Implementation of WFQ in a Distributed Open Software Router

Azeem J. Khan, Anirudha Sahoo, D. Manjunath
{azeem,sahoo}@cse.iitb.ac.in, dmanju@ee.iitb.ac.in
IIT Bombay, Mumbai, India.

***Abstract*—There has been a considerable body of research devoted to the design and performance of PC based software routers running open source software. Most of the research on open software routers (OSRs) have focused on improving the performance of single PCs with a few proposals for a distributed design. Modern routers are equipped with enhanced functionality such as QoS features in addition to packet forwarding. However, providing enhanced functionality in distributed OSR architectures has largely remained unaddressed. A distributed design introduces challenges in its implementation due to looser coupling between different subsystems.**

WFQ is a widely used scheduler that enables QoS features in a router. The inherent centralized nature of the design of WFQ schedulers in most switches and routers creates several challenges when exported to distributed architectures. In this paper, we study the challenges of implementing WFQ in a distributed OSR, propose some novel techniques to address these challenges and compare the performance of our WFQ implementation in distributed OSR with that of a centralized WFQ scheme.

***Index Terms*—QoS, WFQ, software routers, distributed**

I. Introduction

The programmability of the general purpose CPU based PC has led to the creation of several robust and high performance software such as Zebra, Quagga, Linux, Click to perform packet processing in software. Software routers that utilize open source software and open hardware are termed as Open Software Routers (OSRs). Recently, OSRs have gained the attention of several researchers because of their ability of high performance packet processing while retaining all the advantages of an open platform [4]–[6], [9]–[11]. Most recent research has focused only on the performance of single PC based OSRs. Certain performance bottlenecks related to I/O and computational capacity in single PC OSRs can be overcome in a distributed configuration. In fact, large routers using commodity PCs can only be achieved by a distributed configuration. Therefore, a distributed design that provides a single logical router has been proposed by some researchers [5], [9], [11]. A distributed architecture introduces several challenges as we shall see below.

In recent years, routers have had to provide significantly more functionality. Some of these new functionalities are needed to support firewalls, encryption capabilities, QoS and per user forwarding rules. Some of these functionalities can be easily extended to a distributed architecture because of the inherent parallel processing characteristic of packet forwarding. Some other functionalities, such as providing QoS, due to the inherent assumptions on the centralised nature

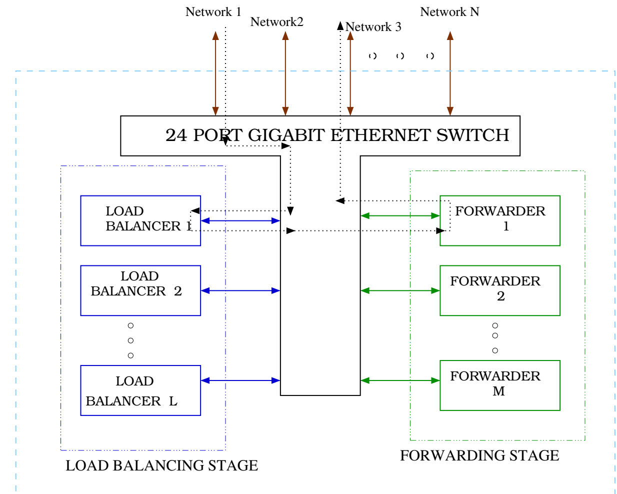

Fig. 1. The Switched Two-stage Distributed Router (STSDR). The router comprises of two stages: a load balancer stage and a packet forwarding stage comprised of forwarders. Each load balancer and forwarder is a Linux PC. A dotted line shows how ingress packets are directed by a load balancer to a forwarder where they are processed before being sent to the next hop.

of processing and communications, are not easily extendable to a distributed architecture. We are therefore interested in incorporating the implementation of such a functionality in a distributed architecture.

The distributed OSR design considered in this paper is the 'Switched Two-stage Distributed Router' (STSDR) [2] shown in Figure 1. The STSDR has two stages; a load balancing stage and a packet forwarding stage. The PCs in load balancing stage are responsible for directing packets to the forwarders or forwarding engines (FEs). The FEs perform layer-3 packet processing on the packets. The FEs and the load balancers are connected to each other by a regular layer-2 Gigabit Ethernet switch, represented as T-switch because of its shape. The external interfaces to the STSDR terminate on the switch. However, the IP addresses visible to the outside world are actually assigned to the switch interfaces on which the load balancers are connected. The arrows on the dotted line show the direction of packet flow through the STSDR. Packets from an external network enter the STSDR through the switch. ARP resolution by external router would give the MAC address of one of the load balancer interface as the layer-2 address where the packet should be forwarded. Thus, the T-switch sends the packet to the correct balancer. The load balancer

uses a suitable criteria to select a forwarder and sends the packet to that forwarder. The forwarder receives the packet, processes it and selects the next hop for the packet. Since the forwarder and the next hop are directly connected via the T-switch, it executes the ARP, finds the layer-2 address of the next hop and transmits the packet. The packet enters the T-switch through the forwarder's network interface and is transmitted to the next hop. It is assumed that all packets from a traffic flow are directed to the same forwarder through mechanisms such as hashing. We observe from Figure 1 that the distributed nature of processing will necessarily create challenges to the implementation of any feature that provides services aggregated over the entire system.

Our work is motivated by the following reasons. Today, medium and high end routers are typically equipped with QoS features. WFQ is one of the most widely used scheduling algorithms which enables many QoS features (e.g., DiffServ). The STSDR proposal is a high performance distributed architecture and should therefore support implementation of WFQ. WFQ implementation in centralized routers is very well understood and is quite mature. However, implementing WFQ in a distributed system such as the STSDR has several challenges. In our STSDR implementation, we observed that a lot of computing power was left over after implementing basic routing functionality [2], [11]. Hence we choose to implement WFQ to take advantage of the available computing capacity in the STSDR. In this paper, we shall study the challenges in the implementation of WFQ in distributed OSR and address those challenges. Note that our aim is not to provide the best possible implementation of WFQ in distributed OSR but to investigate the issues and provide solutions to those issues with reasonably good performance.

