OWLBIT: Orchestrating Wireless Transmissions for Launching Big Data Platforms in an Internet of Things Environment

Nam Son Nguyen*, Teng Wang*, Tengpeng Li*, Xiaoqian Zhang*, Bo Sheng*, Ningfang Mi†, Bin Zhao‡
*Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
† Department of Electrical and Computer Engineering, Northeastern University , 360 Huntington Ave., Boston, MA 02115
‡ School of Computer Science and Technology, Nanjing Normal University, Nanjing, China

***Abstract*—The emergence of Edge Computing and the success of Internet of Thing and (IoT) has tremendously changed the way we think about data computing. With edge devices changing from data producer to both data producer and consumer, the chance for processing large data sets with Big Data on a cloud of IoT devices is more realistic. In Big Data systems such as Hadoop-Yarn, Spark, Pig, etc., the shuffling stage is by far the most dominant source of network traffic. Unreliable performance of network will greatly impact the shuffling process. Since IoT devices mostly rely on wireless network based on 802.11, providing proper throughput to big data computing system is an important challenge that needs to be address. In this paper, we argue that a cluster of IoT computers can support big data by considering the information fed by the big data applications. We propose a cross-layer framework that uses the application layer information to guide the packet scheduling at the link layer. We implement our system as an extension module in Hadoop-Yarn system. The experimental evaluation shows significant performance improvement.**

I. Introduction

In recent years, big data processing applications have become very popular in various fields helping extract useful information from a large set of data. One important category of big data applications is related to our physical world, where electronic devices are deployed in our surrounding environments and supply data for processing. The emerging IoT (Internet of Things) devices are representative devices in this domain that could measure different types of data and communicate with other devices. These IoT devices are often massively deployed and continuously collect data yield a large volume of data set that would need a big data platform to process.

In this paper, we present a middleware framework to support the deployment of big data platforms over distributed Internet of Things (IoT) devices. It is motivated by the fact that nowadays the computation ability of IoT devices have increased with the advance of hardware. In the traditional view of data-oriented applications, IoT devices, considered inexpensive but computationally weak, mainly play the role of data provider while the data processing is conducted at cloud-side servers or more recently introduced edge computing sites. This traditional approach, however, causes heavy network traffics that collect raw data from IoT devices, and computation burden on the centralized processing sites.

In fact, with the advance of the hardware, recent IoT devices such as Raspberry Pi have been equipped with adequate RAMs and strong CPUs that are capable of processing a certain amount of data in the applications. The participation of IoT devices in big data processing can significantly save the data transfer traffic and reduce the server load. It is practically possible to migrate data processing tasks from the centralized servers to massive IoT devices to avoid data transfer traffic and better serve the applications. On the other hand, new big data platforms such as Hadoop and Spark (see Fig. 1) are designed to process a large volume of data in a distributed manner where each node in a cluster handles a piece of data and the results are merged and further processed in the next stage. Such platforms are feasible to be deployed over a cluster of IoT devices to help process various data.

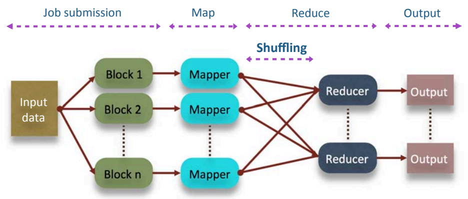


Fig. 1. MapReduce process

This paper explores the integration of big data platforms (e.g., Hadoop) and small IoT devices, reveals the new issues in such systems, and develops an efficient supporting layer to improve the performance. To the best of our knowledge, this is the first system work that integrates big data processing framework with IoT devices.

II. Related Work

In this section, we briefly cover network flow scheduling approaches for big data framework and wireless interference awareness that are related to our work.

Network flow scheduling for big data framework. Several previous efforts have tried to optimize network traffic in Big Data MapReduce clusters, including *Delay Scheduling* [1] and *Quincy* [2]. These works target on optimizing map-input data locality for the purpose of reducing network contentions. However they didn't take in to account the shuffle stages which, by observations in [3] [4], sometimes intermediate data (map-output data) size can be as large as input size e.g. sort or even larger for k-means clustering [5]. In contrary, *CoGRS* [5] focuses on schedule placement of reduce tasks or *ShuffleWatcher* [6] localizes the shuffle by scheduling both maps and reducers on the same rack. Without a doubt the contributions of above researches, none of them can be applied to reduce wireless network congestions in Big Data on IoT clusters. On the other hand, integrating these techniques will benefit our solution.

