PROJECT REPORT

Modified Weighted Iterative SLIP Algorithm

Submitted By
Muhammad Hassan (Group 8)
03030025@lums.edu.pk

Submitted To:
Dr. Zartash Afzal Uzmi
(zartash@lums.edu.pk)
# Table of Contents

1. **INTRODUCTION** .................................................................................................................. 4

2. **PARALLEL ITERATIVE MATCHING** ................................................................................ 5
   A. Parallel-Iterative Matching ............................................................................................... 5
   B. Parallel Round-Robin Iterative Matching .......................................................................... 5
   C. Iterative-SLIP Matching .................................................................................................. 6

3. **MODIFIED WEIGHTED ITERATIVE-SLIP** ..................................................................... 8

4. **PERFORMANCE COMPARISONS** .................................................................................. 9
   A. Under Bernoulli Arrivals ................................................................................................. 13
   B. Under Bursty Arrivals ..................................................................................................... 13
   C. As a Function of Number of Iterations ............................................................................ 15

5. **NETWORK SIMULATOR** ................................................................................................. 16
   A. PIM Implementation ....................................................................................................... 16
   B. PRM Implementation ....................................................................................................... 18
   C. ISLIP Implementation ..................................................................................................... 21
   D. MW-ISLIP Implementation ............................................................................................ 23

6. **CONCLUSION** .............................................................................................................. 26
Abstract – With the advancements in the current network technologies, it is now possible to build high bandwidth connection networks. The working of switches is very critical to the performance of the network. The switch throughput in return depends upon type of queuing and scheduling algorithms used in it. Iterative – SLIP algorithm is an improvement over PRM matching algorithm. But simulations have shown that above 60% arrival rate the performance of iterative – SLIP degrades sharply. Secondly, iterative – ISLIP is non-responsive to traffic bursts and non-uniform traffic patterns. Unfortunately, this is the traffic pattern in real networks. This paper has proposed a slightly modified version of iterative – OCF algorithm, Modified Weighted Iterative SLIP. The new algorithm replaces the randomization adopted in iterative – OCF for breaking ties with round-robin approach of iterative – SLIP. Like iterative – SLIP the new algorithm is designed to be readily implemented in hardware. Simulations have shown that the new algorithm performs remarkably well against non-uniform traffic. Simulations have also shown the throughput of this new algorithm with single iteration is significantly improved over iterative – SLIP.
1. INTRODUCTION

An increasing number of high performance internetworking protocol routers, LAN and ATM switches are using input-queued switches, largely because the switching fabric and the buffer are not required to be speedup. The scheduling algorithms are very critical for high performance. It is well known that if simple FIFO input queues are used to hold packets then, even under benign Bernoulli arrival pattern, the achievable bandwidth is limited to approximately 58.6 % due to head-of-line (HOL) blocking. This is clearly not acceptable. The HOL blocking can be avoided by using Virtual Output Queuing (VOQ) and it is also possible to get 100 % throughput incase a suitable scheduling algorithm is used with VOQs. In [2], couple of techniques, Longest Queue First (LQF) and Longest Cell First (LCF) are presented and proved to give 100 % throughput.

![Fig 1 An input queued switch with VOQ](image)

The iterative-SLIP algorithm presented in [3] gives an efficient and easier to implement scheduling algorithm. The iterative-SLIP gives 100% throughput with independent and identically distributed (i.i.d.) uniform arrival patterns. But in case of bursty and non-uniform packet arrivals, iterative-SLIP can lead to longer queue delays. Unfortunately, this is the traffic pattern in real networks making this a major drawback of iterative-SLIP. The transport protocols can easily confuse longer delays as packet lost thus resending the packets again, causing traffic congestion.

The longer queue delays can be avoided by using OCF weight matching algorithm [2]. The OCF algorithm gives preferential services to the cells that have been queued for a long time thus reducing variance in cell latency. However, the maximum weight matching algorithms like LQF and OCF are not readily implemented in hardware. The theoretical running time of these algorithms is $O(N^2 \log_2 N)$ which is not very optimal.

