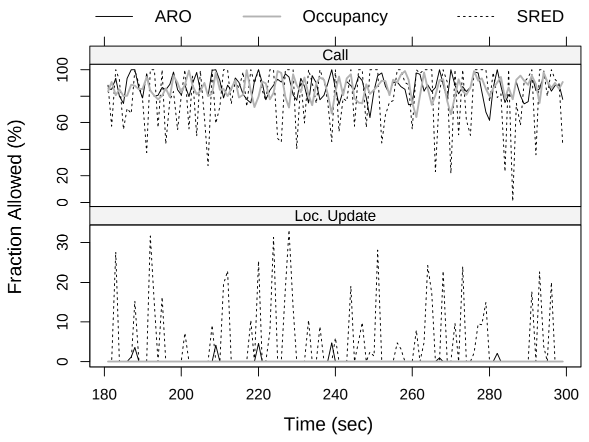


Fig. 6. Inter-quartile ranges of fraction allowed versus call attempt rate by overload control algorithm and event class.

We consider the case of a steady state call rate of 1.38 million BHCA (385 calls/s), considerably above the nominal capacity of the system. As before, the location update rate was set to ten times the call rate. Figure 7 shows the behavior of the fractions of calls and location updates allowed (given as averages per second), between the third and the fifth minutes of operation, for the three overload control algorithms.



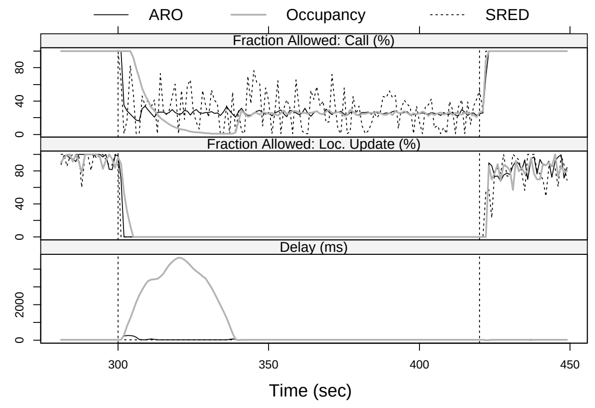


Fig. 7. Evolution of fractions allowed under a steady overload.

Fig. 8. Evolution of performance under non-steady overload.

The ARO and Occupancy algorithms show similar, stable fractions allowed with respect to both calls and location updates, which are consistent with the strict priority scheme. The SRED algorithm, however, presents quite unstable fractions allowed in both event classes, which, over time, leads to throttling that is not consistent with strict priority. The ARO and Occupancy algorithms show more consistent behavior under steady overload.

B. Performance Under Non-Steady Load

In order to study the reactiveness of the algorithms to sudden changes in arrival rates, we consider the scenario in which the call and location update processes operate at mean rates of, respectively, 167 calls/s and 1667 location updates/s (corresponding to a condition of non-overload) up to 300 seconds, at which point both rates experience an eight-fold increase over a period of 1.5 seconds, staying at that level for two minutes, and then dropping back to their original non-overload rates in 1.5 seconds. Note that the arrival process in each interval is still Poisson but with different mean arrival rates. The objective of this simulation is to study how fast the overload algorithms react to a sudden onset of overload and to a sudden cessation of overload. Figure 8 presents the evolution of the average delay and the fractions allowed for the three algorithms.

The SRED algorithm has the best overall performance with respect to average delay under this overload scenario, showing no significant changes in average delay during the onset of overload. The Occupancy algorithm has the worst delay performance, taking about 38 seconds to recover from the event rate ramp up and experiencing maximum delay of 4.5 seconds. The ARO algorithm has a much better performance than the Occupancy algorithm, but slightly worse than SRED: about 5 seconds to recover and a maximum delay of 260 milliseconds. The basic reason for the poorer performance of the Occupancy algorithm is that processor occupancy measures processed load which cannot be higher than 100% even when the offered load is very high. This reduces the processor occupancy measure's response to sudden overload in comparison to to either queue length (SRED) or call acceptance rate (ARO).

Once again, the SRED algorithm displays a marked increase in variability of fraction allowed (for call originations, in this case) under overload, which is not observed for the other two algorithms. All three algorithms show almost immediate recovery when the system goes from overload to non-overload.

In summary, under sudden load ramp up, ARO and SRED reduce the response time by an order of magnitude in comparison to the algorithm that uses processor occupancy only. In comparison to SRED, ARO experiences a slightly higher response time under sudden load ramp up but exhibits more stable fractions allowed under overload. We also investigate the behavior of the three algorithms under a variety of other simulation settings (including variations in Tables I and II, different load patterns, etc.) and find that the results are qualitatively the same as the ones presented here.