Wireless interference. Centralized scheduling schemes

which focus on reducing wasted airtime incurred by random back-off DCF (Distributed Coordination Function), such as the state of art *Centaur* [7], *CO-Fi* [8] and *Shuffle* [9]. Such works use a centralized scheduler to schedule down-link traffics toward the stations through WiFi access points. In common, all are Micro-Scheduling, providing fast access to the medium that requires powerful computation as well as MAC-802.11 modifications through an upgrade firmware. Our scheduling in owlBIT integrated with Big Data Framework which often transmits data in batches. This typical scenario do not requires per packet scheduling and computation overhead. More importantly, our solution is lightweight and portable, totally lies on application level hence it is sufficient for IoT devices with a vision to Edge Computing. Oppose to Micro scheduling, *Cooperative Packet Scheduling Pipeline* [10] offers a closer approach to our work. The authors provide a Macro Scheduling that schedules sets of packets through a $2\frac{1}{2}$-stage pipeline, an extension layer between Network and MAC layers. However, this approach requires driver modifications to make necessary calls to the extension layer. Moreover, our work aims to an universal solution that can be integrated into any commodity 802.11 wireless devices.
Beside the researches previously mentioned, many proposals [11] [12] [13] [14] try to alleviate another method that defined in the 802.11 standard, the Point Coordination Function (PCF). In [11] the authors discussed two new draft modes of operation, EDCF (Enhanced DCF) and HCF (Hybrid Coordination Function). These new MAC-802.11 modes are being defined under 802.11e support up to eight Traffic Classes of QoS.The *DiffServ MAC Extension* (DIME) [12] provides Differentiated Services for MAC-802.11 with Expedited Forward (EF) layer, they reuse the inter frames spaces of PCF which guaranties low delay traffic. Similarly, in combination with PCF, *DPCF* [13] improves the performance of ad-hoc network while *Adaptive Polling* in [14] considers traffic characteristics in polling. Unfortunately, PCF has not yet widely supported by most wireless network cards [15] while our proposed solution can implement the same functions and QoS schemes.

III. Motivation And Challenges

The main obstacle that hinders the deployment of big data platforms on IoT devices is the limited on-board resources and the low network bandwidth is the most critical problem. Compared to data center servers that are wired with 1$\sim$10Gbps ethernet links, IoT devices, however, rely on WiFi connections whose bandwidth is much lower and less stable. The network limitation significantly impacts the shuffling phase in big data jobs, where the results of a processing stage from all the nodes are re-distributed across the cluster for the next stage. In some applications, the data needed to be shuffled is large, e.g., in a typical MapReduce benchmark 'sort', the intermediate data is as large as the input data. Shuffling data over the WiFi connections could be time consuming. The problem is magnified by the fact that big data tasks are often finished in waves and start to shuffle their intermediate output data around the same time causing serious *self-interferences*. While all the nodes shuffle their data around the same time, their network traffics cause significant interferences with each other. The consequences include packet retransmissions and lowered transmitting rates by rate adaptation that further prolong the shuffling phase.

Here we use a simple experiment to demonstrate the impact of self-interferences on the system performance. We conduct a typical MapReduce job (sort) on an IoT cluster of 9 nodes, and capture all the wireless packets. All the nodes are close to each other with good wireless link qualities, and the external wireless traffic is kept to the minimal level. Fig. 2 illustrates the WiFi traffic along the job execution time. A MapReduce job includes two computing stages (Fig. 1), Map phase consisting of multiple map tasks and Reduce phase with reduce tasks. The vertical bars show the amount of bytes sent by each node per second (each color represents a different node). The horizontal blue bars indicate the execution time of all the map tasks, and the long red line is the execution of the only reduce task.

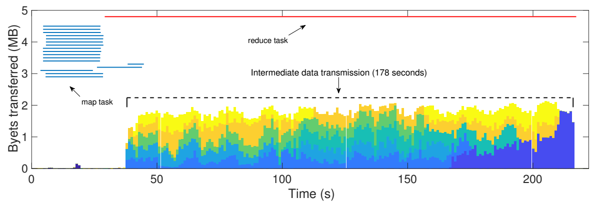


Fig. 2. An example of sorting 256M data with Hadoop running on a cluster of 9 Raspberry Pis

Obviously, shuffling intermediate data in this job takes an extremely large portion of the whole execution time. In this experiment, a total of 227K data packets are transmitted. The following three measurements show the major negative impact of the self-interferences.

Link layer retransmissions. The most direct effect is the link layer retransmission caused by signal interferences between different transmitting pairs in the cluster. In this particular test, there is only one reduce task, i.e., the receiver of the intermediate data, but multiple map task hosts may send data at the same time causing the interferences. We observe that there are 82K link layer retransmissions accounting for 36.1% of all the packets. All these retransmissions incur additional network workload and prolong the shuffling time.