An iterative approximation of OCF (iterative-OCF) is suggested by [8]. The algorithm is designed to be readily implemented in hardware and to find a maximal weight matching rather quickly. However there is still one bottleneck, the ties are broken randomly.

In this paper, a new Modified Weighted Iterative-SLIP (MW-ISLIP) is proposed. Through simulations it is shown that performance of MW-ISLIP algorithm is far better than other earlier parallel iterative algorithms, even with single iteration.
2. PARALLEL ITERATIVE MATCHING

There are various three phased parallel iterative matching algorithms earlier proposed. Such algorithms do not require coordination among output ports, therefore are relatively fast. Each input port sends request to every output port, for which it has any packet to send. The arbiters at each input and output then independently decide through some mechanism, whether to make a connection or not. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of iteration are eligible for matching in the next.

A. Parallel- Iterative Matching

Parallel Iterative Matching (PIM) was developed by DEC Systems Research Center. It consists of random-access input buffers (or equivalently multiple queues per input) with the following three phase parallel randomized scheduling algorithm operating independently at each input and output ports.

Step 1: Request. Each unmatched input sends a request to every output for which it has a queued packet.

Step 2: Grant. If an unmatched input receives any request, it chooses one randomly to grant. The granted input is notified.

Step 3: Accept. If the input receives at least one grant, it chooses one by randomly selecting one of them, thus completing the connection.

Randomization helps in eliminating at least one third of the remaining possible connections at single iteration, thus the algorithm will converge to a maximal matching in O (log N) iterations. The algorithm gives better performance in the sense that no input queue is starved, no memory state is required to be maintained and output arbiters remain relatively desynchronized. However, randomization comes with its cost i.e., it is fairly difficult and expensive to implement randomization in hardware. Secondly, it performs poorly with a single iteration for large N with maximum throughput of 63 %.

B. Parallel Round-Robin Iterative Matching

The basic round-robin (PRM) algorithm in [8] is designed to overcome the problems of complexity and unfairness in PIM. Rather than arbitrate randomly, the selection at input and output arbiters is done according to a deterministic round-robin fashion. The round-robin pointers are updated after each grant (at each output) and after each accept (at each input). The performance of PRM is slightly better than PIM with less than 80% load. However, the performance degrades sharply as the arrival rate increases above 80%. The reason for poor performance is that the PRM output-pointers get synchronized very soon and starts moving in locked-steps, thus are only able to service few inputs per time slot.
C. Iterative-SLIP Matching

The iterative-SLIP proposed in [3] is an improvement over PRM. The main aim is to prevent synchronization in output pointers. It consists of following three steps:

**Step 1: Request.** Each input sends a request to every output for which it has a queued packet.

**Step 2: Grant.** If an output receives any request it chooses the one that appears next in a fixed round robin schedule starting from a highest priority element. The output notifies each input whether or not its request is granted.

**Step 3: Accept.** If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The input pointer $a_i$ is incremented to one location beyond the accepted output. The respective output pointer $g_j$ is also incremented to one location beyond the accepted input.

![Diagram of iterative-SLIP matching algorithm](image)

Although through selective increments, iterative SLIP avoids the synchronization problem of PRM. There is another inherent flaw in both matching algorithms, PRM and iterative-SLIP, that comes with the deterministic round-robin scheme. This is the poor
response to non-uniform arrivals. Considering that real network traffic is non-uniform in nature, the flaw becomes very critical. Whenever there are bursts in network traffic, the arrivals are non-uniform; the fair scheduling priority scheme leads to longer delays for packets in a congested queue. The stability of these algorithms is inversely proportional to the arrival rates of packets. Even smaller bursts can have huge impacts on the queue sizes and cell latency for the particular input at higher arrival rates.