Our principal contributions are as follows. First, we analyze the challenges of implementing WFQ on a distributed system. Second, we propose a mechanism to implement WFQ on the distributed OSR with a novel approach. Third, we show through experiments on a real system, the performance impact of our proposed implementation. Fourth, we perform an experimental comparison of the fairness of our implementation using a real router's traffic trace.

II. Background

A. Linux Open Software Routers

There is precedent both in recent years as well as in the early days of the Internet for OSRs. In the data plane, early versions of BSD Unix and other Unix based operating systems performed routing entirely in software. In a number of simple environments, Linux and BSD Unix can be easily used for packet routing. Newer data plane software like Click [13] have been introduced for the Linux and Linux/BSD Unix environment respectively. In [4], [6], it was shown that Linux based PCs could achieve high performance as packet forwarders. Further improvements in performance of a Linux based OSR was achieved through optimizations in Linux(e.g., [7]). Some researchers have proposed using programmable Graphical Processing Units (GPUs) to perform packet forwarding computations [10].

In [5], it was shown that a distributed architecture was necessary and feasible to scale an OSR's performance and a control plane for the distributed design was proposed in [3]. Routebricks [9] defines an architecture that parallelizes the packet forwarding functionality across multiple PCs and across multiple cores of the CPUs within each PC. A technique to improve performance of a distributed OSR was studied in [2]. Thus considerable research effort has been devoted to analyzing and improving the performance of OSRs in both a single PC and in distributed configurations. However, implementing additional features on the distributed OSRs remains unaddressed.

B. WFQ Implementations

Most implementation proposals for WFQ utilize the centralized nature of the operation of a switch and assume fixed-size packets to provide fair queuing [1], [14]. In typical WFQ implementations on centralized routers, a WFQ algorithm at each egress port executes independently and in parallel with those at other egress ports [15]. This implementation is ideal for centralized routers since all packets queued at an egress port are destined on the same outgoing link. In the distributed OSR, packets for different outgoing links may be queued at the same egress port and packets at different egress ports may be destined to the same outgoing link. Hence, adapting existing WFQ implementation techniques of centralized routers to the distributed OSR is not possible. One solution could be to emulate centralized operation for WFQ in the distributed OSR. However, this requires frequent communication between the distributed egress ports which is a very expensive overhead. Moreover, emulating centralized WFQ in a distributed setting needs tight time synchronization. Therefore emulation of centralized WFQ scheduler is not efficient and a distributed implementation of WFQ is necessary.

A distributed fair queuing mechanism for network switches has also been proposed [16]. In [16], where several distributed fair queuing schedulers operate to provide approximately the same service as a centralized single fair queuing scheduler would provide when operating at the output port. The distributed schedulers in [16] operate at both the input and the output ports of a switch and communicate frequently to update their global state information. Although this research proposal is a distributed design, it cannot be adapted to the distributed OSR because (1) frequent communication is expensive in the distributed OSR (2) output ports are non-programmable and hence cannot execute scheduler processes. In the wireless domain also, there have been research proposals on distributed fair queuing (e.g., [18]). These proposals usually exploit the broadcast nature of the wireless medium to a considerable degree to meet their goals. Hence, these proposals cannot be applied to the distributed OSR.

Thus we have seen that most research proposals on WFQ implementations cannot be used in the distributed OSR because (1) expensive communication overheads prevent em-

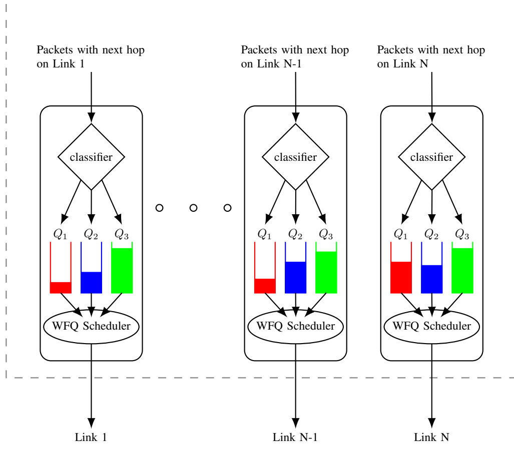

Fig. 2. WFQ in a centralized router: Packets for transmission on each output link arrive at the corresponding line cards where they are classified, queued and scheduled for transmission.

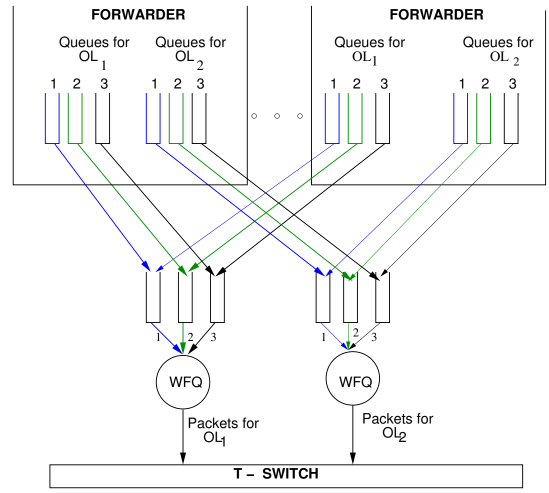

Fig. 3. Emulation of centralized WFQ in the STSDR. We assume two outgoing links, two forwarders and three classes of traffic in the figure. Packets from different FEs of the same class are ordered in their arrival sequence and WFQ is executed.

ulation of centralized behavior (2) the next hop links are not tied to a specific egress port (3) the output ports are not programmable and (4) the distributed OSR's processing nodes do not communicate over a broadcast medium. We therefore propose some new techniques to adapt the central ideas present in input queued switches that emulate output queued switches for the purpose of providing QoS [8], [17]. In brief, the distributed WFQ problem is converted into a matching problem with the requirement that the matching is performed in a distributed fashion but identical results are obtained at each processing node.

