Multi-Resource Round Robin: A Low Complexity Packet Scheduler with Dominant Resource Fairness

Wei Wang, Baochun Li, Ben Liang
Department of Electrical and Computer Engineering
University of Toronto
{weiwang, bli}@eecg.toronto.edu, liang@comm.utoronto.ca

***Abstract*—Middleboxes are widely deployed in today's enterprise networks. They perform a wide range of important network functions, including WAN optimizations, intrusion detection systems, network and application level firewalls, *etc*. Depending on the processing requirement of traffic, packet processing for different traffic flows may consume vastly different amounts of hardware resources (*e.g.,* CPU and link bandwidth). Multi-resource fair queueing allows each traffic flow to receive a fair share of multiple middlebox resources. Previous schemes for multi-resource fair queueing, however, are expensive to implement at high speeds. Specifically, the time complexity to schedule a packet is $O(\log n)$, where n is the number of backlogged flows. In this paper, we design a new multi-resource fair queueing scheme that schedules packets using Elastic Round Robin (ERR). Our scheme requires only $O(1)$ work to schedule a packet and is simple enough to implement in practice. We show, both analytically and experimentally, that our queueing scheme achieves nearly perfect Dominant Resource Fairness.**

I. Introduction

Network appliances or "middleboxes" are ubiquitous in today's networks. Recent studies report that the number of middleboxes deployed in enterprise networks is on par with the traditional L2/L3 devices [1], [2]. These middleboxes perform a variety of critical network functions, ranging from basic operations such as packet forwarding and HTTP caching to more complex processing such as WAN optimization, intrusion detection system (IDS) and firewalls.

As the traffic through middleboxes surges [3], it is important to have a scheduling discipline that provides *predictable service isolation* for flows passing through them. Although traditional fair queueing allows flows to have a fair share of the output bandwidth [4], [5], packet scheduling in a middlebox is more complicated because flows are competing for multiple hardware resources (*e.g.,* CPU, memory bandwidth, and link bandwidth) and may have vastly different resource requirements, depending on the network functions they go through. For example, forwarding a large amount of small packets of a flow via software routers congests the memory bandwidth [6], while performing intrusion detection for external traffic is CPU intensive. Despite the heterogeneous resource requirements of traffic, flows are expected to receive predictable service isolation. This requires a *multi-resource fair queueing scheme* that makes scheduling decisions across all middlebox resources. The following properties are desired.

Fairness: The middlebox scheduler should provide some measure of service isolation to allow competing flows to have a fair share of middlebox resources. In particular, each flow should receive the service at least at the level when *every resource* is equally allocated (assuming flows are equally weighted). Moreover, this service isolation should not be compromised by strategic behaviours of other flows.

Low complexity: With the ever growing line rate and the increasing volume of traffic passing through middleboxes [3], [7], it is critical to schedule packets at high speeds. This requires low time complexity when making scheduling decisions. In particular, it is desirable that this complexity is a small constant, independent of the number of traffic flows. Equally importantly, the scheduling algorithm should also be amenable to practical implementation.

While both fairness and scheduling complexity have been extensively studied for bandwidth sharing [4], [5], [8], [9], [10], multi-resource fair queueing remains a largely uncharted territory. The recent work of Ghodsi *et al.* [11] suggests a promising alternative, known as DRFQ, that implements Dominant Resource Fairness (DRF) [12] in the time domain. While DRFQ provides nearly perfect service isolation, it is expensive to implement. Specifically, DRFQ requires $O(\log n)$ time complexity per packet, where n is the number of backlogged flows. With large n, it is hard to implement DRFQ at high speeds. This problem is aggravated in the recent middlebox innovations, where software-defined middleboxes deployed as VMs and processes are now replacing traditional network appliances with dedicated hardwares [13], [14]. As more software-defined middleboxes are consolidated onto commodity and cloud servers [1], [2], a device will see an increasing amount of flows competing for resources.

In this paper, we design a packet scheduling algorithm, called Multi-Resource Round Robin (MR^3), that takes $O(1)$ time complexity to schedule a packet, and achieves similar fairness performance as DRFQ. While round-robin schemes have found successful applications to fairly share outgoing bandwidth of L2/L3 devices [9], [15], [16], directly applying them to schedule multiple resources may lead to *arbitrary unfairness*. We show, analytically, that simply withholding the scheduling opportunity of a packet until the progress gap between two resources falls below a small threshold leads to nearly perfect fairness. We explore the design space of round-robin algorithms, and implement this idea in a way similar to Elastic Round Robin [16], which we show is the most suitable round-robin variant for the middlebox environment. Both theoretical analyses and extensive simulation show that as compared with DRFQ, the price we pay is a slight increase in packet latency. To our knowledge, this is the first multi-resource fair queueing scheme that offers near-perfect fairness with $O(1)$ time complexity. We believe that our scheme is amenable to an extremely simple implementation, and may find a variety of applications in other multi-resource scheduling contexts, *e.g.*, VM scheduling inside a hypervisor.

II. Related Work

Unlike switches and routers where the output bandwidth is the only shared resource, middleboxes handle a variety of hardware resources and require a more complex packet scheduler. Many recent measurements, such as [6], [11], [17], report that packet processing in a middlebox may bottleneck on any of CPU, memory bandwidth, and link bandwidth, depending on the network functions applied to the traffic flow. Such a multi-resource setting significantly complicates the scheduling algorithm. As pointed out in [11], simply applying traditional fair queueing schemes [4], [5], [8], [9], [18], [19] per resource (*i.e.*, per-resource fairness) or on the bottleneck resource (*i.e.*, bottleneck fairness) fails to offer service isolation: by strategically claiming some resources that are not needed, a flow may increase its service share at the price of other flows.

Ghodsi *et al.* [11] suggest a promising scheduler that implements Dominant Resource Fairness (DRF) in the time domain and therefore achieves service isolation across multiple resources. Their design, referred to as DRFQ, schedules packets in a way such that flows receive roughly the same processing time on their most congested resources. Following this intuition, Wang *et al.* [20] extend the idealized GPS model [4], [5] to Dominant Resource GPS (DRGPS) that implements the strict DRF at all times. By emulating DRGPS, well-known fair queueing algorithms, such as WFQ [4] and WF^{2}Q [21], can have direct extensions in the multi-resource setting. While all these algorithms achieve nearly perfect service isolation, they are *timestamp-based* schedulers and are expensive to implement. In particular, packets, upon their arrivals, are stamped some timestamps and are scheduled in increasing order of their timestamps. To maintain the right scheduling order, the scheduler has to select a packet with the earliest timestamp among n active flows, requiring $O(\log n)$ time complexity per packet. With a large number of flows passing through a middlebox, these algorithms are hard to implement at high speeds.

Such a challenge to reduce the scheduling complexity should come at no surprise to network researchers. When there is only a single resource to schedule, round-robin schedulers [9], [15], [16], [22] have been proposed to multiplex the output bandwidth of switches and routers, in which flows are served in a round-robin fashion. These algorithms eliminate the sorting bottleneck associated with timestamp-based schedulers, and achieve $O(1)$ time complexity per packet. Due to their extreme simplicity, they have been widely implemented in high-speed routers such as Cisco GSR [23].

Despite the successful applications of round-robin algorithms in traditional L2/L3 devices, it remains unclear whether their attractiveness, *i.e.*, the implementation simplicity and low time complexity, extends to multi-resource scheduling, and if it does, how a round-robin scheduler should be designed and implemented in middleboxes. We answer these questions in the following sections.

III. Multi-Resource Round Robin

In this section, we revisit round-robin algorithms in the traditional fair queueing literature and discuss the challenges of extending them to the multi-resource setting. We see that directly applying them to schedule multiple middlebox resources may lead to arbitrary unfairness across flows. Before we inspect this problem in depth, we first introduce some basic concepts that will be used throughout the paper.

A. Preliminaries

Packet Processing Time: Depending on the network functions applied to a flow, processing a packet of the flow may consume different amounts of middlebox resources. Following [11], we define the *packet processing time* as a metric to measure the resource requirements of a packet. Specifically, for packet p, its packet processing time on resource r, denoted $\tau_r(p)$, is defined as the time required to process the packet on resource r, normalized to the middlebox's *processing capacity* of resource r. For example, a packet may require 10 μs to process using one CPU core. A middlebox with 2 CPU cores can process 2 such packets in parallel. As a result, the packet processing time of this packet on CPU is 5 μs.