VI. Estimation of Processing Costs

The algorithms and simulations studied so far assume that the relative processing costs of events in different classes is known. In this section, we investigate the sensitivity of the algorithms to mis-specification of the relative processing costs, and derive reliable, measurement-based estimates of the relative processing costs.

A. Behavior under mis-specified costs

Figure 9 compares the performance of the algorithms when there is an abrupt increase in the rate of the low priority location updates. The scenario used in the simulations (the same for all three plots) is similar to the one described in Section V-B, but this time only the location update rate increases during overload: call and location update processes operate at rates of, respectively, 167 calls/s and 1667 location updates/s (corresponding to a condition of non-overload) up to 300 seconds, at which point the location update rate increases eight-fold to 13,333 location updates/s, over a period of 1.5 seconds, staying at that level for two minutes, and then dropping back to its original non-overload level in 1.5 seconds.

For the occupancy and ARO algorithms, the interpretation of these plots is straightforward. Under-specification of the cost of the (lower priority) location updates results in the algorithms overreacting to overload. The result is oscillation, and some new calls are unnecessarily rejected. When the cost is over-specified, the algorithms are more stable, but reaction to the onset of overload is slower. In all cases, SRED shows high variability in the fraction allowed, and correspondingly drops an unnecessary number of call originations.

B. Regression Estimates of Processing Cost

A reliable strategy for determining the relative costs of the various event types is to estimate them directly from the data collected at the MSC, under non-overload conditions. We use a simple, easy to implement methodology for estimating the relative costs using linear regression. The underlying idea for a regression model is to collect measurements over successive time intervals of the processor occupancy and event processing counts, and to use this data to model the behavior of processor occupancy as a function of the count data. The fitted model can then be used both to estimate the system capacity, and to estimate the relative costs of processing the different event classes.

Our regression model for estimating the relative costs of each originating event class is based on the relation between the overall processor occupancy and the processor occupancies associated with each event class.

$$\rho = \rho_0 + \sum_i \rho_i = \rho_0 + \sum_i \lambda_i / \mu_i \tag{5}$$

where ρ_0 represents the "background" processor occupancy when no call traffic is present, and ρ_i, λ_i, and μ_i^{-1} are, respectively, the processor occupancy, the arrival rate and the average processing time associated with the $i^{\rm th}$ event

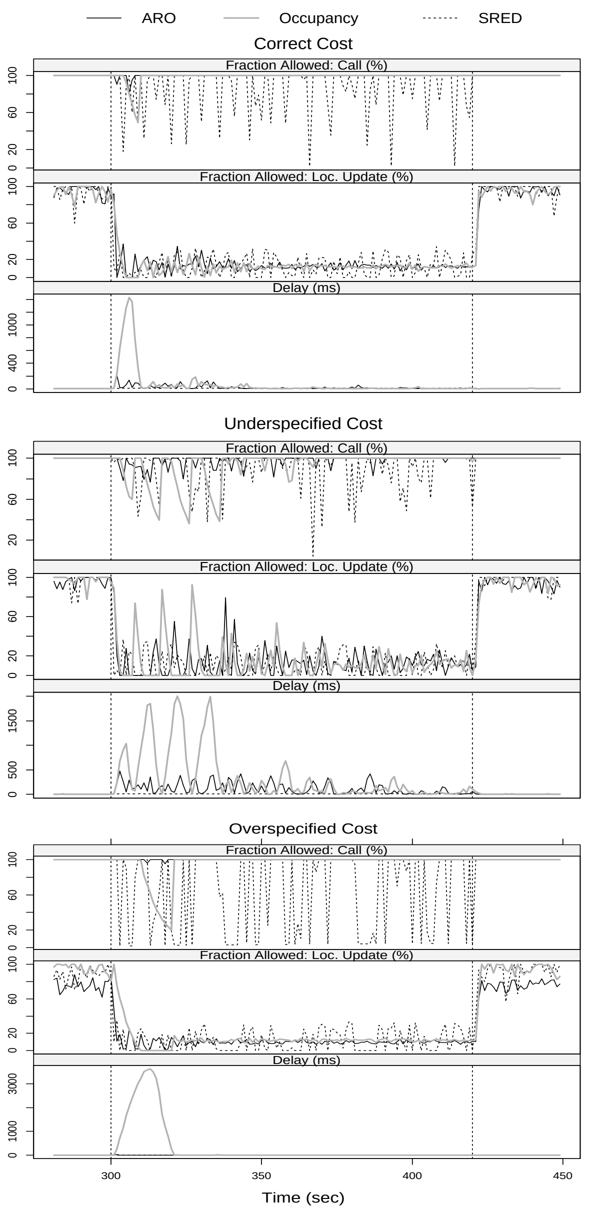


Fig. 9. Fractions allowed and average delay with a plateau in the location update rate. The costs are correctly specified (top); under-specified for location updates (middle) and over-specified for location updates (bottom).

class. Recall from Section IV-A that the relative costs π_i are proportional to the average processing times of the originating events. Note that (5) includes all events, not just the originating ones.