III. Implementation Challenges for WFQ in STSDR

In Section II-B, we briefly discussed WFQ implementation proposals on network switches and a centralized OSR. In this section, we shall first discuss in greater detail the typical implementation of WFQ in a centralized router and then describe the challenges that arise in a distributed configuration.

A. *WFQ in Centralized Software Routers*

Consider Fig. 2 which shows one possible implementation of WFQ on a centralized router. The router has N egress ports each on its own line card and it classifies packets into K different classes. In Figure 2, $K = 3$. The scheduling proceeds as follows. Once the packet reaches the egress line card, the packet is classified into one of three classes and enqueued in the queue meant for that class. The WFQ algorithm is executed at each line card before transmission of a packet. One packet is chosen from Q_1, Q_2 or Q_3 by the WFQ scheduler and then transmitted on the outgoing link. The classification and WFQ execution are independent of each other and may occur in parallel. Since each outgoing link is directly connected to the line card, the WFQ algorithm at each line card executes independently and in parallel with those of other line cards. This level of parallelism can be extended to more egress ports per line card.

In the distributed OSR, the operation at egress ports is quite different. Consider Figure 1. We observe that the outgoing links (OLs) from the STSDR to next hops are connected to the T-switch and not the FEs. Packets from any FE can be transmitted on an OL and any OL can carry packets from a FE. Let us assume there are M FEs and N outgoing links and K classes of traffic. Therefore, in the STSDR, each FE has $N \times K$ queues. If we denote the queue of class k for outgoing link OL_j at forwarder FE_i as Q_{ijk}, then we have $1 \leq i \leq M$, $1 \leq j \leq N$ and $1 \leq k \leq K$.

If we are to emulate the WFQ behaviour of the centralized router in the STSDR, the WFQ implementation will take the form shown in Figure 3. The packets of a traffic class k meant for OL_j exist in each of the M FEs. These packets are ordered in the sequence of their arrival times and WFQ is executed over the K classes for OL_j. The selected packet is then transmitted by the appropriate FE. Although in principle this process is straightforward, in practice there are a number of implementation challenges.

B. *Distributed state*

Let us assume that the WFQ algorithm for each OL is separate and is executed in one of the STSDR nodes. The algorithm uses queue information for scheduling and this information represents the *state* for the algorithm. In the centralized system, the K class queues reside locally on the same line card and querying the state is a local memory access. In the STSDR, each forwarder is a physically discrete entity with local memory and hence there is no shared memory across the FEs. Therefore, the WFQ algorithm cannot query all the queues meant for the same outgoing link as no FE has state information about the other FEs. Thus explicit communication of state information is necessary between forwarders. Once state information is known, the head of the line (HoL) packets of different FEs meant for the same outgoing link need to be

ordered according to their arrival times. This is an issue that we study below in Section III-C. Also, the WFQ scheduler's decision must be communicated to all the forwarders.

C. Time synchronization

Let us assume that the forwarders have communicated in some manner and attained global state synchronization. As shown in Figure 3, packets from queues of the same class and destined to the same OL, for example Q_{111} and Q_{211}, should be merged into a single queue. This ordering requires precise knowledge of the arrival times of packets into the queues. This requires that all the forwarders are perfectly time synchronized in their clocks which is not the case.

The forwarders and the load balancers in the STSDR use NTP for time synchronization using in-band signaling over the links to the T-switch. NTP allows time synchronization at the granularity of a few milliseconds. With links operating at gigabit speeds, this level of time synchronization is not sufficient. The time synchronization problem can be removed with expensive high-precision mechanisms using out of band signaling. Even with perfect time synchronization, communication between forwarders will be necessary for sharing state information. However, as we shall see in Section IV, we avoid the need for tight time synchronization by using a rather novel approach.

D. Output link arbitration

In the absence of a coordinated transmission, multiple forwarders may attempt to transmit packets on the same OL simultaneously. This may result in queuing at the OL and this can violate the fairness requirements of WFQ in the short term. On the other hand, in the absence of time synchronization and coordinated transmission, it is possible that a FE delays its transmission assuming that a particular OL is busy. Such delays should be avoided. It is also possible that the buffers in the port of the T-switch may be insufficient for handling high number of contentions. Simultaneous transmission on some ports may therefore result in dropped packets and reduced throughput. Thus, arbitration of an OL for a short period of time is beneficial.

E. Expensive communication

We noted above that global state synchronization across all forwarders is necessary for WFQ. Since explicit communication is necessary, broadcast or multicast of state information should be used. In the STSDR, as in most modern computing systems, computation is cheap and communication across forwarders is very expensive. With a packet being processed every few hundred nanoseconds on a Gigabit link, a communication overhead of even a few milliseconds is significant. Hence communication overheads are to be avoided as far as possible to maintain throughput.

In summary, for the STSDR, we require a technique that is able to synchronize the state of all forwarders and arbitrate the link usage while satisfying the WFQ-based bandwidth sharing without compromising performance.

IV. WFQ Implementation on the STSDR

In this section, we propose a distributed WFQ scheduling implementation scheme that overcomes the challenges described in Section III. First, we briefly explain the central ideas for the implementation mechanism. Then we go into much greater detail about each challenge. Finally, we make a quick comparison of our implementation with that of a centralized WFQ scheduler.