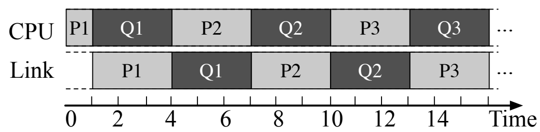
Dominant Resource Fairness (DRF): The recently proposed Dominant Resource Fairness (DRF) [12] serves as a promising notion of fairness for multi-resource scheduling. Informally speaking, with DRF, any two flows receive the same processing time on their *dominant resources* in all backlogged periods. The dominant resource is the one that requires the most packet processing time. Specifically, for a packet p, its dominant resource, denoted $d(p)$, is defined as

$$d(p) = \arg\max_r \{\tau_r(p)\} \ . \tag{1}$$

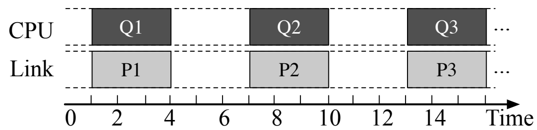
For example, consider two flows in Fig. 1a. Flow 1 sends packets P1, P2, ..., while flow 2 sends packets Q1, Q2, ... Packet P1 requires 1 time unit for CPU processing and 3 time units for link transmission, and has the processing time $\langle 1, 3\rangle$. All the other packets require the same processing time $\langle 3, 3\rangle$ on both CPU and link bandwidth. In this case, the dominant resource of packet P1 is the link bandwidth, while the dominant resource of packets Q1, P2, Q2, P3, ... is CPU (or bandwidth). We see that the scheduling scheme shown in Fig. 1a achieves DRF, under which both flows receive the same processing time on their dominant resources (see Fig. 1b).

It has been shown in [20] that by achieving strict DRF at all times, the resulting scheduling scheme offers the following properties.

Predictable Service Isolation: For each flow i, the received service is at least at the level when every resource is equally allocated.



(a) The scheduling discipline.



(b) The processing time received on the dominant resources.

Fig. 1. Illustration of a scheduling discipline that achieves DRF.

Truthfulness: No flow can receive better service (finish faster) by misreporting the amount of resources it requires.

Work Conservation: No resource that could be used to serve a backlogged flow is wasted in idle.

Due to these highly desired scheduling properties, DRF is adopted as the notion of fairness for multi-resource scheduling. To measure how well a packet scheduler approximates DRF, the following Relative Fairness Bound (RFB) is used as a fairness metric [11], [20]:

Definition 1: For any packet arrivals, let $T_i(t_1, t_2)$ be the packet processing time flow i receives on its dominant resource in the time interval (t_1, t_2). $T_i(t_1, t_2)$ is referred to as the *dominant service* flow i receives in (t_1, t_2). Let $\mathcal{B}(t_1, t_2)$ be the set of flows that are backlogged in (t_1, t_2). The *Relative Fairness Bound* (RFB) is defined as

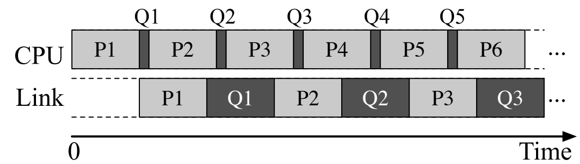
$$\mathrm{RFB} = \sup_{t_1, t_2; i, j \in \mathcal{B}(t_1, t_2)} |T_i(t_1, t_2) - T_j(t_1, t_2)| \ . \tag{2}$$

RFB generalizes the fairness measure of Golestani [8] to the multi-resource setting. We require a scheduling scheme to have a small RFB, such that the difference between the normalized dominant service received by any two flows i and j, over any backlogged time period (t_1, t_2), is bounded by a small constant.

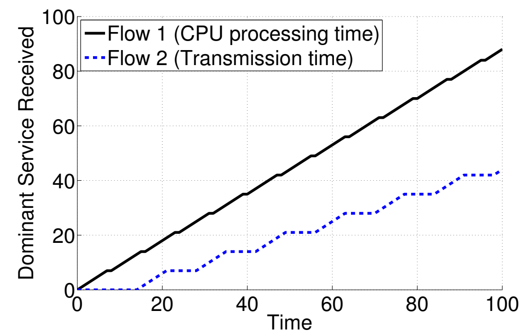
B. Challenges of Round-Robin Extension

As mentioned in Sec. II, among various scheduling schemes, round-robin algorithm is of particular attractiveness for practical implementation due to its extreme simplicity and constant time complexity. To extend it to the multi-resource setting with DRF, a natural way is to directly apply it on flows' dominant resources, such that in each round, flows receive roughly the same dominant services. Such a general extension can be applied to many well-known round-robin algorithms. However, a naive extension may lead to arbitrary unfairness.

Take the well-known Deficit Round Robin (DRR) [9] as an example. When there is a single resource, DRR assigns some predefined quantum size to each flow. Each flow maintains a *deficit counter*, whose value is the current unused transmission quota. In each round, DRR polls every backlogged flow and transmits its packets up to an amount of data equal to the sum of its quantum and deficit counter. The unused transmission quota will be carried over to the next round as the value of the flow's deficit counter. Similar to the single-resource case, one can apply DRR [9] on flows' dominant resources as follows.



(a) Direct application of DRR to schedule multiple resources.



(b) The dominant services received by two flows.

Fig. 2. Illustration of a direct DRR extension. Each packet of flow 1 has processing time $\langle 7, 6.9 \rangle$, while each packet of flow 2 has processing time $\langle 1, 7 \rangle$.

Initially, the algorithm assigns a predefined quantum size to each flow, which is also the amount of dominant service the flow is allowed to receive in one round. Each flow maintains a deficit counter that measures the current unused portion of the allocated dominant service. Packets are scheduled in rounds, and in each round, each backlogged flow schedules as many packets as it has, as long as the dominant service consumed does not exceed the sum of its quantum and deficit counter. The unused portion of this amount is carried over to the next round as the new value of the deficit counter.

As an example, consider two flows where flow 1 sends P1, P2, ..., while flow 2 sends Q1, Q2, ... Each packet of flow 1 has processing time $\langle 7, 6.9 \rangle$, *i.e.*, it requires 7 time units for CPU processing and 6.9 time units for link transmission. Each packet of flow 2 requires processing time $\langle 1, 7 \rangle$. Fig. 2a illustrates the resulting schedule of the above naive DRR extension, where the quantum size assigned to both flows is 7. In round 1, both flows receive a quantum of 7, and can process 1 packet each, which consumes up the quantum awarded on the dominant resources in this round. Such a process repeats in the following rounds. As a result, packets of the two flows are scheduled alternately. Since in each round, the received quantum is always used up, the deficit counter remains 0 in the end of each round.

Similar to single-resource DRR, the extension above has $O(1)$ time complexity per packet[1]. However, such an extension fails to provide fair services, with respect to DRF. Instead, it may lead to arbitrary unfairness with unbounded RFB! To see this, we depict the dominant services received by two flows

[1]The $O(1)$ time complexity is conditioned on the quantum size being at least the maximum packet processing time.

in Fig. 2b. We see that flow 1 receives nearly two times the dominant service flow 2 receives, *i.e.*, $T_1(0,t) \approx 2T_2(0,t)$. With more packets being scheduled, the service gap increases, eventually leading to unbounded RFB.

It is to be emphasized that the problem of arbitrary unfairness is not limited to DRR extension only, but generally extends to all round-robin variants. For example, one can extend Surplus Round Robin (SRR) [15] and Elastic Round Robin (ERR) [16] to the multi-resource setting in a similar way (more details will be given in Sec. IV). It is easy to verify that running the example above will give exactly the same schedule shown in Fig. 2a with unbounded RFB[2]. In fact, due to the heterogeneous resource requirements among flows, a service round may span different time intervals on different resources. As a result, the work progress on one resource may be far ahead of that on the other. For example, in Fig. 2a, when CPU starts to process packet P6, the transmission of packet P3 remains unfinished. It is such a progress mismatch that leads to a significant gap between the two flows' dominant services.

In summary, directly applying round-robin algorithms on flows' dominant resources fails to provide fair services. A new design is therefore required. We preview the basic idea in the next subsection.

C. Deferring the Scheduling Opportunity

The key reason that direct round-robin extensions fails is because they cannot track flows' dominant services in real-time. Take the DRR extension as an example. In Fig. 2a, after packet Q1 is completely processed on CPU, flow 2's deficit counter is updated to 0, meaning that flow 2 has already used up the quantum allocated for dominant services (*i.e.*, link transmission) in round 1. This allows P2 to be processed but erroneously, as the actual consumption of this quantum incurs only when Q1 is transmitted on the link, after the transmission of packet P1.

To circumvent this problem, a simple fix is to withhold the scheduling opportunity of *every packet* until its previous packet is completely processed on *all resources*, which allows the scheduler to track the dominant services accurately. Fig. 3 depicts the resulting schedule when applying this fix to the DRR extension shown in Fig. 2a. We see that the difference between the dominant services received by two flows is bounded by a small constant. However, such a fairness improvement is achieved at the expense of significantly lower resource utilization. Even though multiple packets can be processed in parallel on different resources, the scheduler serves only one packet at a time, leading to poor resource utilization and high packet latency. As a result, this simple fix cannot meet the demand of high-speed networks.

To strike a balance between fairness and latency, packets should not be deferred as long as the difference of two flows' dominant services is small. This can be achieved by bounding the progress gap on different resources by a small

[2]In either SRR or ERR extension, by scheduling 1 packet, each flow uses up all the quantum awarded in each round. As a result, packets of the two flows are scheduled alternately, the same as that in Fig. 2a.