In order to estimate the relative costs using a regression model based on (5), measurements of overall processor occupancy and the counts of processed events are collected over successive time intervals. The counts are broken down by the type of event: the number of location updates; number of new call originations, handovers etc.. For each time interval n, we denote the observed variables as follows:

$$\begin{aligned} L_n &= \text{No. of Location Updates} \\ O_n &= \text{No. of New Call Originations} \\ H_n &= \text{No. of Handovers} \\ T_n &= \text{No. of Call Terminations} \\ \rho_n &= \text{Processor Occupancy} \end{aligned}$$

The number of events over each unit time interval represents an estimate of the arrival rate (λ_i). The following linear regression model is then used to explain the behavior of processor occupancy as a function of the different event counts during the nth time interval.

$$\rho_n = a_0 + a_1 L_n + a_2 O_n + a_3 H_n + a_4 T_n + \epsilon_n$$

where a_0, a_1, a_2, a_3 and a_4 are the regression coeffi cients to be estimated from the observed data representing, respectively, the background occupancy ρ_0 when no call traffi c is present and the average processing times μ_i^{-1} of event classes L, O, H, and T. The motivation for this model is that processor occupancy is expected to behave linearly in the amount of traffi c handled. The error term ϵ_n accommodates the stochastic nature of the observed data and represents sources of variation that cannot be controlled, variability in event processing time, measurement error, and any other random fluctuations.

The coeffi cients $a_0, \ldots, a_4$ are unknown, and must be estimated from the data. The regression coeffi cients should be estimated so that the regression model explains as much variation of ρ_n as possible. Equivalently, we want the "unexplained variation" represented by the error terms ϵ_n to be as small as possible. The regression coeffi cients are chosen to minimize the squared-error loss [7],

$$\sum_n (\rho_n - (a_0 + a_1 L_n + a_2 O_n + a_3 H_n + a_4 T_n))^2.$$

This minimization problem can be reduced to solving a system of linear equations.

As mentioned above, the regression model can include terms that do not correspond to the originating event classes. Here, we have included Handover and Call Termination. A mobile originated call consists of one origination, possibly some handovers, and one termination. The corresponding processing costs can be lumped together in a single event, as done in Section IV-A. However, when the events are collected over shorter time intervals, there may be an imbalance between originations and terminations, and the regression model provides a better fi t when these are treated separately. This leads to additional complications when converting the estimated coeffi cients to relative cost, as addressed below.

One could also extend the regression model by including separate terms for each subtask of call origination, for example. But this seems less benefi cial; since all subtasks will typically be completed within a small time interval, the imbalance between the counts of subtasks will generally be small, and there is less benefi t to treating these separately.

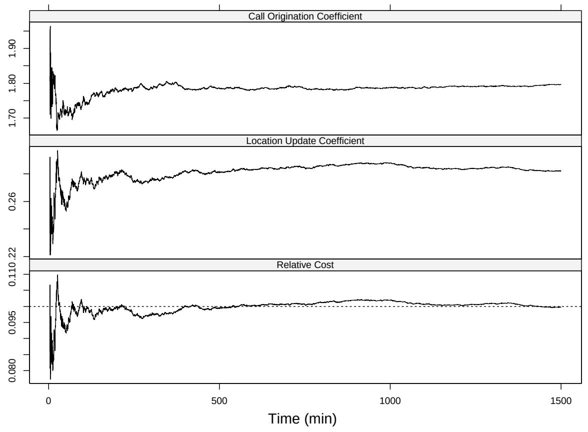


Fig. 10. Regression coefficients (in m.s. per event) for call origination (top) and location updates (middle); and estimated relative cost (bottom).

Figure 10 shows the behavior of regression coeffi cients a_1 and a_2 over a period of 1500 minutes, measured and updated every 10 seconds. The coeffi cients show the average cost of processing these events; they quickly stabilize around long-term asymptotic values. The main computational effort in updating the coeffi cients is the solution of a system of linear equations with fi ve variables.