This phenomenon is more apparent as the arrival rate reaches close to 100%. Even with Bernoulli arrivals, and single iteration, the throughput of PRM comes down to 30%. Where as the throughput of ISLIP come down to 40% approximately in the same scenario. Although the throughput can be increased by increasing the number of iterations, iterations have their own costs in terms of time and computing resources, therefore it is quite essential that the performance of matching algorithms should not be degraded significantly when run with only single iteration.
3. MODIFIED WEIGHTED ITERATIVE- SLIP

The Modified Weighted iterative- SLIP (MW-ISLIP) is an approximation of OCF matching [2] with the iterative nature of i – SLIP algorithm.

WM – ISLIP is more adaptable to traffic bursts. This is achieved by considering the packet waiting times in addition to round-robin arbitration scheme of iterative – SLIP. The proposed algorithm has the following three steps:

**Step 1: Request.** Each input sends a weighted request to every output for which it has a queued packet. The weight is set equal to the corresponding packet wait time in the input queue.

**Step 2: Grant.** If an output receives any request it chooses the one that has the maximum weight attached to it. In case multiple requests have same maximum weight, output arbiter chooses the one that appears next in a fixed round robin schedule starting from a highest priority element. The output notifies each input whether or not its request is granted.

**Step 3: Accept.** If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The input pointer is incremented to one location beyond the accepted output. The respective output pointer is also incremented to one location beyond the accepted input.

The proposed algorithm is different from iterative – OCF as it uses deterministic round-robin scheme instead of complex randomization to break ties. The proposed algorithm gives priority to packets with relatively greater wait time. Therefore in the case of bursty arrivals, the effected input port will get its packet granted every second time slot. Instead of every Nth time slot in iterative – SLIP.
4. PERFORMANCE COMPARISONS

This section does a comparison between iterative – SLIP and MW – ISLIP matching algorithms. The comparison assumes the number of iterations to be one for Bernoulli and Bursty arrivals unless otherwise stated. The number of timeslots is assumed to be 1000 unless otherwise stated.

The Average queue sizes of PRM are highest, ISLIP has the next highest, than PIM and MW – ISLIP has the least queue size among all. Using average wait time as weight has its effect on the queue size of MW – ISLIP (Fig 3, 4 and 5).

Fig 3 Average Queue Size with single iteration

Fig 4 Average Queue Size with 2 iterations
The average packet latency for PIM is highest, than MW – ISLIP, PRM and ISLIP has the lowest cell latency. The results for 1000 timeslots are misleading. As the latency is calculated for the arrived packets only, the number of arrived packets is far less in PRM and ISLIP, resulting in lower latency. A close to correct calculation can be obtained by running simulations for one million packets (Fig 6, 7 and 8).
The throughput of MW – ISLIP is highest among all. The throughput of PIM is second highest, than is ISLIP and PRM has the lowest throughput (Fig 9, 10 and 11).
Fig 9 Throughput with single iteration

Fig 10 Throughput with 2 iterations
**A. Under Bernoulli Arrivals**

The Bernoulli traffic arrivals can be divided into two classes on the basis of their arrival rate.

For the arrival rate less than 60%, known as benign Bernoulli arrival, the performance of MW – ISLIP is equal to iterative – SLIP matching algorithm. Both touch nearly 100% throughput level. The packet latency and mean queue size of VOQs is close to zero in both cases.

However, as the arrival rate goes above 60% the performance of iterative – SLIP sharply degrades. In fact as the arrival rate comes closer to 100%, throughput comes down to mere 35%. This is significantly different from WM – ISLIP, which is able to maintain above 80% throughput under similar conditions. This phenomenon seems to be related to poor response of iterative – SLIP to bursty arrivals. As the arrival rate increases, there can be many small bursts, not by the packets arriving at the same input for the same output, but by the packets arriving for the same output at different inputs (bursts in a different sense). With the increase in queue sizes of various VOQs, the chances of one input getting grant by various outputs becomes greater. This decreases the throughput of ISLIP in a single iteration.