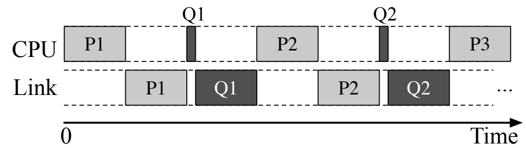
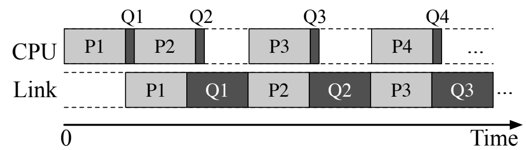
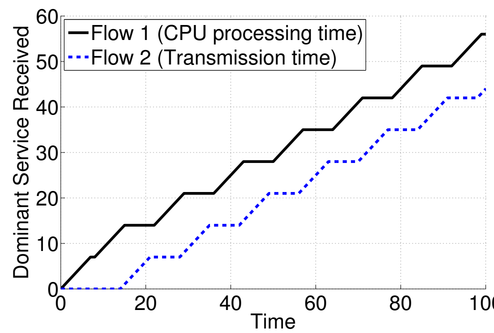


Fig. 3. Naive fix of the DRR extension shown in Fig. 2a by withholding the scheduling opportunity of every packet until its previous packet is completely processed on all resources.



(a) Schedule by MR^3.



(b) The dominant services received by two flows.

Fig. 4. Illustration of a schedule by MR^3.

amount. In particular, we may serve flows in rounds as follows. Whenever a packet p of a flow i is ready to be processed on the first resource (usually CPU) in round k, the scheduler checks the work progress on the last resource (usually the link bandwidth). If flow i has already received services on the last resource in the previous round $k-1$, or it is a new arrival, then packet p is scheduled immediately. Otherwise, packet p is withheld until flow i starts to receive service on the last resource in round $k-1$. As an example, Fig. 4a depicts the resulting schedule with the same input traffic as that in the previous example of Fig. 2a. In round 1, both packets P1 and Q1 are scheduled without delay because both flows are new arrivals. In round 2, packet P2 (resp., Q2) is also scheduled without delay, because when it is ready to be processed, flow 1 (resp., flow 2) has already started its service on the link bandwidth in round 1. In round 3, while packet P3 is ready to be processed right after packet Q2 is completely processed on CPU, it has to wait until packet P2 starts to be transmitted, as it has to wait until flow 1 receives service on the link bandwidth in round 2. Similar process repeats for all the subsequent packets.

We will show later in Sec. V that such a simple idea leads to nearly perfect fairness across flows, without incurring high packet latency. In fact, the schedule in Fig. 4a *incurs the same packet latency* as that in Fig. 2a, but is much more fair. As we see from Fig. 4b, the difference between dominant services received by two flows is bounded by a small constant.

IV. MR^3 Design

While the general idea introduced in the previous section is simple, implementing it as a concrete round-robin algorithm is nontrivial. We next explore the algorithm design space and implement the idea in a way similar to Elastic Round Robin [16], which we show is the most suitable round-robin variants for middleboxes. The resulting algorithm is referred to as Multi-Resource Round Robin (MR^3).

A. Design Space of Round-Robin Algorithms

There are many round-robin variants in the traditional fair queueing literature. While all these variants achieve similar performance and are all feasible for the single-resource scenario, not all of them are suitable to implement our idea in a middlebox. We investigate three typical variants, *i.e.*, Deficit Round Robin (DRR) [9], Surplus Round robin (SRR) [15], and Elastic Round Robin (ERR) [16], and discuss their implementation issues in middleboxes as follows.

Deficit Round Robin (DRR): We have introduced the basic idea of DRR in Sec. III-B. As an analogy, one can view the behavior of each flow as maintaining a banking account. In each round, a predefined quantum is deposited into a flow's account, tracked by the deficit counter. The balance of the account (*i.e.*, the value of the deficit counter) represents the dominant service the flow is allowed to receive in the current round. Scheduling a packet is analogous to withdrawing the corresponding packet processing time on the dominant resource from the account. As long as there is sufficient balance to withdraw from the account, a packet is allowed to process.

However, DRR is not amenable to implement in middleboxes due to the following two reasons. First, to ensure that a flow has sufficient account balance to schedule a packet, the processing time required on the dominant resource has to be known before packet processing. However, it is hard to know what middlebox resources are needed and how much processing time is required until the packet is processed. Moreover, the $O(1)$ time complexity of DRR is conditioned on the quantum size that is at least the same as the maximum packet processing time, which is hard to estimate in a real system. Without satisfying this condition, the time complexity could be as high as $O(N)$ [16].

Surplus Round Robin (SRR): SRR [15] allows a flow to consume more processing time on its dominant resource in one round than it has in its account. As a compensation, the excessive consumption, tracked by a *surplus counter*, will be deducted from the quantum awarded in the future rounds. In SRR, as long as the account balance (*i.e.*, surplus counter) is positive, the flow is allowed to schedule packets, and the corresponding packet processing time is withdrawn from the account after the packet finishes processing on its dominant resource. In this case, the packet processing time is only needed *after* the packet has been processed.

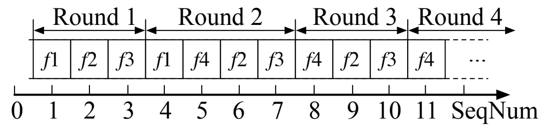


Fig. 5. Illustration of the round-robin service and the sequence number.

While SRR does not require knowing packet processing time beforehand, its $O(1)$ time complexity remains conditioned on the predefined quantum size that is at least the same as the maximum packet processing time. Otherwise, the time complexity could be as high as $O(N)$ [16]. For the same reason, SRR is not amenable to implement in middleboxes either.

Elastic Round Robin (ERR): Similar to SRR, ERR [16] does not require knowing the processing time before the packet is processed. It allows flows to overdraw its permitted processing time in one round on the dominant resource, with the excessive consumption deducted from the quantum received in the next round. The difference is that instead of depositing a predefined quantum with fixed size, in ERR, the quantum size in one round is *dynamically* set as the *maximum* excessive consumption incurred in the previous round. This ensures that each flow will *always have a positive balance* in its account at the beginning of each round, and can schedule *at least one packet*. In this case, ERR achieves $O(1)$ time complexity without knowing the maximum packet processing time *a priori*, and is the most suitable to implement in middleboxes at high speeds.

B. MR^3 Design

While ERR serves as a promising round-robin variant to extend for middleboxes, there remain several challenges to implement the idea presented in Sec. III-C. How can the scheduler quickly track the work progress gap of two resources and decide when to withhold a packet? To ensure efficiency, such a progress comparison must be completed within $O(1)$ time. Note that simply comparing the numbers of packets that have been processed on two resources does not give any clue about the progress gap: due to traffic dynamics, each round may consist of different amounts of packets.

To circumvent this problem, we associate each flow i a *sequence number* $SeqNum_i$, which increases from 0 and is the scheduling order of the flow. We use a global variable *NextSeqNum* to record the next sequence number that will be assigned to a flow. The value of *NextSeqNum* is initialized to 0 and increases by 1 every time a flow is processed. Each flow i also records its sequence number in the previous round, tracking by $PreviousRoundSeqNum_i$. For example, consider Fig. 5. Initially, flows 1, 2 and 3 are backlogged and are served in sequence in round 1, with sequence numbers 1, 2 and 3, respectively. Later, while flow 2 is being served, flow 4 becomes active. Flow 4 is therefore scheduled right after flow 1 in round 2, with a sequence number 5. After round 2, flow 1 has no new packet to serve and becomes inactive. As a result, only flows 2, 3 and 4 are serviced in round 3, where their sequence numbers in the previous round are 6, 7 and 5, respectively.

We use sequence numbers to track the work progress on a resource. Whenever a packet p is scheduled to be processed, it is stamped a service tag (*i.e.*, p.*Tag*) whose value is its flow's sequence number. By checking the service tag of the packet that is being processed on a resource, the scheduler knows exactly the work progress on that resource.

Besides sequence number, the following important variables are also used in the algorithm.

Active list: The algorithm maintains an *ActiveFlowList* to track backlogged flows. Flows are served in a round-robin fashion. The algorithm always serves the flow at the head of the list, and after the service, this flow, if remaining active, will be moved to the tail of the list for service in the next round. Newly arrived flows is always appended to the tail of the list, and will be served in the next round. We also use *RoundRobinCounter* to track the number of flows that have not yet been served in the current round. Initially, *ActiveFlowList* is empty and *RoundRobinCounter* is 0.

Excess counter: Each flow i maintains an *excess counter* EC_i, recording the excessive dominant service flow i incurred in one round. The algorithm also uses two variables, *MaxEC* and *PreviousRoundMaxEC*, to track the maximum excessive consumption incurred in the current and the previous round, respectively. Initially, all these variables are set to 0.

Our algorithm, referred to as MR^3, consists of 2 functional modules, *PacketArrival* (Module 1), which handles packet arrival events, and *Scheduler* (Module 2), which decides which packet should be processed next.