The regression coeffi cient a_1 provides an estimate of the cost of a location update. The cost of a new call is more complicated, since it consists of a call origination, call termination and possibly one or more handovers. Let κ be the average number of handovers per call; this may be known from experience or estimated as a long-run average. The average cost of processing a call is then

$$a_2 + \kappa a_3 + a_4. \tag{6}$$

The relative cost of a location update is a_1 divided by the quantity (6). The third panel of Figure 10 shows this estimate stabilizing over time; the true value of 0.1 is shown for reference.

VII. Implementation Issues

In this section we discuss issues related to the implementation of SRED and ARO. The main challenge in implementing SRED (or RED) is the specifi cation of parameters for maximum and minimum queue lengths. The right choice requires tuning and will change with processor speed. Hence SRED is not robust against system upgrades.

Implementation of ARO requires measurement of acceptance rate, arrival rate and processor occupancy. Acceptance rate/arrival rate is easily measured by counting the number of requests accepted/arrived in a given time interval. Measuring processor occupancy is a hardware and operating system dependent task. Some Unix-like operating systems including Solaris and Linux allow applications to monitor CPU usage of individual processes by simply reading kernel data structures. However, line cards typically implement light weight real time embedded systems, which do not always provide straightforward ways to measure processor occupancy. While some embedded real time executives, for example VxWorks, provide utilities to measure processor occupancy on a per task basis, others do not provide such tools.

One way to measure occupancy in such cases is to launch a low priority idle task which increments a counter. By periodically measuring this counter, it is possible to determine the amount of time the idle process has run and conversely the processor occupancy. The main drawback of this method is that it does not distinguish between time spent in high priority signaling request processing tasks and lower priority background tasks. For example, a sudden run of a lower priority accounting task could cause a low value for the idle task count and a high value for the processor occupancy. Therefore this method of measuring processor occupancy needs additional validation by ensuring that a minimum threshold of signaling requests have arrived in a given interval. Note that this arrival rate measurement is already done in case of the ARO algorithm, making this an attractive way to measure processor occupancy in such systems.

There are also real time executives like pSOS which provide a *call-back* function that is called each time a task switch occurs. By providing a customized call-back function which records the real time clock and the current task each time it is invoked, it is possible to accurately measure the time spent in individual tasks at the cost of a little added overhead with each task switch.

Even though ARO requires measurement of multiple system measures, it requires specifi cation of only one parameter, the processor occupancy threshold. Processor occupancy is dimensionless and need not be changed with software and processor upgrades. The choice of a processor occupancy threshold should come from experience of the switch operator. Typically a switch operator would engineer the value of ρ_{thresh} such that there is enough room to process administrative tasks as well as sudden signaling congestion. However, if the threshold is exceeded most of the time than the switch operator must upgrade its system.

We end this section by noting that we have used a timer-based overload control instead of an event-based overload control where the system load is measured on every task arrival. The main reasons for this choice is to reduce processing overhead of system load measurement.

VIII. Conclusions and Future Extensions

We proposed and evaluated approaches for monitoring and controlling processor overload due to excessive signaling traffi c in a wireless access switch, involving multiple classes of traffi c. We developed the concept of equivalent system load measure to convert multiple system measures associated with different classes of traffi c into a single measure with respect to a pre-defi ned base class. We also proposed a new procedure for allocating the allowable equivalent load across multiple traffic classes, according to a strict priority scheme.

We found that our algorithm that measures system load using a combination of acceptance rate and processor occupancy, provides highly reactive and robust overload control. All the three multi-class overload control algorithms developed in this paper, require relative processing cost of each class of traffic. We studied the performance of these algorithms when these processing costs were mis-specified and derived measurement-based regression estimates of the relative costs. We also discussed implementation issues. Potential future extensions to our work are described below.

- *Throttling cost case* – In this paper, we have only considered and evaluated the case where no processing costs are associated with new service requests that are eventually throttled. Even though this is a realistic assumption for some of the cards in the MSC, it does not necessarily hold for all cards. If processing costs are associated with new service requests that are throttled (Scenario 2 in Figure 2), under heavy overload a substantial part of the processor occupancy might be due to throttling of new service requests and not due to processing of accepted calls. As a result, the processing power of the system might be reduced considerably. Hence schemes that appropriately adjust the system load threshold(s) with variations in the degree of overload need to be developed.
- *Network of Cards, Switches* – We believe that our single queue model can be easily extended to a network of queues, but this will require further research. There is also a need to study the local overload control algorithms examined in this paper in conjunction with remote overload control at other switches.
- *Weighted Fair Allocation Across Multiple-Traffic Classes* – The strict priority throttling scheme described in this paper may be impractical in situations where there is an interest in allowing events from all classes into the system, at possibly different, prioritized rates. A *weighted fair* allocation, would be preferred in such cases. Any allocation scheme producing fractions allowed $f_i,\ i = 1, \ldots, M$ for an equivalent fraction f^{eq} must satisfy

$$\sum_{i=1}^{M} f_i \lambda_i^{\text{eq}} = f^{\text{eq}} \lambda^{\text{eq}}. \tag{7}$$