**B. Under Bursty Arrivals**

As already mentioned, iterative - SLIP works fine with benign uniform i.i.d. arrivals. When ever there are bursts in network flow, and the traffic is non-uniform, the fair scheduling priority scheme leads to longer queue delays for packets in the congested queue. This can be seen by the following example.
Consider an N by N switch. The arrival of packets is such that:

**Time slot 1**: packets for output 1 arrive at input 1 and N. The arbiter at output 1 grant input 1’s request. Input 1 accepts the grant and the output pointer 1 is incremented to 2.

**Time slot 2**: packets for output 1 arrive at input 2 and N. The arbiter at output 1 grant input 2’s request. Input 2 accepts the grant and the output pointer 1 is incremented to 3.

…

**Time slot N-1**: packets for output 1 arrive at input N-1 and N. The arbiter at output 1 grant input (N-1)’s request. Input N-1 accepts the grant and the output pointer 1 is incremented to N.

**Time slot N**: the first packet from VOQ at input N is finally served.

Consider that by this time VOQ at input N has N packets for output 1 and the oldest packet is N-1 time slots old. Now if the same pattern continues for some time t, the queue size will increase in the multiple of N per iteration. This will increase the packet delay to a significant level such that the network transport protocol can assume the packet as lost. This will further induced the transport layer to resend the whole message thus increasing the network congestion and in return creating another arrival burst.

The average latency imposed by iterative – SLIP scheduler is a function of number of ports. The latency increases with the increase in number of ports. In other words, the performance degrades as the switch size increases. The fair scheduling scheme is critical as bursty arrivals are quite normal in real networks.

Consider the same example of bursty arrival in N by N switch:

**Time slot 1**: packets for output 1 arrive at input 1 and N. As the wait time for both packets is 1. There is a tie, which will be solved through randomly selecting one of them. Consider that the arbiter at output 1 grant input 1’s request. Input 1 accepts the grant as there are no other options for input 1.

**Time slot 2**: packets for output 1 arrive at input 2 and N. The input 2 and N request for output 1 with wait time of one and two respectively. The arbiter at output 1 grant input N’s request. Input
N accepts the grant. This is different from original iSLIP where the input N request was not granted at this stage.

**Time slot 3**: packets for output 1 arrive at input 3 and N. The input 2, 3 and N request for output 1 with wait time of two, one and two. There is again a tie between input 2 and N. Lets assume that input 2 is granted the service. Input 2 accepts the grant.

**Time slot 4**: packets for output 1 arrive at input 4 and N. The input 3, 4 and N request for output 1 with wait time of two, one and three. The arbiter at output 1 grant input N’s request. Input N accepts the grant. This is different from original iterative – SLIP where the input N request was not granted at this stage.

…

Note that there is difference is how the requests of input N for output 1 are granted. The service is granted to VOQ 1 of input 1 on every second time slot thus greatly reducing the latency of its packets. In return, this reduction in latency is causing a slight delay in the arrivals at other input ports for output 1 packets but this delay is far lesser than the one caused to input N packets. The actual throughput remains the same.

C. As a function of Number of Iterations

Simulations have shown that with single iteration, the performance of iterative – SLIP is significantly less than MW – ISLIP remains close to 100% throughput for fewer than 60% arrival rate. But as the arrival rate goes above 60%, the throughput of iterative – SLIP degrades sharply. It is already mentioned that with 100 % arrival rate the iterative – ISLIP is able to touch 40 % throughput only, whereas the proposed algorithm registers above 80% throughput, a significant difference by any standard.