PacketArrival: This module is invoked upon a packet arrival. It enqueues the packet to the input queue of the flow to which the packet belongs. If this flow is previously inactive, it is then appended to the tail of the active list and will be serviced in the next round. The sequence number of the flow is also updated, as shown in Module 1 (line 3 to line 5).

Module 1 MR^3 PacketArrival

1: Let i be the flow to which the packet belongs
2: **if** *ActiveFlowList*.Contains(i) == FALSE **then**
3:     *PreviousRoundSeqNum*$_i$ = *SeqNum*$_i$
4:     *NextSeqNum* = *NextSeqNum* + 1
5:     *SeqNum*$_i$ = *NextSeqNum*
6:     *ActiveFlowList*.AppendToTail(i)
7: **end if**
8: Enqueue the packet to queue i

Scheduler: This module decides which packet should be processed next. The scheduler first checks the value of *RoundRobinCounter* to see how many flows have not yet been served in the current round. If the value is 0, then a new round starts. The scheduler sets *RoundRobinCounter* to the length of the active list (line 3), and updates *PreviousRoundMaxEC* as the maximum excessive consumption incurred in the round that has just passed (line 4), while *MaxEC* is reset to 0 for the new round (line 5).

Module 2 MR^3 Scheduler

1: **while** TRUE **do**
2:     **if** *RoundRobinCounter* == 0 **then**
3:         *RoundRobinCounter* = *ActiveFlowList*.Length()
4:         *PreviousRoundMaxEC* = *MaxEC*
5:         *MaxEC* = 0
6:     **end if**
7:     Flow i = *ActiveFlowList*.RemoveFromHead()
8:     B_i = *PreviousRoundMaxEC* $-$ *EC*$_i$
9:     **while** $B_i \geq 0$ **and** QueueIsNotEmpty(i) **do**
10:         Let q be the packet being processed on the last resource
11:         WaitUntil(q.*Tag* $\geq$ *PreviousRoundSeqNum*$_i$)
12:         Packet p = Dequeue(i)
13:         p.*Tag* = *SeqNum*$_i$
14:         ProcessPacket(p)
15:         B_i = B_i $-$ DominantProcessingTime(p)
16:     **end while**
17:     **if** QueueIsNotEmpty(i) **then**
18:         *ActiveFlowList*.AppendToTail(i)
19:         *NextSeqNum* = *NextSeqNum* + 1
20:         *PreviousRoundSeqNum*$_i$ = *SeqNum*$_i$
21:         *SeqNum*$_i$ = *NextSeqNum*
22:         *EC*$_i$ = $-B_i$
23:     **else**
24:         *EC*$_i$ = *0*
25:     **end if**
26:     *MaxEC* = Max(*MaxEC*, *EC*$_i$)
27:     *RoundRobinCounter* = *RoundRobinCounter* $-$ 1
28: **end while**

The scheduler then serves the flow at the head of the active list. Let flow i be such a flow. Flow i receives a quantum equal to the maximum excessive consumption incurred in the previous round, and has its account balance B_i equal to the difference between the quantum and the excess counter, *i.e.*, $B_i =$ *PreviousRoundMaxEC* $-$ *EC*$_i$. Since *PreviousRoundMaxEC* $\geq$ *EC*$_i$, we have $B_i \geq 0$.

Flow i is allowed to schedule packets (if any) as long as its balance is positive (nonnegative). To ensure a small work progress gap between two resources, the scheduler keeps checking the service tag of the packet that is being processed on the last resource[3] (*i.e.*, output bandwidth) and compares it with flow i's sequence number in the previous round. The scheduler waits until the former exceeds the latter, at which time the progress gap between any two resources is within 1 round. The scheduler then dequeues a packet from the input queue of flow i, stamps a service tag equal to flow i's sequence number, and performs deep packet processing on CPU, which is also the first middlebox resource required by the packet. After CPU processing, the scheduler knows exactly how the packet should be processed next and what resources are required. The packet processing time on each resource can now be accurately estimated, for example, via some simple packet profiling technique introduced in [11]. The scheduler then deducts the dominant processing time of the packet from flow i's balance. The service for flow i continues until flow i has no packet to process or its balance becomes negative.

[3]If no packet is being processed, we take the service tag of the packet that has recently been served.

If flow i is no longer active after service in the current round, its excess counter will be reset to 0. Otherwise, flow i is appended to the tail of the active list for service in the next round. In this case, a new sequence number is associated with flow i. The excess counter EC_i is also updated as the account deficit of flow i. Finally, before serving the next flow, the scheduler updates *MaxEC* and decrements *RoundRobinCounter* by 1, indicating that one flow has already finished service in the current round.

V. Analytical Results

In this section, we analyze the performance of MR^3 by deriving its time complexity, fairness, and delay bound.

A. Complexity and Fairness

MR^3 is highly efficient as compared with DRFQ [11]. One can verify that under MR^3, at least one packet is scheduled for each flow in one round. Formally, we have

Theorem 1: The time complexity of MR^3 is $O(1)$ per packet.

Proof: We prove the theorem by showing that both enqueuing a packet (Module 1) and scheduling a packet (Module 2) finish within $O(1)$ time.

In Module 1, determining the flow at which a new packet arrives is an $O(1)$ operation. By maintaining the per-flow state, the scheduler knows if the flow is contained in the active list (line 2) within $O(1)$ time. Also, updating the sequence number (line 3 to 5), appending the flow to the active list (line 6), and enqueueing a packet (line 8) are all of $O(1)$ time complexity.

We now analyze the time complexity of scheduling a packet. In Module 2, since the quantum deposited into each flow's account is the maximum excessive consumption incurred in the previous round, a flow will always have a positive balance at the beginning of each round, *i.e.*, $B_i \geq 0$ in line 15. As a result, at least one packet is scheduled for each flow in one round. The time complexity of scheduling a packet is therefore no more than the time complexity of all the operations performed during each service opportunity. These operations include determining the next flow to be served, removing the flow from the head of the active list and possibly adding it back at the tail, all of which are $O(1)$ operations if the active list is implemented as a linked list. Additional operations include updating the sequence number, *MaxEC*, *PreviousRoundMaxEC*, *RoundRobinCounter*, and dequeuing a packet. All of them are also executed within $O(1)$ time. ■

Despite such low time complexity, MR^3 achieves similar fairness performance as DRFQ. To see this, let EC_i^k be the excess counter of flow i after round k, and $MaxEC^k$ the maximum EC_i^k over all flow i's. Let D_i^k be the dominant service flow i receives in round k. Also, let L_i be the maximum packet processing time of flow i across all resources. Finally, let L be the maximum packet processing time across all flows, *i.e.*, $L = \max_i\{L_i\}$. We can show that the following lemmas and corollaries hold throughout the execution of MR^3 algorithm.

Lemma 1: $EC_i^k \leq L_i$ for all flow i and round k.

Proof: If flow i has no packets to serve (*i.e.,* the input queue is empty) after round k, then $EC_i^k = 0$ (line 24 in Module 2) and the statement holds. Otherwise, let packet p be the last packet of flow i served in round k. Let B_i' be the account balance of flow i before packet p is served. We have

$$EC_i^k = \mathit{DominantProcessingTime}(p) - B_i' \leq L_i, \quad (3)$$

where the inequality holds because $B_i' \geq 0$ and $\mathit{DominantProcessingTime}(p) \leq L_i$. ■

Corollary 1: $MaxEC^k \leq L$ for all round k.

Lemma 2: For all flow i and round k, we have

$$D_i^k = MaxEC^{k-1} - EC_i^{k-1} + EC_i^k, \quad (4)$$

where $EC_i^0 = 0$ and $MaxEC^0 = 0$.

Proof: At the beginning of round k, flow i has an account balance $B_i = MaxEC^{k-1} - EC_i^{k-1}$. After round k, all this amount of normalized processing time has been consumed on its dominant resource, with an excessive consumption EC_i^k. The normalized dominant service flow i received in round k is therefore $D_i = B_i + EC_i^k =$ RHS of (4). ■

Corollary 2: $D_i^k \leq 2L$ for all flow i and round k.

Proof: By Lemma 2, we derive as follows

$$\begin{aligned} D_i^k &= MaxEC^{k-1} - EC_i^{k-1} + EC_i^k, \\ &\leq L - 0 + L = 2L, \end{aligned} \quad (5)$$

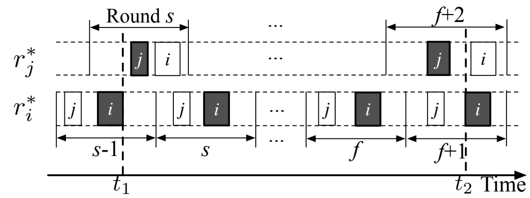
where the inequality holds because of Corollary 1 and Lemma 1. ■

With lemmas and corollaries above, we now analyze the fairness performance of MR^3. For simplicity, we assume flows are *dominant-resource monotonic*, *i.e.*, the flow's dominant resource does not change during any of its backlogged periods, which is usually the case in middleboxes as observed in [11]. The following theorem bounds the difference of dominant services received by two flows that are dominant-resource monotonic. Similar analysis also extends to general flows.