We now introduce some notations which we will use to explain the implementation of WFQ on the STSDR. Let a_{ijk} be the arrival time of a packet in the queue Q_{ijk}. a_{ijk} is stamped with respect to the real clock of FE_i. We refer to the virtual finish time of the HoL packet of Q_{ijk} as v_{ijk}. Each outgoing link OL_j has a virtual clock V_j which is always updated to the virtual finish of the last packet transmitted on OL_j. Since each FE_i knows which packets were last transmitted on OL_j, each FE_i knows the value V_j at all times. The packet from FE_i sent out for communicating state information is identified as DS_i.

The procedure for the WFQ selection and transmission of packets at each forwarder consists of three separate phases. In the first phase each forwarder broadcasts information about the HoL packets in its $K \times N$ queues to other forwarders. Thus, at the end of the first phase, information about the HoL packets at all $K \times N \times M$ queues of the entire STSDR systems is known to every other FE.

In the second phase, FE_i selects one packet for transmission per OL_j (by running the WFQ algorithm over all the queues destined to OL_j). Since each FE_i knows the queue information about other FEs, it also determines packets to be transmitted per OL_j by other FEs. Each FE_i knows the V_j for every OL_j. All this information is utilized in the third phase.

In the third and final phase, the stable matching algorithm [12] is executed at every FE with inputs derived from second phase. The algorithm outputs a set of matched FE and OL pair $\{(FE_i, OL_j)\}$. The number of elements in this matched set will be $\min(M, N)$. A matched pair (FE_i, OL_j) implies that FE_i will transmit the packet selected by itself (by running WFQ over all the queues destined to OL_j) over OL_j. The three phases together constitute a cycle and this cycle is repeated. We discuss each of the three phases in greater detail next.

A. Communication for global state synchronization

The forwarders use the standard broadcast mechanism at the link layer, i.e., Ethernet broadcast frames to synchronize global state information. Each FE_i broadcasts the HoL packet's information for each of its $K \times N$ queues in the DS_i packet. A master-slave paradigm is adopted for disseminating state information. One of the forwarders is assigned to be the master and the remaining forwarders behave as slaves. The master forwarder broadcasts a DS frame which triggers a broadcast reply from the slaves.

In the master FE, the contents of the DS packet is assembled as explained below. This is then encapsulated in an Ethernet frame. Using raw Ethernet frames avoids processing delay

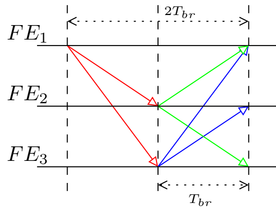

Fig. 4. Broadcasts of queue status information for global state synchronization

at higher layers in the network stack. Since a link layer broadcast does not offer reliability guarantees, there is a chance that DS packets may be lost. However, additional reliability mechanisms may be introduced if needed. For the Ethernet frame, the destination address is set to the broadcast Ethernet address and the Ethertype is set to a custom value. The master broadcasts this Ethernet frame to the slaves. An internal VLAN is employed to restrict these broadcasts to STSDR nodes. When the slave forwarders receive such a broadcast frame from the master forwarder, they construct their own DS packets and broadcast it to all forwarders.

We illustrate this with an example in Figure 4. FE_1 is assigned to be the master FE. FE_2 and FE_3 are slaves. FE_1 constructs DS_1 and broadcasts it. Let us say that the time for the broadcast to be received by all FEs is T_{br}. At the end of at most T_{br}, FE_2 constructs DS_2 and FE_3 constructs DS_3 and broadcasts them over the internal VLAN. The value of T_{br} depends on the link-layer technology and topology and therefore must be found through experimentation. After at most $2T_{br}$, each forwarder has information about the HoL packets in all the queues of all forwarders.

The structure of the DS packets is as follows. Two quantities about each HoL packet are contained in the DS_i packet. The first is the arrival time-stamp (four bytes) and the second is the packet length (two bytes). Thus six bytes per queue are needed. These values can be modified for different setups if the time-stamp resolution and/or the maximum packet lengths are different. A two byte header identifies FE_i. Each DS_i therefore has a body of $(6 \times N \times K + 2)$ bytes.

Robustness of the information exchange is attained through several procedures. To ensure timeliness, a timer is set for T_{br} at each slave FE_i once the slave receives the master DS packet. Only after waiting for T_{br}, all the FEs enter the second phase of packet scheduling.

One issue is that of a DS packet not being received by an FE. DS_i packets are sent in every cycle. However, the information $\{a_{ijk}, L_{ijk}\}$, $k = 1, 2, \ldots K$, does not change if no packet was transmitted from FE_i to OL_j in the previous cycle. Thus, it is possible to reuse stale information in most cases. Thus, it is clear that if a DS packet is not received by a forwarder, it does not result in a critical failure. In the special case where a slave receives DS packets from other forwarders before hearing from the master forwarder, it behaves like other slaves and broadcasts its own DS packet. In the rare event of the master FE going silent, mechanisms for leader election in distributed systems may be adopted to select a new master forwarder.

At the end of the first phase, each FE_i has information about HoL packets in all the queues of all other FEs. Thus each FE has the necessary information of all the queues in the entire STSDR system. We now move onto the second phase of packet scheduling.

B. Creation of preference lists at each forwarder

Many forwarders may have packets destined on the same OL. Therefore arbitration is necessary to match a forwarder with an outgoing link. The scheduling algorithm uses stable matching to arrange this match. The stable matching algorithm requires a preference list from each of the forwarders and each of the outgoing links. Therefore, in the second phase, sets of preference lists are created.

First, we obtain the preference list of the forwarders. Each forwarder has one packet destined to each OL (after executing the WFQ algorithm on its class queues per OL). Therefore for every OL_j at FE_i, we will get

$$P_{ij} = WFQ(Q_{ij1}, Q_{ij2}, ..., Q_{i1K}) \quad (1)$$

where WFQ($\cdot$) represents running WFQ algorithm on the set of HoL packets in the queues indicated in the argument. Since there are N outgoing links in the STSDR, FE_i has a sequence of N packets $(P_{i1}, ..., P_{iN})$ for each OL_j. We obtain the preference list of FE_i by sorting these N packets in increasing order of their arrival times (i.e., earliest arrival first). Since there are M FEs and N OLs, there are M forwarder preference lists each containing N elements.