But with a subsequent iteration, the throughput of iterative – ISLIP comes closer to the proposed algorithm results. Up to 80% arrival rate, the throughput remains close to 100%. Although again with 100 % arrival rate, the throughput of iterative – SLIP comes down to 70 % whereas the proposed algorithm gives around 90 % throughput.
5. NETWORK SIMULATOR

Four iterative matching algorithms are implemented and integrated in Network Switch Simulator [9]. These algorithms are PIM, PRM, iterative – SLIP and WM – ISLIP.

For each algorithm, two classes are implemented: one for Input and other for scheduler (algorithm). The implementation is in java. The implementations details are as follows:

A. PIM Implementation

```java
package edu.lums.s03.cs678.input;

import sun.misc.Queue;
import java.util.Hashtable;
import java.util.List;
import java.util.Random;
import edu.lums.s03.cs678.packet.Packet;

public class PIMInput extends Input{

    protected void rearrangeWeights() throws Exception{
        Queue tmp= new Queue();
        while (!queue.isEmpty()){
            Packet p = (Packet)queue.dequeue();
            p.incrementWeight();
            tmp.enqueue(p);
        }
        queue= tmp;
    }

    public Hashtable requestOutPorts() throws Exception {
```

Hashtable req = new Hashtable();
Queue tmp = new Queue();
while (!queue.isEmpty())
{
    Packet p = (Packet) queue.dequeue();
    req.put(new Integer(p.getTargetHost()), new Integer(p.getWaitingTime()));
    tmp.enqueue(p);
}
queue = tmp;

return req;
}

protected int selectOutputGrant(List grantedRequests) {
    Random random = new Random();
    int t = random.nextInt(grantedRequests.size());
    //System.out.println("the value of t is " + t + " against size " +
    grantedRequests.size());
    return ((Integer) grantedRequests.get(t)).intValue();
}
*/

* PIMScheduler.java
*
* Created on May 4, 2004, 10:30 PM
*/

package edu.lums.s03.cs678.input;

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import edu.lums.s03.cs678.packet.Packet;

package edu.lums.s03.cs678.input;

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import edu.lums.s03.cs678.packet.Packet;

/**
 * @author 03030035
 */
public class PIMScheduler extends PacketScheduler{
    /**
     * Creates a new instance of PIMScheduler */
    public PIMScheduler() {
        super();
    }
public Hashtable calculateMatchings()
{
    Hashtable outPutGrants= new Hashtable();
    Enumeration enumRequests= outPortRequests.keys();
    Random random= new
    Random(java.util.Calendar.getInstance().getTimeInMillis());

    while (enumRequests.hasMoreElements())
    {
        Object ki= enumRequests.nextElement();
        Iterator requestingPorts= ((List) outPortRequests.get(ki)).iterator();
        int rand= random.nextInt(((List) outPortRequests.get(ki)).size());
        rand++;

        int current= 1;
        Request selectedOne= null;
        while (requestingPorts.hasNext())
        {
            Request r= (Request) requestingPorts.next();
            if (rand == current++) selectedOne= r;
        }
        if (null!= selectedOne)
            outPutGrants.put(ki, new Integer(selectedOne.getInPort()));
    }

    Enumeration outPutGrantsKeys= outPutGrants.keys();
    while (outPutGrantsKeys.hasMoreElements())
    {
        Integer outPort= (Integer) outPutGrantsKeys.nextElement();
        Integer inputPort= (Integer)outPutGrants.get(outPort);
        removeRequest(inputPort.intValue(), outPort.intValue());
    }*/
  return outPutGrants;
}