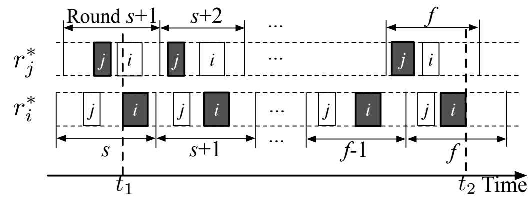
Theorem 2: For any packet arrivals, let $T_i(t_1, t_2)$ be the dominant service flow i received in the interval (t_1, t_2) under MR^3. The following relationship holds for any two dominant-resource monotonic flows that are backlogged in (t_1, t_2):

$$|T_i(t_1, t_2) - T_j(t_1, t_2)| \leq L_i + L_j + 2L. \quad (6)$$

Proof: Let r_i^* (resp. r_j^*) be the dominant resource of flow i (resp. j). Without loss of generality, we assume $r_j^* \leq r_i^*$, that is, a packet is processed on resource r_j^* before it is processed on resource r_i^*. Suppose during (t_1, t_2), flow i receives its dominant service from round s to round f. For $s \leq k \leq f$, let S_i^k be the time from which flow i begins to receive dominant service in round k, and F_i^k the time when flow i finishes its the dominant service in round k. The difference $|T_i(t_1, t_2) - T_j(t_1, t_2)|$ reaches its maximal value when $(t_1, t_2) = (F_i^{s-1}, S_i^{f+1})$ or $(t_1, t_2) = (S_i^s, F_i^f)$. In either



(a) Within (t_1, t_2), flow j receives dominant services at most in rounds $s, \ldots, f+2$.



(b) Within (t_1, t_2), flow j receives dominant services at least in rounds $s+2, \ldots, f$.

Fig. 6. The difference of dominant services received by two flows i and j within (t_1, t_2), where flow j is served before flow i in one round. The dominant services are highlighted as shaded areas.

case, we have

$$T_i(t_1,t_2) = \sum_{k=s}^{f} D_i^k = \sum_{k=s}^{f} \mathit{MaxEC}^{k-1} - \mathit{EC}_i^{s-1} + \mathit{EC}_i^{f} \ , \qquad (7)$$

where the second equality is derived from Lemma 2.

Since both flows i and j are backlogged in (t_1, t_2), flow i is served either before j or after j in all rounds in (t_1, t_2). We hence consider the following two cases.

Case 1: Flow j is served *before* flow i in all rounds in (t_1, t_2). Since under MR^3, the work progress on resource r_j^* is never ahead of that on resource r_i^* by more than 1 round, it is easy to check that flow j receives dominant services *at most* in rounds $s, \ldots, f+2$ (see Fig. 6a), *i.e.*,

$$T_j(t_1,t_2) \le \sum_{k=s}^{f+2} D_j^k = \sum_{k=s}^{f+2} \mathit{MaxEC}^{k-1} - \mathit{EC}_j^{s-1} + \mathit{EC}_j^{f+2} \ . \qquad (8)$$

For the same reason, flow j receives dominant services *at least* in rounds $s+2, \ldots, f$ (see Fig. 6b), *i.e.*,

$$T_j(t_1,t_2) \ge \sum_{k=s+2}^{f} D_j^k = \sum_{k=s+2}^{f} \mathit{MaxEC}^{k-1} - \mathit{EC}_j^{s+1} + \mathit{EC}_j^{f} \ . \qquad (9)$$

Now let the RHS of (8) be $\alpha_j(t_1, t_2)$, and let the RHS of (9) be $\beta_j(t_1, t_2)$. Further, let

$$\Delta_1(t_1,t_2) = |T_i(t_1,t_2) - \alpha_j(t_1,t_2)| \qquad (10)$$

and

$$\Delta_2(t_1,t_2) = |T_i(t_1,t_2) - \beta_j(t_1,t_2)| \ . \qquad (11)$$

We have

$$\begin{aligned}\Delta_1(t_1,t_2) &= \Bigg| \sum_{k=f+1}^{f+2} \mathit{MaxEC}^{k-1} - \mathit{EC}_j^{s-1} + \mathit{EC}_j^{f+2} \\ &\qquad + \mathit{EC}_i^{s-1} - \mathit{EC}_i^{f} \Bigg| \\ &\le 2L + L_i + L_j \ , \end{aligned} \qquad (12)$$

where the inequality is derived from Lemma 1 and Corollary 1. Similarly, we have

$$\Delta_2(t_1,t_2) \le 2L + L_i + L_j \ . \qquad (13)$$

Finally, we see that the statement holds because

$$\begin{aligned}|T_i(t_1,t_2) - T_j(t_1,t_2)| &\le \max\{\Delta_1(t_1,t_2), \Delta_2(t_1,t_2)\} \\ &\le 2L + L_i + L_j \ . \end{aligned} \qquad (14)$$

Case 2: Flow j is served *after* flow i in all rounds in (t_1, t_2). It is easy to check that flow j receives dominant services *at most* in rounds $s-1, \ldots, f+1$ (see Fig. 7a), *i.e.*,

$$T_j(t_1,t_2) \le \sum_{k=s-1}^{f+1} D_j^k = \sum_{k=s-1}^{f+1} \mathit{MaxEC}^{k-1} - \mathit{EC}_j^{s-2} + \mathit{EC}_j^{f+1} \ . \qquad (15)$$

For the same reason, flow j receives dominant services *at least* in rounds $s+1, \ldots, f-1$ (see Fig. 7b), *i.e.*,

$$T_j(t_1,t_2) \ge \sum_{k=s+1}^{f-1} D_j^k = \sum_{k=s+1}^{f-1} \mathit{MaxEC}^{k-1} - \mathit{EC}_j^{s} + \mathit{EC}_j^{f-1} \ . \qquad (16)$$

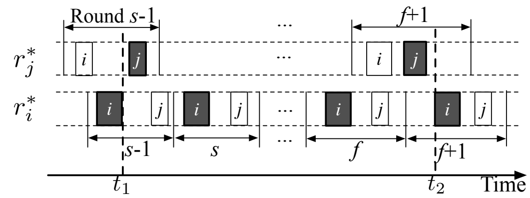
By (7), (15), (16) and deriving similarly as Case 1, we see that the statement holds. ■

Corollary 3: MR^3 has $\mathrm{RFB} = 4L$.

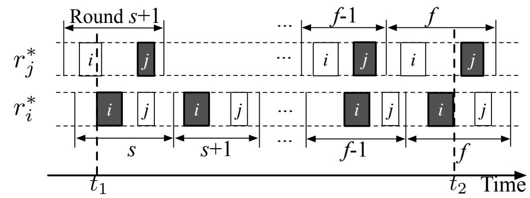
Based on Theorem 2 and Corollary 3, we see that MR^3 bounds the difference between dominant services received by two backlogged flows in *any time interval* by a small constant. Note that the interval (t_1, t_2) may be arbitrarily large. MR^3 therefore achieves nearly perfect DRF across all active flows.

B. Latency

In addition to complexity and fairness, latency is also an important concern for a packet scheduling algorithm. Two metrics are widely used in the fair queueing literature to measure the latency performance: *startup latency* [11], [16] and *single packet delay* [22]. The former measures how long it takes for a previously inactive flow to receive service after it becomes active, while the latter measures the latency from the time when a packet reaches the head of the input queue to the time when this packet finishes service on all resources.



(a) Within (t_1, t_2), flow j receives dominant services at most in rounds $s-1, \ldots, f+1$.



(b) Within (t_1, t_2), flow j receives dominant services at least in rounds $s+1, \ldots, f-1$.

Fig. 7. The difference of dominant services received by two flows i and j within (t_1, t_2), where flow i is served before j in one round. The dominant services are highlighted as shaded areas.

TABLE I
PERFORMANCE COMPARISON BETWEEN MR^3 AND DRFQ, WHERE L IS THE MAXIMUM PACKET PROCESSING TIME; m IS THE NUMBER OF RESOURCES; AND n IS THE NUMBER OF BACKLOGGED FLOWS.

Performance	**MR^3**	**DRFQ** [11]
Complexity	$O(1)$	$O(\log n)$
Fairness (RFB)	$4L$	$2L$
Startup Latency	$2(m+n-1)L$	nL
Single Packet Delay	$(4m+4n-2)L$	Unknown

Our analysis begins with the startup latency. Let m be the number of resources concerned, and n the number of backlogged flows. We have the following theorem. The proof is given in Appendix A.

Theorem 3: Under MR^3, for any newly backlogged flow i, the startup latency SL_i is bounded by

$$\mathrm{SL}_i \leq 2(m+n-1)L \ . \tag{17}$$

We next state the following theorem on the single packet delay. The proof is given in Appendix B.