We illustrate this with an example where $M = 2$, $N = 2$ and $K = 3$ as shown in Figure 5. Consider the example of FE_1. First, WFQ is performed on Q_{111}, Q_{112} and Q_{113} for OL_1 to obtain P_{11}. Similarly P_{12} is obtained. P_{11} and P_{12} are then sorted by the algorithm ASORT (a simple sorting algorithm), in the non-decreasing order of their arrival times. For example if $a_{12} < a_{11}$, the ordered sequence is (P_{12}, P_{11}). This ordered list is X_1 and represents the preference list of FE_1. Similarly X_2 is computed for FE_2.

Now, we obtain the preference list of all the outgoing links. As before, we assume that each forwarder has one packet destined to OL_j. Since there are M forwarders, each OL_j has a sequence (P_{1j},P_{2j},...,P_{Mj}.) of packets destined to it. We obtain the preference list of OL_j by sorting this sequence of packets in the non-decreasing order of their virtual finish times. Since there are N OLs, N preference lists are produced and each list contains M elements.

We note here that OL_j cannot choose the FE with the packet having the lowest virtual finish time (say FE_i). This is because FE_i may have a packet destined to another OL (say $OL_{j'}$) whose arrival time is earlier than the one destined for OL_j. Hence there is a need for stable matching.

We return to the example in Figure 5. Consider the case for OL_1. P_{11} at FE_1 and P_{21} are the packets destined for OL_1. The algorithm VSORT (a simple sorting algorithm) orders the two packets in non-decreasing order of their virtual

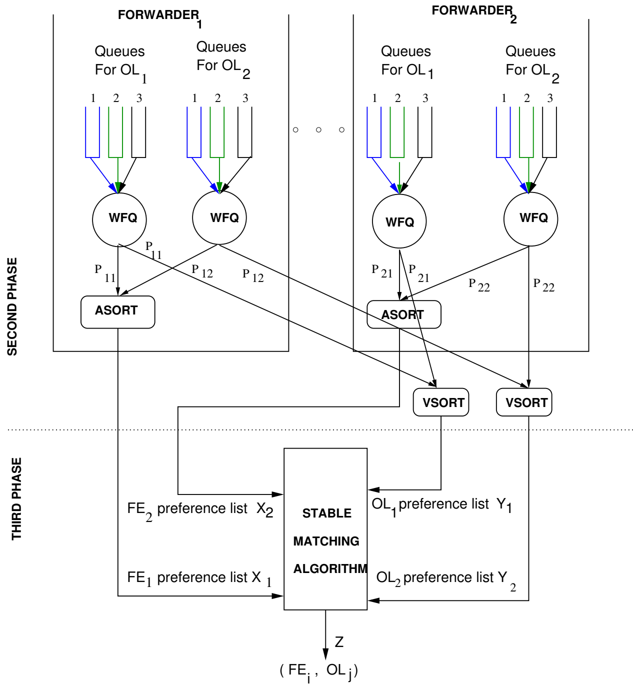

Fig. 5. WFQ packet scheduling in the STSDR. Phase one is assumed as completed. Assume two outgoing links, two forwarders and three classes of traffic. ASORT orders packets by their arrival times inside each FE. VSORT orders packets by their virtual finish times inside each FE (even though it is shown outside the FE). In the second phase, $X_i \forall i$ and $Y_j \forall j$ is computed. In the third phase, each FE_i is matched to an OL_j.

finish times to obtain Y_1. For example, if $v_{11} < v_{21}$, then the ordered sequence is (P_{11}, P_{21}). This ordered sequence is Y_1 and represents the preference list for OL_1. Similarly, Y_2 is obtained for OL_2.

In the actual implementation on the STSDR, each FE computes the preference lists for itself, for the other FEs and for the OLs independently. Hence, the computation is distributed and deterministic; i.e., all the FEs will end up having the same decision of order of scheduling. Note that the arrival time of packets belonging to different forwarders are never compared for a computation. Hence, the need for tight time synchronization between FEs for the purpose of WFQ scheduling is avoided. The virtual clock V_j is maintained at each FE_i and hence a direct comparison of the virtual finish time is possible. We now move to the third phase of the scheduling process.

C. Link Selection at each forwarder

In this phase, the preference lists produced at the end of the second phase are fed as input into a stable matching algorithm [12]. The stable matching algorithm assigns each FE to a particular OL. At FE_i who is assigned OL_j, the outgoing link corresponds to a packet selected by the WFQ from one of its local queues destined to OL_j. The FE transmits that selected packet packet over OL_j and updates the virtual clock V_j. Once again we note that the stable matching algorithm is executed on every FE and the output of the algorithm is identical and deterministic at every FE_i. Hence, all FEs have exact knowledge of the packet transmitted by other FEs. All FEs therefore update the virtual clock for all the OLs assigned to the FEs.

Once all the transmissions are completed, the master forwarder once again initiates the queue status information exchange protocol and the cycle repeats. We thus have a technique that is able to satisfy the WFQ-based bandwidth sharing without compromising performance in a distributed OSR such as the STSDR. The algorithm of the scheduler at every FE_i is listed in Algorithm 1.