A trivial solution to (7) is to set $f_i = f^{\text{eq}}$ for all classes, corresponding to the equal-priority case. Under weighted fair allocation, *priority weights* $w_i = p_i / p_{\text{min}}$ could be defined with respect to the smallest throttling weight p_{min} and a *base* fraction allowed f_{base} (corresponding to events of priority weight 1) could be obtained such that $f_i = \min(w_i f_{\text{base}}, 1),\ i = 1, \ldots, M$ and (7) is satisfied. An algorithm to obtain f_{base} needs to be developed.
- *Application of ARO in the Internet* – It would be interesting to study the robustness of our multi-class ARO algorithm for active queue management in router queues forwarding datagram traffic especially in conjunction with TCP congestion control. The application of our ideas could also be studied in the context of overload due to signaling traffic resulting from resource reservation mechanisms.

REFERENCES

[1] A. W. Berger, "Comparison of Call Gapping and Percent Blocking for Overload Control in Distributed Switching Systems and Telecommunications Networks," *IEEE Transactions on Communications*, vol. 39, pp. 574–580, 1991.

[2] A. W. Berger and W. Whitt, "The Brownian approximation for rate-control throttles and the G/G/1/C queue," *Journal of Discrete Event Dynamic Systems*, vol. 2, pp. 685–717, 1992.

[3] U. Bodin, O. Schelen, and S. Pink, "Load-tolerant differentiation with active queue management," in *In ACM Computer Communications Review*, July 2000.

[4] D. Clark and W. Fang, "Explicit allocation for best effort packet delivery service," *IEEE/ACM Transactions on Networking*, August 1998.

[5] B. L. Cyr, J. S. Kaufman, and P. T. Lee, "Load balancing and overload control in a distributed processing telecommunications system," United States Patent No. 4,974,256, 1990.

[6] B. T. Doshi and H. Heffes, "Analysis of overload control schemes for a class of distributed switching machines," *Proceedings of ITC-10*, Section 5.2, paper 2, June 1983.

[7] N.R. Draper and H. Smith, "Applied Regression Analysis," Wiley, 1998.

[8] A. I. Elwalid and D. Mitra, "Statistical multiplexing with loss priorities in rate based congestion control of high-speed networks," *IEEE Transactions on Communications*, vol. 42, no. 11, pp. 2989–3002, November 1994.

[9] A. I. Elwalid and D. Mitra, "Analysis, approximations and admission control of a multi-service multiplexing system with priorities," in *Proceedings of IEEE Infocom*, Boston, April 1995.

[10] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, August 1993.

[11] B. Hajek, "External splitting of point processes," *Mathematics of Operations Research*, vol. 10, pp. 543–556, 1985.

[12] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, and T. LaPorta, "Fast and Robust Signaling Overload Control," in *Proceedings of IEEE ICNP*, November 2001.

[13] T. LaPorta *et al*, "Cluster mobile switching center for third generation wireless systems," in *Proceedings of IEEE PIMRC*, September 1998.

[14] T.J. Ott, T. V. Lakshman, and L. H. Wong, "SRED: Stabilized RED," in *Proceedings of IEEE Infocom*, New York, March 1999.

[15] R. Pillai, "A distributed overload control algorithm for delay-bounded call setup," *IEEE/ACM Transactions on Networking*, vol. 9, pp. 780–789, 2001.

[16] M. Rumsewicz, "Analysis of the Effects of SS7 Messages Discard Schemes on Call Completion Rates During Overload," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 491–502, 1993.

[17] M. Schwarz, *Telecommunications Networks: Protocols, Modeling and Analysis*, Addison-Wesley, 1988.

[18] M. Rumsewicz, "Ensuring Robust Call Throughput and Fairness for SCP Overload Controls," *IEEE/ACM Transactions on Networking*, vol. 3, pp. 538–548, 1995.

[19] B. Wallstrom, "A Feedback Queue with Overload Control," *Proceedings of ITC-10*, Section 1.3, paper 4, June 1983.

[20] Technical Specifi cation from CISCO, "Distributed weighted random early detection," .