Theorem 4: Under MR^3, for any packet p, the single packet delay $\mathrm{SPD}(p)$ is bounded by

$$\mathrm{SPD}(p) \leq (4m+4n-2)L \ . \tag{18}$$

Table I summarizes the derived performance of MR^3, as compared with those of DRFQ [11]. We see that MR^3 significantly reduces the time complexity per packet. Similar to DRFQ, MR^3 also achieves nearly perfect fairness across flows. The price we paid, however, is longer startup latency for newly active flows. Since the number of middlebox resources is typically much smaller than the number of active flows, *i.e.,* $m \ll n$, the startup latency bound of MR^3 is two times that of DRFQ, *i.e.*, $2(m+n-1)L \approx 2nL$. Since single packet delay is usually hard to analyze, no analytical delay bound is given in [11]. We experimentally compare the latency performance of MR^3 and DRFQ in the next section.

TABLE II
LINEAR MODEL FOR CPU PROCESSING TIME IN 3 MIDDLEBOX MODULES. MODEL PARAMETERS ARE BASED ON THE MEASUREMENT RESULTS REPORTED IN [11].

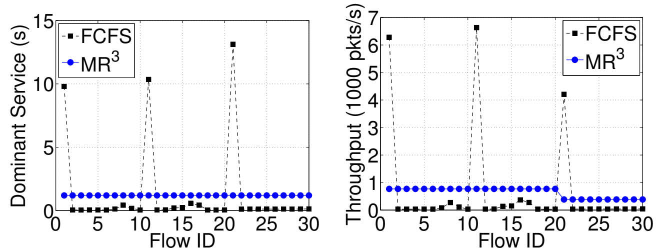
Module	**CPU processing time** (μs)
Basic Forwarding	$0.00286 \times$ PacketSizeInBytes $+ 6.2$
Statistical Monitoring	$0.0008 \times$ PacketSizeInBytes $+ 12.1$
IPSec Encryption	$0.015 \times$ PacketSizeInBytes $+ 84.5$

VI. SIMULATION RESULTS

As a complementary study of theoretical analysis, we evaluate the performance of MR^3 via extensive simulations. In particular, (1) we would like to confirm experimentally that MR^3 offers predictable service isolation and is superior to the naive *first-come-first-served* (FCFS) scheduler, as the theory indicates. (2) We want to confirm that MR^3 can quickly adapt to traffic dynamics and achieve nearly perfect DRF across flows. (3) We compare the latency performance of MR^3 with DRFQ [11] to see if the extremely low time complexity of MR^3 is achieved at the expense of significant packet delay. (4) We also investigate how sensitive the performance of MR^3 is when packet size distributions and arrival patterns change.

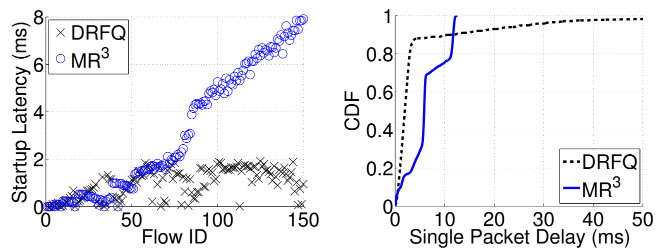
General Setup: All simulation results are based on our event-driven packet simulator written with 3,000 lines of C++ codes. We assume resources are consumed serially, with CPU processing first, followed by link transmission. We implement 3 schedulers, FCFS, DRFQ and MR^3. The last two inspect the flows' input queues and decide which packet should be processed next, based on their algorithms. By default, packets follow Poisson arrivals. The simulator simulates resource consumption of packet processing in 3 typical middlebox modules, each corresponds to one type of flows, basic forwarding, per-flow statistical monitoring, and IPSec encryption. The first two modules are bandwidth-bound, with statistical monitoring consuming slightly more CPU resources than basic forwarding, while IPSec is CPU intensive. For direct comparison, we set the packet processing times required for each middlebox module the same as those in [11], which are based on real measurements. In particular, the CPU processing time of each module is observed to follow a simple linear model based on packet size x, *i.e.*, $\alpha_k x + \beta_k$, where α_k and β_k are linear parameters of module k. Table II summarizes the detailed parameters based on the measurement results reported in [11]. The link transmission time is proportional to the packet size, and the output bandwidth of the middlebox is set to 200 Mbps.

Service Isolation: We start off by confirming that MR^3 offers nearly perfect service isolation, which naive FCFS fails to provide. We initiate 30 flows that send 1300-byte UDP packets for 30 seconds. Flows 1 to 10 undergo basic forwarding; 11 to 20 undergo statistical monitoring; 21 to 30 undergo IPSec encryption. We generate 3 rogue flows, *i.e.*, 1, 11 and 21, each sending 10,000 pkts/s. All other flows behaves normally, each sending 1,000 pkts/s. Fig. 8a shows the dominant services received by different flows under FCFS and MR^3. We see that under FCFS, rogue flows grab an arbitrary share of middlebox resources, while under MR^3, flows receive fair services on their dominant resources. This result is further



(a) Dominant service received. (b) Packet throughput of flows.

Fig. 8. Dominant services and packet throughput received by different flows under FCFS and MR^3. Flows 1, 11 and 21 are ill-behaving.



(a) Startup latency of flows. (b) CDF of the single packet delay.

Fig. 9. Latency comparison between DRFQ and MR^3.

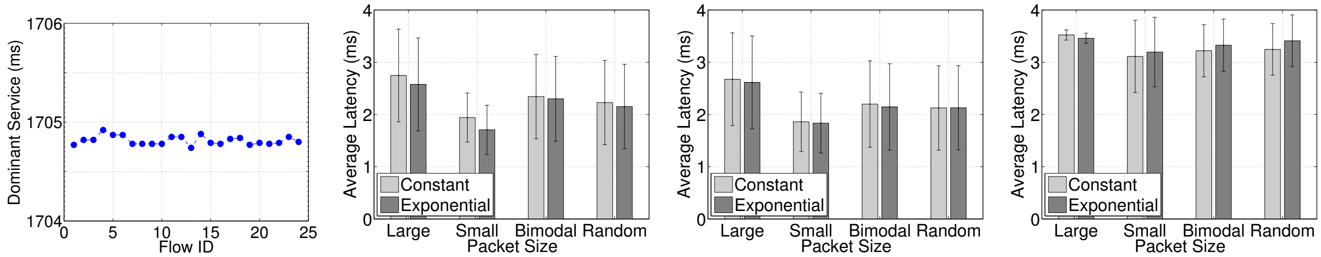
confirmed in Fig. 8b: Under FCFS, the presence of rogue flows squeezes normal traffics to almost zero. In contrast, MR^3 ensures that all flows receive deserved, though uneven, throughput based on their dominant resource requirements, irrespective of the presence and (mis)behaviour of other traffic.

Latency: We next evaluate the latency price MR^3 pays for its extremely low time complexity, as compared with DRFQ [11]. We implement DRFQ and measure the startup latency as well as the single packet delay of both algorithms. In particular, 150 UDP flows start generating traffic in serial, where flow 1 is active at time 0, followed by flow 2 at time 0.2, and flow 3 at time 0.3, and so on. A flow randomly chooses one of the three middlebox modules to pass through. To congest the middlebox resources, the packet arrival rate of each flow is set to 500 pkts/s, and the packet size is uniformly drawn from 200 B to 1300 B. Fig. 9a depicts the per-flow startup latency using both DRFQ and MR^3. Clearly, the dense and sequential flow starting times in this example represent a *worst-case scenario* for a round-robin scheduler. We see that under MR^3, flows joining the system later see larger startup latency, while under DRFQ, the startup latency is relatively consistent. This is because under MR^3, a newly active flow will have to wait for a whole round before getting served. The more active flows, the more time is required to finish serving one round. As a result, the startup latency is linearly dependent on the number of active flows. While this is also true for DRFQ in the worst-case analysis (see Table I), our simulation results show that on average, the startup latency of DRFQ is smaller than MR^3. However, we see next that this advantage of DRFQ comes at the expense of highly uneven single packet delays.

Compared with the startup latency, single packet delay is a much more important delay metric. As we see from Fig. 9b, MR^3 exhibits more consistent packet delay performance, with all packets delayed less than 15 ms. In contrast, the latency distribution of DRFQ is observed to have a long tail: 90% packets are delayed less than 5 ms while the rest 10% are delayed from 5 ms to 50 ms. Further investigation reveals that these 10% packets are uniformly distributed among all flows. All results above indicate that the low time complexity and near-perfect fairness of MR^3 is achieved at the expense of only slight increase in packet latency.