B. **PRM Implementation**

/*
 * BasicRRInput.java
 * *
 * Created on May 5, 2004, 1:33 AM
 */

package edu.lums.s03.cs678.input;
import sun.misc.Queue;
import java.util.Hashtable;
import java.util.List;
import java.util.Collections;
/**
 * @author 03030025
 */
public class PRMInput extends PIMInput{
    private int rrPointer;
    /** Creates a new instance of BasicRRInput */
    public PRMInput() {
        rrPointer= 0;
    }
    protected int selectOutputGrant(List grantedRequests) {
        int t=0;
        Collections.sort(grantedRequests);
        for (int i=0; i < grantedRequests.size(); i++)
        {
            t= ((Integer)grantedRequests.get(i)).intValue();
            if (t >= rrPointer)
            {
                rrPointer= t+1;
                return t;
            }
        }
        t= ((Integer)grantedRequests.get(0)).intValue();
        rrPointer= t;
        //System.out.println(" the value of t is " + t + " against size " + grantedRequests.size());
        return t;
    }
}

package edu.lums.s03.cs678.input;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Random;
import java.util.ArrayList;
import java.util.Collections;
import edu.lums.s03.cs678.packet.Packet;
import java.util.Comparator;

public class PRMScheduler extends PacketScheduler{
    protected ArrayList outPortPointers;
    /** Creates a new instance of BasicRRScheduler */
    public PRMScheduler() {
        super ();
    }

    public void setSize(int s) {
        size = s;
        outPortPointers= new ArrayList(size);
        for (int i=0; i < size; i++)
        {
            outPortPointers.add( new Integer(1));
        }
    }

    public Hashtable calculateMatchings()
    {
        Hashtable outPutGrants= new Hashtable();
        Enumeration enumRequests= outPortRequests.keys();
        //Random random= new Random(java.util.Calendar.getInstance().getTimeInMillis());
        while (enumRequests.hasMoreElements())
        {
            Integer ki= (Integer)enumRequests.nextElement();
            List listRqPorts= (List) outPortRequests.get(ki);
            ArrayList sortedList= new ArrayList();
            for (int i=0; i < listRqPorts.size(); i++)
            {
                sortedList.add(i, new Integer(((Request)listRqPorts.get(i)).getInPort()));
            }
            Collections.sort(sortedList);
            Iterator requestingPorts= sortedList.iterator(); 
            int rand= ((Integer)outPortPointers.get(ki.intValue()-1)).intValue();//random.nextInt(((List) outPortRequests.get(ki)).size());
            //rand++;
        }
    }
}
Integer selectedOne= null;
while (requestingPorts.hasNext())
{
    Integer r= (Integer) requestingPorts.next();
    if (r.intValue() >= rand)
    {
        selectedOne= r;
        incrementPointer(ki.intValue(), r.intValue()+1);
    }
}

if (null== selectedOne)
{
    selectedOne= (Integer)sortedList.get(0);
    incrementPointer(ki.intValue(), selectedOne.intValue()+1);
}
outPutGrants.put(ki, selectedOne);

/*
Enumeration outPutGrantsKeys= outPutGrants.keys();
while (outPutGrantsKeys.hasMoreElements())
{
    Integer outPort= (Integer) outPutGrantsKeys.nextElement();
    Integer inputPort= (Integer)outPutGrants.get(outPort);
    removeRequest(inputPort.intValue(), outPort.intValue());
}*/
return outPutGrants;
}
public void incrementPointer(int o, int i)
{
    outPortPointers.set(o-1, new Integer(i));
}

C. ISLIP Implementation

/*
* ISLIPInput.java
* *
* Created on May 3, 2004, 8:17 PM
* /
package edu.lums.s03.cs678.input;
import edu.lums.s03.cs678.packet.Packet;
/**
 * @author 03030025
 */
public class ISLIPInput extends PRMInput{

    /** Creates a new instance of ISLIPInput */
    public ISLIPInput() {
    }

}

package edu.lums.s03.cs678.input;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import java.util.ArrayList;
import java.util.Collections;
import edu.lums.s03.cs678.packet.Packet;
import java.util.Comparator;
/**
 * @author 03030025
 */
public class ISLIPScheduler extends PRMScheduler{