Algorithm 1 WFQ scheduler algorithm executed at every FE

```
Broadcast DS_i packet
Collect all DS_m packets from other FEs where m ≠ i, 1 ≤ m ≤ M
for i = 1 to M do
    {/* for all FEs */}
    P_i = () {/* Initialize empty ordered list */}
    for j = 1 to N do
        {/* For all OLs */}
        P_ij = WFQ(Q_ij1, ..., Q_ijK)
        Insert P_ij into P_i {/*P_i = (P_i1, P_i2, ..., P_iN) is the
        sequence of packets selected for transmission from
        FE_i to OL_1, OL_2, ..., OL_N respectively*/}
    end for
    Let X_i = ASORT(P_i1, P_i2, ..., P_iN) {/* X_i is a list
    of elements P_ij, in non-decreasing order of their arrival
    times.*/}
end for
for j = 1 to N do
    {/* for all OLs */}
    R_j = () {/* Initialize empty ordered list */}
    for i = 1 to M do
        {/* for all FEs */}
        Insert P_ij into R_j {/*R_j = (P_1j, P_2j, ..., P_Mj) is
        the sequence of packets selected for transmission by
        FE_1, FE_2, ..., FE_M respectively*/}
    end for
    Let Y_j = VSORT(P_1j, P_2j, ..., P_Mj) {/* Y_j is a list
    of elements R_j, in non-decreasing order of their virtual
    finish times */}
end for
Z = StableMatch((X_1, X_2, ..., X_M), (Y_1, Y_2, ..., Y_N)) {/* Z
is set of pairs. Each pair φ contains an FE and its matched
OL. Number of elements in Z = min(M, N)*/}
```

D. Comparison with centralized WFQ

There are two major differences between our implementation of the distributed WFQ and the centralized WFQ of a single PC router. The biggest difference is that in our implementation of the distributed WFQ, the scheduler is non work conserving. This occurs due to two reasons. IP packets are variable in length and it is possible for some forwarders to finish their transmission early and then wait for other

forwarders to complete their transmissions. This necessarily makes our implementation a non-work conserving one. A second reason is the latency introduced due to the three phases of the scheduling process. Although some pipelining of the computations does occur, this non-work conserving behavior does affect performance as we shall see shortly. The second difference is that the stable matching algorithm may cause some packets to be transmitted in a different sequence compared to a centralized scheduler.

In the next section, we report the results of the experiments that were performed on the distributed WFQ implementation proposed in this section.

V. Experimental Results

In this section we present the results of our experiments after implementing the distributed WFQ design. We present two sets of results. In the first set we compare the performance of the STSDR forwarder with WFQ. In the ideal case, introduction of WFQ functionality should not affect the throughput of the STSDR forwarder. In the second set we compare the fairness of the STSDR with a centralized WFQ scheduler. In the ideal case, the fairness of the distributed OSR implementation should remain the same as that of a centralised single PC OSR having the same number of interfaces as the total number of external interfaces on the STSDR. The hardware and software details of the experimental setup is listed in the Appendix.

For the performance experiments, two packet generators are used to inject packets into the two load balancers which then direct these packets to a forwarder. The generated packets have a randomly chosen destination IP address from a valid range and are of type UDP. The UDP port is also set randomly from a fixed range. The maximum queue length value at each forwarder was set at 10000 packets for each queue. One million 64-byte packets were generated and the average throughput noted for a set of twelve experiments. For the fairness experiments, each traffic class in the forwarders is assigned an equal weight w_k and the distributed WFQ scheduler is configured with these values. Ties occurring at the distributed WFQ scheduler are broken randomly. For both sets of experiments we chose $N = 3$ and $K = 3$. Thus, each forwarder transmits information about 9 queues in its DS_i packet.

A. Scheduling a single packet per cycle

In this experiment, a single packet is transmitted at the end of every cycle. We call this the single DS_i or S-DS_i approach. Initially, T_{br} was set to $20\mu s$. It was observed with $T_{br} = 20\mu s$ that several broadcast packets never arrived at the forwarders. Further experiments showed that even on idle forwarders, a $T_{br} \geq 50\mu s$ was necessary for all forwarders to receive DS_i. Note that a 100-byte packet inside an Ethernet frame transmitted on a 1Gbps link corresponds to a time of about $1\mu s$. The DS_i packets were only 50 bytes long. This implies a severe limitation in the operation of the commodity hardware based T-switch. An improvement in the processing speed of the T-switch should result in reduction of T_{br}. This also implies

t_{br} (μs)	Input Rate (pps)	Percentage of lost broadcast frames	Output rate (pps)	Percentage throughput
72	13888	2.31	12831	92.39
76	13157	1.82	12322	93.65
80	12500	2.05	11822	94.57

TABLE II
Packet throughput for distributed WFQ on the STSDR using the S-DS_i approach

that T_{br} is not constrained by the layer-2 technology used here (Gigabit Ethernet) or the topology of the distributed OSR.

We begin by comparing the fairness of the bandwidth allocation of our proposed implementation. One million packets, both of fixed length and variable lengths were transmitted (i.e., one million rounds of scheduling were conducted). We chose the fixed packet length to be 64 bytes. For variable packet lengths, we chose the values of packet lengths from a real trace (details in the Appendix). Packets from the trace were randomly assigned to one of three classes. We chose $T_{br} = 72\mu s$. In Table I, we report the number of bytes transmitted on outgoing link OL_1 for a centralized WFQ scheduler and the distributed WFQ scheduler. We observe that for fixed packet lengths, the difference between the schedulers is a maximum of 256 bytes, i.e., four packets. For variable length packets the difference is larger, especially Class #3 (about 0.0008%). A careful analysis revealed that the packets assigned to Class #3 packets had a lower mean packet length and hence fewer bytes on average were transmitted per cycle. We note that the bandwidth allocation of our distributed WFQ scheduler is very close to the centralized scheduler for all the classes. We now consider the performance of the distributed WFQ scheduler.

We compare the packet per second throughput performance of the WFQ implementation of the STSDR forwarder with the STSDR forwarder without WFQ. As mentioned before, in an ideal scenario, the throughputs in both the cases should be identical. In our experiments, the STSDR with no WFQ implementation achieved a throughput of $\sim$ 186 kpps. Table II lists the observed performance on a STSDR forwarder for different values of T_{br}. We observe that the forwarder throughput with WFQ is very low and there are some lost broadcast frames as well. We measured the execution time of the cycle's three phases. The first phase executed in $\sim 105\mu s$. The second phase took ∼600ns. The third phase of stable matching took ∼450ns. Clearly, the first phase dominates the total time taken for the entire cycle. The performance of the STSDR forwarder is limited by the first phase, as during the first phase no packets are being transmitted on the OLs. This issue can be addressed in several ways. We look at two particular techniques next.