Dynamic Allocation: We further investigate if the DRF allocation achieved by MR^3 can quickly adapt to traffic dynamics. To congest middlebox resources, we initiate 3 UDP flows each sending 20,000 1300-byte packets per second. Flow 1 undergoes basic forwarding and is active in time interval $(0, 15)$. Flow 2 undergoes statistical monitoring and is active in two intervals $(3, 10)$ and $(20, 30)$. Flow 3 undergoes IPSec encryption and is active in $(5, 25)$. The input queue of each flow can cache up to 1,000 packets. Fig. 10 shows the resource share allocated to each flow over time. Since flow 1 is bandwidth-bound and is the only active flow in $(0, 3)$, it receives 20% CPU share and all bandwidth. In $(3, 5)$, both flows 1 and 2 are active. They equally share the bandwidth on which both flows bottleneck. Later, when flow 3 becomes active at time 5, all three flows are backlogged in $(5, 10)$. Because flow 3 is CPU-bound, it grabs only 10% bandwidth share from 2 and 3, respectively, yet is allocated 40% CPU share. Similar DRF allocation is also observed in subsequent time intervals. Through the whole process, we see that MR^3 quickly adapts to traffic dynamics, leading to nearly perfect DRF across flows.

Sensitivity: Our final experiment is to evaluate the performance sensitivity of MR^3 under a mixture of different packet size distributions and arrival patterns. The simulator generates 24 UDP flows with arrival rate 10,000 pkts/s each. Flows 1 to 8 undergo basic forwarding; 9 to 16 undergo statistical monitoring; 17 to 24 undergo IPSec encryption. The 8 flows passing through the same middlebox module is further divided into 4 groups. Flows in group 1 send *large* packets with 1400 B; Flows in group 2 send *small* packets with 200 B; Flows in group 3 send *bimodal* packets that alternate between small and large; Flows in group 4 send packet with *random* size uniformly drawn from 200 B to 1400 B. Each group contains exactly 2 flows, with exponential and constant packet interarrival times, respectively. The input queue of each flow can cache up to 1,000 packets. The simulation lasts for 30 seconds. Fig. 11a shows the dominant services received by all 24 flows, where no paritcular pattern is observed in response to distribution changes of packet sizes and arrivals. Figs. 11b, 11c and 11d show the average single packet delay observed in three middlebox modules, respectively. We find that while the latency performance is highly consistent under different arrival patterns, it is affected by the distribution of packet size. In general, flows with small packets are slightly preferred and will see smaller latency than those with large packets. Similar preference for small-packet flows has also been observed in



(a) Dominant service received. (b) Basic forwarding. (c) Statistical monitoring. (d) IPSec encryption.

Fig. 11. Fairness and delay sensitivity of MR^3 in response to mixed packet sizes and arrival distributions.

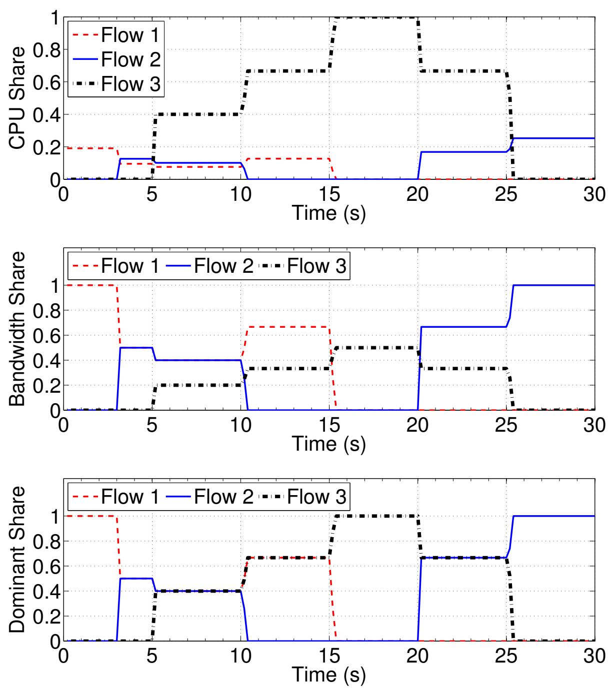


Fig. 10. MR^3 can quickly adapt to traffic dynamics and achieve DRF across all 3 flows.

our experiments with DRFQ.

VII. Concluding Remarks

The potential congestion of multiple resources in a middlebox complicates the design of packet scheduling algorithms. Previously proposed multi-resource fair queueing schemes require $O(\log n)$ complexity per packet, where n is the number of backlogged flows. With a large n, these schemes are hard to implement at high speeds. In this paper, we present MR^3, a multi-resource fair queueing algorithm with $O(1)$ time complexity. MR^3 serves flows in a round robin fashion. It keeps track of the work progress on each resource and withholds the scheduling opportunity of a packet until the progress gap between any two resources falls below one round. Through such a simple scheduling deferral, MR^3 avoids the unfairness problem suffered by direct extensions to single-resource round robin. Our theoretical analyses have indicated that MR^3 implements near-perfect DRF across flows. The price we have paid is a slight increase of packet latency. We have also validated our theoretical results via extensive simulation studies. To our knowledge, MR^3 is the first multi-resource fair queueing algorithm that offers near-perfect fairness with $O(1)$ time complexity. We believe that MR^3 should be easy to implement, and may find applications in other multi-resource scheduling contexts where jobs must be scheduled as entities, *e.g.*, VM scheduling inside a hypervisor.

References

[1] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI*, 2012.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM*, 2012.

[3] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proc. ACM IMC*, 2011.

[4] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. ACM SIGCOMM*, 1989.

[5] A. Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, 1993.

[6] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Towards high performance virtual routers on commodity hardware," in *Proc. ACM CoNEXT*, 2008.

[7] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *Proc. SIGCOMM*, 2011.

[8] S. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *Proc. IEEE INFOCOM*, 1994.

[9] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, 1996.

[10] P. Goyal, H. Vin, and H. Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 690–704, 1997.

[11] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proc. ACM SIGCOMM*, 2012.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX NSDI*, 2011.

[13] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proc. ACM Hotnets*, 2012.

[14] J. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible open middleboxes with commodity servers," in *Proc. ACM/IEEE ANCS*, 2012.

[15] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Netw.*, vol. 3, no. 4, pp. 365–386, 1995.

[16] S. Kanhere, H. Sethu, and A. Parekh, "Fair and efficient packet scheduling using elastic round robin," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 324–336, 2002.

[17] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Predicting the resource consumption of network intrusion detection systems," in *Recent Advances in Intrusion Detection (RAID)*, vol. 5230. Springer, 2008, pp. 135–154.

[18] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switching networks," in *Proc. ACM SIGCOMM*, 1990.

[19] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proc. IEEE*, vol. 83, no. 10, pp. 1374–1396, 1995.

[20] W. Wang, B. Liang, and B. Li, "Multi-resource generalized processor sharing for packet processing," in *Proc. ACM/IEEE IWQoS*, 2013.

[21] J. Bennett and H. Zhang, "WF^{2}Q: Worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM*, 1996.

[22] S. Ramabhadran and J. Pasquale, "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," in *Proc. ACM SIGCOMM*, 2003.

[23] "Cisco GSR," http://www.cisco.com/.

APPENDIX A
STARTUP LATENCY ANALYSIS

Proof of Theorem 3: Without loss of generality, suppose n flows are backlogged in round $k-1$, with flow 1 served the first, followed by flow 2, and so on. Upon flow n finishes service on resource 1, flow $n+1$ becomes active and is appended to the tail of the active list. Flow $n+1$ is therefore served right after flow n in round k. In particular, suppose flow $n+1$ becomes active at time 0. Let $S_{i,r}^k$ be the time when flow i starts to receive service in round k on resource r, and let $F_{i,r}^k$ be the time when flow i finishes service in round k on resource r. Specifically, $F_{n,1}^{k-1} = 0$. As shown in Fig. 12, the startup latency of flow $n+1$ is

$$\mathrm{SL}_{n+1} = F_{n,1}^k \ . \tag{19}$$

The following two relationships are useful in the analysis. For all flows $i = 1, \ldots, n$ and resources $r = 1, \ldots, m$, we have

$$F_{i,r}^k \le S_{i,r}^k + D_i^k \le S_{i,r}^k + 2L \ , \tag{20}$$

where the last inequality holds because of Corollary 2. Further, for all flows $i = 1, \ldots, n$, we have

$$S_{i,1}^k = \begin{cases} \max\{S_{1,m}^{k-1}, F_{n,1}^{k-1}\}, & i = 1 \\ \max\{S_{i,m}^{k-1}, F_{i-1,1}^{k}\}, & i = 2, \ldots, n \ . \end{cases} \tag{21}$$

That is, flow i is scheduled in round k after its previous-round service starts on the last resource (in this case, the progress gap on two resources is no more than 1 round) and its previous flow has finished service on resource 1.

The following two lemmas are also required in the analysis.

Lemma 3: For all flows $i = 1, \ldots, n$ and all resources $r = 1, \ldots, m$, we have

$$\begin{cases} S_{i,r}^{k-1} \le 2(i+r-2)L \ , \\ F_{i,r}^{k-1} \le 2(i+r-1)L \ . \end{cases} \tag{22}$$

Proof of Lemma 3: We observe the following relationship for all resources $r = 2, \ldots, m$:

$$S_{i,r}^{k-1} \le \begin{cases} \max\{F_{n,r}^{k-2}, F_{1,r-1}^{k-1}\}, & i = 1, \\ \max\{F_{i-1,r}^{k-1}, F_{i,r-1}^{k-1}\}, & i = 2, \ldots, n. \end{cases} \tag{23}$$

That is, flow i starts to receive service on resource r no later than the time when it finishes service on resource $r-1$ and the time when its previous flow finishes service on the same resource (see Fig. 12).