    /** Creates a new instance of ISLIPScheduler */
    public ISLIPScheduler() {
        super();
    }

    public void selectedMatches(Hashtable outPutInputMatches)
    {
        Enumeration enumKeys = outPutInputMatches.keys();
        while (enumKeys.hasMoreElements())
        {
            // Code continues here...
        }
    }
}
D. MW-ISLIP Implementation

/*
 * MWISLIPInput.java
 *
 * Created on May 9, 2004, 11:30 AM
 */

package edu.lums.s03.cs678.input;

/**
 * @author 03030025
 */
public class MWISLIPInput extends WISLIPInput {

    /** Creates a new instance of MWISLIPInput */
    public MWISLIPInput() {
    }
}

/*
 * MWISLIPScheduler.java
 *
 * Created on May 9, 2004, 11:30 AM
 */

package edu.lums.s03.cs678.input;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
/**
 * @author 03030025
 */
public class MWISLIPScheduler extends WISLIPScheduler{

/** Creates a new instance of MWISLIPScheduler */
public MWISLIPScheduler() {
}

public Hashtable calculateMatchings()
{
    Hashtable outPutGrants= new Hashtable();
    Enumeration enumRequests= outPortRequests.keys();
    //Random random= new
    Random(java.util.Calendar.getInstance().getTimeInMillis());

    while (enumRequests.hasMoreElements())
    {
        Integer ki= (Integer)enumRequests.nextElement();
        List listRqPorts= (List) outPortRequests.get(ki);
        Collections.sort(listRqPorts, new RequestComparator());
        Iterator requestingPorts= listRqPorts.iterator();
        int rand= ((Integer)outPortPointers.get(ki.intValue()-1)).intValue();//random.nextInt(((List) outPortRequests.get(ki)).size());
        //rand++;

        Request selectedOne= null;
        if (requestingPorts.hasNext())
            selectedOne= (Request) requestingPorts.next();
        if (selectedOne.getInPort() < rand)
            while (requestingPorts.hasNext())
            {
                Request r= (Request) requestingPorts.next();
                if (selectedOne.getWeight() == r.getWeight() && r.getInPort() >= rand)
                {
                    selectedOne= r;
                    incrementPointer(ki.intValue(), r.getInPort());
                    break;
                }else if (selectedOne.getWeight() != r.getWeight()) break;
            } //selectedOne= r;
        if (null!= selectedOne)
outPutGrants.put(ki, new Integer(selectedOne.getInPort()));

*/
Enumeration outPutGrantsKeys= outPutGrants.keys();
while (outPutGrantsKeys.hasMoreElements())
{
    Integer outPort= (Integer) outPutGrantsKeys.nextElement();
    Integer inputPort= (Integer)outPutGrants.get(outPort);
    removeRequest(inputPort.intValue(), outPort.intValue());
}*/
return outPutGrants;
}
6. CONCLUSION

The paper has presented a Modified Weighted Iterative SLIP algorithm to give an even better performance than iterative – SLIP. The performance of iterative–SLIP, PIM and PRM is simulated against different arrival rates under Bernoulli traffic pattern. It is shown that simply desynchronizing the output grant pointers is not enough to get significant throughput in a single iteration for above 60% arrival rate.

The proposed algorithm is different from iterative – ISLIP, in the 2\textsuperscript{nd} step output chooses the request with maximum weight. The ties are broken by deterministic round-robin scheme. The input and output pointers are updated in a similar way to iterative – ISLIP.

As the real network traffic is non-uniform, the iterative – ISLIP inability to handle bursts is critical. However if the algorithm is slightly modified to take wait times of packets in combination of the round-robin approach to break ties, the algorithm become flexible enough to absorb packet bursts. Simulations are done for Bernoulli traffic with 100 % weights to imitate small bursts in traffic. The results have shown a significant drop in the throughput incase of iterative – ISLIP, where as the throughput of MW – ISLIP remains above 80% with a single iteration.

However it is also observed that as we increase the number of iterations, both algorithms start to converge to around 100 % throughput.
REFERENCES