B. Scheduling multiple packets per cycle

Our earlier experiments confirmed that the communication overhead is very expensive in the distributed OSR. Hence, a means of either avoiding communication or amortizing the

Packet Size	Class #1		Class #2		Class #3	
	Centralized	Distributed	Centralized	Distributed	Centralized	Distributed
64 bytes	21,333,312	21,333,3440	21,333,312	21,333,120	21,333,312	21,333,376
Variable	246,333,087	246,315,351	246,333,123	246,339,218	246,341,016	246,145,323

TABLE I
FAIRNESS COMPARISON BETWEEN CENTRALIZED AND $S\text{-}DS_i$. THE BYTES TRANSMITTED PER CLASS AT THE END OF 1 MILLION ROUNDS ARE NOTED. ALL CLASSES HAVE SAME WEIGHT.

Packets per DS_i	Throughput in kpps
15	115.3
20	130.9
25	150.6

TABLE III
PACKET THROUGHPUT FOR DISTRIBUTED WFQ ON THE STSDR USING THE $M\text{-}DS_i$ APPROACH

Packets per cycle	Throughput in kpps
15	130.4
20	149.9
25	174.2
100	183.1

TABLE IV
PACKET THROUGHPUT FOR DISTRIBUTED WFQ ON THE STSDR USING THE $L\text{-}DS_i$ APPROACH

cost of communication is necessary. The value of T_{br} was kept at $72\mu s$.

One approach is to transmit information about the first l packets of a queue in DS_i. Thus, distributed WFQ and stable matching can occur for up to l packets before state synchronization is necessary. We call this the multiple DS_i or the $M\text{-}DS_i$ approach. We vary l with three different values of 15, 20 and 25 shown in the first column. Since each packet's information consists of 6 bytes (4 bytes time-stamp and 2 bytes of packet length), information for about 25 packets per queue can be enclosed in one single Ethernet frame. The average throughput across the three links are reported. No DS_i losses were observed in this approach. We report the performance results of $M\text{-}DS_i$ in Table III. We observe that the throughput improves considerably from 12.8 kpps in the $S\text{-}DS_i$ approach to 115.3 kpps. The $M\text{-}DS_i$ technique is clearly far more efficient than $S\text{-}DS_i$. However, the maximum throughput still falls short of its potential value of $\sim$186 kpps due to three major reasons. First, about $2\mu s$ per cycle are utilized for the second and third phase of scheduling. Second, information of only about 25 packets can be accommodated in the maximum length Ethernet frame. Third, between two consecutive cycles, every FE has to wait for the longest packet to finish transmission before starting transmission of the next round thereby causing idle periods on most links. Thus we need an alternative technique that overcomes these issues.

The second approach that we consider is *lazy scheduling*. In lazy scheduling, the packet selection and link assignment happen as described in Section IV. However, instead of transmitting just one packet, l packets from the selected queue are transmitted. We call this the $L\text{-}DS_i$ approach. Transmitting l packets from a single queue will necessarily alter the short term fairness in the STSDR compared to the centralized WFQ or the $S\text{-}DS_i$ approach. However, there are two advantages to $L\text{-}DS_i$. First, phase two and three, i.e., distributed WFQ and stable matching, are executed only once every l packets. Second, transmitting l packets from a single queue allows device driver optimizations to transmit/fetch several packets in one DMA transfer.

For $L\text{-}DS_i$, we vary l for four values of 15, 20 and 25 and 100. The transmission at each forwarder stops when the sum of packet lengths crosses $l * 64$ bytes. Note that in this technique, l can be varied more widely. However, the value of l should not cause some queues to drain completely before the other queues have finished transmission to reduce non-work conserving behaviour and reduce short-term fairness imbalance. In Table IV, we report the performance of $L\text{-}DS_i$. We observe that $L\text{-}DS_i$ performs better than $M\text{-}DS_i$ for the same values of l. The $L\text{-}DS_i$ is thus able to avoid the issues limiting the performance of $M\text{-}DS_i$. We therefore observe that in $L\text{-}DS_i$, a STSDR forwarder is able to achieve almost the same throughput as the STSDR forwarders with no WFQ functionality.

We now address the issue of fairness for $M\text{-}DS_i$ and $L\text{-}DS_i$. $M\text{-}DS_i$ is a direct extension of $S\text{-}DS_i$. Hence, the fairness will remain identical to that of $S\text{-}DS_i$ which was presented in Table I. Let us now consider $L\text{-}DS_i$.

Table V reports the number of bytes transmitted by the $L\text{-}DS_i$ and a centralized WFQ scheduler on OL_1. We retain the same traces (as used for $S\text{-}DS_i$) for variable packet lengths. We report the results for 25 packets transmitted per cycle. For fixed length packets, the fairness declines considerably compared to the single packet per cycle implementation. This is due to the fact that in every cycle, 25 packets get transmitted from a class queue before another class may catch up (recall that all class queues have same weight). The worst case difference in the number of bytes transmitted for fixed length packets is about 0.001%. For variable length packets, the worst case difference is about 0.7%. We thus conclude that lazy scheduling performs a tradeoff for higher throughput performance in exchange for a slight decrease in the fairness of bandwidth allocation. We have thus demonstrated a distributed scheduler implementation ($L\text{-}DS_i$) on a distributed OSR that offers both high performance and almost fair bandwidth allocation.