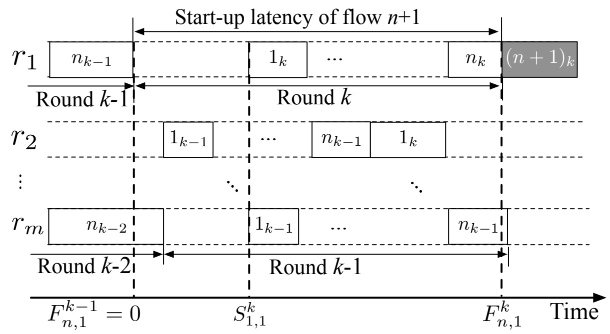


Fig. 12. Illustration of the startup latency. In the figure, flow i in round k is denoted as i_k. Flow $n+1$ becomes active when flow n finishes service on resource 1 in round $k-1$, and will be served right after flow n in round k.

To see the statement, we apply induction to r, i. First, when $r = 1$, the statement trivially holds because

$$S_{i,1}^{k-1} \le F_{i,1}^{k-1} \le 0, \quad i = 1, \ldots, n. \tag{24}$$

When $r = 2, i = 1$, we have

$$\begin{cases} S_{1,2}^{k-1} \le \max\{F_{n,2}^{k-2}, F_{1,1}^{k-1}\} \le 2L \ , \\ F_{1,2}^{k-1} \le S_{1,2}^{k-1} + 2L \le 4L \ . \end{cases} \tag{25}$$

This is because

$$F_{1,1}^{k-1} \le F_{n,1}^{k-1} = 0 \ , \tag{26}$$

and

$$\begin{aligned} F_{n,2}^{k-2} &\le F_{n,m}^{k-2} \\ &\le S_{n,m}^{k-2} + 2L \quad &\text{(By (20))} \\ &\le S_{1,1}^{k-1} + 2L \quad &\text{(By MR}^3\text{ algorithm)} \\ &\le 2L \quad &\text{(By (24))} \ . \end{aligned} \tag{27}$$

Now assume for some r, i and $r-1, i+1$, the statement holds. Note that for $r, i+1$, we have

$$\begin{aligned} S_{i+1,r}^{k-1} &\le \max\{F_{i,r}^{k-1}, F_{i+1,r-1}^{k-1}\} \quad &\text{(By (23))} \\ &\le 2(i+r-1)L, \quad &\text{(By induction)} \end{aligned} \tag{28}$$

and

$$F_{i+1,r}^{k-1} \le S_{i+1,r}^{k-1} + 2L \le 2(i+r)L \ . \tag{29}$$

Therefore, the statement holds for $r, i = 1, \ldots, n$. We then consider the case of $r+1, 1$. We have

$$\begin{aligned} S_{1,r+1}^{k-1} &\le \max\{F_{n,r+1}^{k-2}, F_{1,r}^{k-1}\} \quad &\text{(By (23))} \\ &\le \max\{F_{n,m}^{k-2}, 2rL\} \\ &\le \max\{2L, 2rL\} \quad &\text{(By (27))} \\ &= 2rL \ , \end{aligned} \tag{30}$$

and

$$F_{1,r+1}^{k-1} \le S_{1,r+1}^{k-1} + 2L \le 2(r+1)L \ . \tag{31}$$

Hence by induction, the statement holds. □

Lemma 4: The following relationship holds for all flows $i = 1, \ldots, n$:

$$\begin{cases} S_{i,1}^k \le 2(m+i-2)L \ , \\ F_{i,1}^k \le 2(m+i-1)L \ . \end{cases} \tag{32}$$

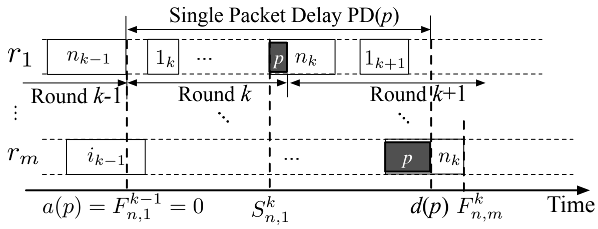


Fig. 13. Illustration of the packet latency, where flow i in round k is denoted as i_k. The figure shows the scenario under which the latency reaches its maximal value: a packet p is pushed to the top of the input queue in one round but is scheduled in the next round because of the account deficit.

Proof of Lemma 4: For $i = 1$, we have

$$\begin{aligned} S_{1,1}^k &= \max\{S_{1,m}^{k-1}, F_{n,1}^{k-1}\} && \text{(By (21))} \\ &\le \max\{2(m-1)L, 0\} && \text{(By Lemma 3)} \\ &= 2(m-1)L \ , \end{aligned} \tag{33}$$

and

$$F_{1,1}^k \le S_{1,1}^k + 2L \le 2mL \ . \tag{34}$$

Assume the statement holds for some i. Then for flow $i+1$, we have

$$\begin{aligned} S_{i+1,1}^k &= \max\{S_{i+1,m}^{k-1}, F_{i,1}^k\} && \text{(By (21))} \\ &\le 2(i+m-1)L && \text{(By Lemma 3 and (32))} \end{aligned}$$

and

$$F_{i+1,1}^k \le S_{i+1,1}^k + 2L = 2(i+m)L \ . \tag{35}$$

Hence by induction, the statement holds. □

Applying Lemma 4 to (19) leads to the statement. ■

Appendix B
Single Packet Delay Analysis

Proof of Theorem 4: For any packet p, let $a(p)$ be the time when packet p reaches the head of the input queue and is ready for service. Let $d(p)$ be the time when packet p finishes processing on all resources and leaves the system. The single packet delay of packet p is defined as

$$\text{SPD}(p) = d(p) - a(p) \ . \tag{36}$$

Without loss of generality, assume packet p belongs to flow n, and is pushed to the top of the input queue at time 0 in round $k-1$. The delay $\text{SPD}(p)$ reaches its maximal value when packet p is scheduled in the next round k, as shown in Fig. 13.

We use the same notations as those in the proof of Theorem 3. Let $S_{i,r}^k$ be the time when flow i starts to receive service in round k on resource r, and let $F_{i,r}^k$ be the time when flow i finishes service in round k on resource r. As shown in Fig. 13, the packet latency is

$$\text{SPD}(p) = d(p) \le F_{n,m}^k \ . \tag{37}$$

We claim the following relationships for all flows $i = 1, \ldots, n$ and resources $r = 1, \ldots, m$, with which the statement holds.

$$\begin{cases} S_{i,r}^k \le 2(m+n+i+r-2)L \ , \\ F_{i,r}^k \le 2(m+n+i+r-1)L \ . \end{cases} \tag{38}$$

To see this, we extend (23) to round k by replacing $k-1$ in (23) with k:

$$S_{i,r}^k \le \begin{cases} \max\{F_{n,r}^{k-1}, F_{1,r-1}^k\}, & i = 1, \\ \max\{F_{i-1,r}^k, F_{i,r-1}^k\}, & i = 2, \ldots, n. \end{cases} \tag{39}$$

We now show (38) by induction. First, by Lemma 4, (38) holds when $r = 1$. Also, for $r = 2, i = 1$, we have

$$\begin{aligned} S_{1,2}^k &\le \max\{F_{n,2}^{k-1}, F_{1,1}^k\} && \text{(By (39))} \\ &\le \max\{2(n+1)L, 2mL\} && \text{(By (22), (32))} \\ &\le 2(m+n+1)L \ , \end{aligned} \tag{40}$$

and

$$F_{1,2}^k \le S_{1,2}^k + 2L = 2(m+n+2)L \ . \tag{41}$$

Now assume for some r, i and $r-1, i+1$, (38) holds. Note that for $r, i+1$, we have

$$\begin{aligned} S_{i+1,r}^k &\le \max\{F_{i,r}^k, F_{i+1,r-1}^k\} && \text{(By (39))} \\ &\le 2(m+n+i+r-1)L, && \text{(By induction)} \end{aligned} \tag{42}$$

and

$$F_{i+1,r}^k \le S_{i+1,r}^k + 2L = 2(m+n+i+r)L \ . \tag{43}$$

Therefore, by induction, (38) holds for $r, i = 1, \ldots, n$. We then consider the case of $r+1, 1$. We have

$$\begin{aligned} S_{1,r+1}^k &\le \max\{F_{n,r+1}^{k-1}, F_{1,r}^k\} && \text{(By (39))} \\ &\le \max\{2(n+r)L, 2(m+n+r)L\} && \text{(By Lemma 3)} \\ &= 2(m+n+r)L, \end{aligned}$$

and

$$F_{1,r+1}^k \le S_{1,r+1}^k + 2L = 2(m+n+r+1)L \ . \tag{44}$$

Hence by induction, (38) holds. ■