Packet Size	Class #1		Class #2		Class #3	
	Centralized	Distributed	Centralized	Distributed	Centralized	Distributed
64 bytes	21,334,400	21,339,200	21,334,400	21,313,600	21,334,400	21,347,200
Variable	246,645,653	246,181,428	246,645,603	246,846,529	246,645,515	244,909,002

TABLE V

FAIRNESS COMPARISON BETWEEN CENTRALIZED AND $L\text{-}DS_i$. THE BYTES TRANSMITTED PER CLASS AT THE END OF $\sim$1 MILLION ROUNDS ARE NOTED. ALL CLASSES HAVE SAME WEIGHT.

VI. DISCUSSIONS AND CONCLUSION

In this paper we first detailed the difficulties that arise when the WFQ functionality designed for a centralized system is implemented over a distributed system. We also gave solutions to each of the issues and implemented a distributed scheduler for WFQ over three forwarders for three classes of traffic. We made some comparisons between the WFQ functionality of a centralized and our distributed solution. We analyzed the performance issues related to our proposed implementation and demonstrated further improvements by amortizing communication costs.

The focus of this paper is not to provide the best mechanism or implementation for a distributed WFQ. Hence, there may be ways to improve the implementation. Techniques that improve time synchronization between the STSDR nodes will hasten synchronization of the globally distributed state information. Out of band distribution of state information can allow pipelining of certain functions. The $M\text{-}DS_i$ approach can be extended further by using stale information for p additional rounds. This extension may be acceptable as information for only one of the N queues needs to be updated in each forwarder per round. For the rest of the queues, the information does not change. Using Jumbo Ethernet frames for DS_i packets can allow further improvement in performance of $M\text{-}DS_i$.

ACKNOWLEDGEMENT

This work was supported in part by the Bharti Centre for Communication at Indian Institute of Technology Bombay.

APPENDIX

For the distributed router STSDR, four PCs each having a Intel C2D 2.1GHz and DG965RYCK motherboard with onboard GbE NICs and a Nortel 3510-24T Gigabit switch was used. The operating system details are as follows. *Linux Kernel Version* : 2.6.16-13. *Click Version*: 1.5.0 for packet generation and reception. *Perl*: Perl scripts during test execution and post processing. The trace files for the fairness comparison was taken from an intercity router of an ISP from Italy in early 2008 and contained about 1.54 million packets in each trace.

REFERENCES

[1] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High-speed switch scheduling for local-area networks," *ACM Transactions on Computer Systems*, vol. 11, pp. 319–352, November 1993.

[2] N. Athaide, A. Khan, D. Manjunath, and A. Sahoo, "Trie Partitioning in Distributed PC Based Routers," in *Proceedings of the First International Communication Systems and Networks and Workshops(COMSNETS)*, 5-10 January 2009, pp. 1–10.

[3] A. Bianco, R. Birke, L. Giraudo, F. Marenko, M. Mellia, A. Khan, and D. Manjunath, "Control and Management Plane in a Multi-stage Software Router Architecture," in *Proceedings of the International Conference on High Performance Switching and Routing*, 15-17 May 2008, pp. 235–240.

[4] A. Bianco, J. M. Finochietto, G. Galante, M. Mellia, and F. Neri, "Open-Source PC-Based Software Routers: A Viable Approach to High-Performance Packet Switching," in *Proceedings of the 3rd International Workshop on QoS in Multiservice IP Networks (QoSIP)*, 2-4 February 2005, pp. 353–366.

[5] A. Bianco, J. M. Finochietto, M. Mellia, , F. Neri, and G. Galante, "Multistage Switching Architectures for Software Routers," *IEEE Network*, vol. 21, no. 4, pp. 15–21, 2007.

[6] R. Bolla and R. Bruschi, "A high-end Linux based Open Router for IP QoS networks: tuning and performance analysis," in *Proceedings of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements*, March 2005, pp. 203–214.

[7] R. Bolla, R. Bruschi, A. Ranieri, and G. Traverso, *Grid Enabled Remote Instrumentation*. Springer, 2008, ch. Analyzing and Optimizing the Linux Networking Stack, pp. 187–199.

[8] S. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE Journal on Selected Areas in Communication*, pp. 1030–1039, 1999.

[9] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, 2009, pp. 15–28.

[10] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a GPU-Accelerated Software Router," in *Proceedings of the 2010 ACM SIGCOMM 2010 Conference*, August 2010, pp. 195–206.

[11] A. Khan, R. Birke, D. Manjunath, A. Sahoo, and A. Bianco, "Distributed PC based routers: Bottleneck analysis and architecture proposal," in *Proceedings of High Performance Switching and Routing 2008*, May 2008.

[12] J. Kleinberg and E. Tardos, *Algorithm Design*. India: Pearson Education, 2005, ch. Introduction: Some representative problems, pp. 1–12.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[14] N. McKeown, M. Izzard, A.Mekkittikul, W. Ellersick, and M. Horowitz, "Tiny Tera: a packet switch core," *IEEE Micro*, vol. 17, no. 1, pp. 26–33, 1997.

[15] C. . Router, "Technical Specifications for the Cisco 7200," 7 August 2009. [Online]. Available: http://www.cisco.com/en/US/products/hw/routers/ps341/products_tech_note09186a0080094ea3.shtml

[16] D. C. Stephens and H. Zhang, "Implementing distributed packet fair queueing in a scalable switch architecture," in *Proceedings of INFOCOM 1998. The Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies.*, 1998, pp. 282–290.

[17] I. Stoica and H. Zhang, "Exact emulation of an output queueing switch by a combined input output queueing switch," in *Proceedings of IEEE/IFIP Sixth International Workshop on Quality of Service (IWQoS)*, 1998, pp. 218–224.

[18] N. Vaidya, A. Dugar, S. Gupta, and P. Bahl, "Distributed Fair Scheduling in a Wireless Lan," *IEEE Transactions on Mobile Computing*, vol. 4, pp. 616–629, 2005.