Transmitting rate. Another consequence of link layer retransmissions is the lowered transmitting rate determined by the rate adaptation algorithms. As a mandatory module in 802.11 standards, rate adaptation adjusts the transmitting rate to adapt the dynamic wireless link quality. However, when a packet fails to be delivered, it is hard for the rate adaptation algorithm to distinguish if it is caused by interferences or poor link quality. As a result, the transmitting rate will be changed to a lower but more reliable one assuming the failure is caused by poor link quality. The following Table I compares the transmitting rates of retransmitted packets and successfully delivered packets. Apparently, retransmission packets tend to use lower rates that further increase the transmission time.

TX Rate (802.11g)	54M	48M	36M	24M	< 24M
Retransmission	0%	42%	31%	15%	12%
Successful delivery	71%	23%	4%	1%	1%

TABLE I. Comparison of Transmitting Rates

TCP retransmissions. Finally, intermediate data transmissions are based on TCP protocol, and severe interferences may cause

TCP retransmission when the delivery delay exceeds the time-out parameter. In this experiment, we find that there are 8.3K TCP retransmissions excluding the link layer retransmissions in the trace. This additional network overhead accounts for about 6% of the total intermediate data size.

Above all, the self-interferences caused by big data processing applications could significantly affect the link-layer transmission over the wireless links. In this paper, we aim to develop a link-layer scheduling scheme with the knowledge of the application-layer run-time status to mitigate the impact of self-interferences.

IV. Design of OWLBIT

In this paper, we develop a holistic cross-layer framework that weaves big data application run-time information with WiFi packet scheduling to improve the overall performance (we maintain our source codes at [16]). To the best of our knowledge, this is the first system work that integrates big data processing platforms with IoT devices. In this section, we present the design details of OWLBIT. We first introduce the architecture of our system including three major components. Then we present each component in a subsection.

A. Sketch of Our Solution

Our solution targets on representative data-parallel platforms deployed on a IoT cluster with a master node and multiple slave nodes. Our main idea is to adopt a centralized token-based WiFi packet scheduler to avoid self-interferences. Only the node granted with the token can transmit data during a time window. We develop an algorithm that selects the best candidate to hold the token and adjusts its window size according to the run-time information of job execution and link quality. In particular, our system consists of the following three components:

- Packet scheduling algorithm: This is the main algorithm running at the master node that determines the packet schedule of all the slave nodes. The main objective is to avoid interferences while satisfying each slave node's traffic demand. Different from prior wireless packet scheduler, our system adopts a large granularity when allocating a time slot to a slave node. Based on the information reported by the monitor module, this algorithm first estimates the time needed for each node to transfer its data. The token is granted to the node that needs the most transfer time. Its window size is determined based on the predication of the data generation so that during the time window, no other nodes will have longer transfer time than the selected node. When there are multiple active jobs, the algorithm gives a node a higher priority if it hosts the tailing tasks of a stage. Quickly transferring those tasks' intermediate data helps speed up the next stage and further release more system resources.
- Monitor module: This module is running on each slave node. It first monitors the status of the big data applications including the progress of the tasks hosted by the slave, the intermediate data that has been generated. In addition, it monitors the link quality between the slave node and all other nodes in the cluster. All the information is reported periodically to the packet scheduling algorithm.
- Packet control module: This module is deployed at slave nodes interacting with the packet scheduling algorithm on the master's side to enforce the determined packet schedule. Our module implements a virtual network interface to capture all the outgoing data packets and keeps them temporarily in a buffer. The buffered packets are sent only during the scheduled slot following a polling protocol initialized by the master node.
- Inter-process messaging module: This is an auxiliary module that supports short message communication between other modules.

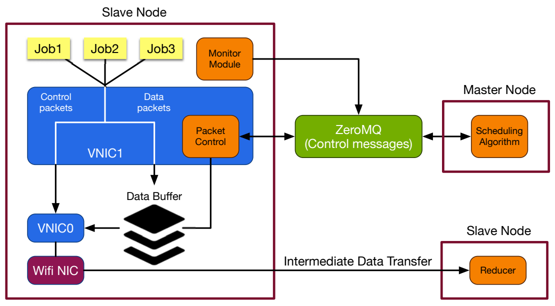


Fig. 3. Architecture of OWLBIT

B. Packet Scheduling Algorithm

In this subsection, we present the packet scheduling algorithm for shuffling the intermediate data. Our basic approach is to allow only one node to transmit the intermediate data during an assigned time window. The objective of this algorithm is to determine the transmitting node and its allocated window size. We consider a cluster of n IoT nodes, where all the nodes can directly communicate with each other via wireless links. The following information is reported by the slave nodes via the monitor module and recorded at the master node: (1) R_{ij} indicates the effective throughput for transmitting intermediate data between node i and j; (2) D_{ij} denotes the amount of data ready for shuffling at node i generated by job j; (3) d_{ij} indicates the data generation rate of job j at node i. In addition, we consider the allocated time window is $k \cdot w$, where w is a fixed epoch length configured by the system. Thus, the scheduling algorithm needs to return a node index and its window size represented by k for shuffling data.

The details are illustrated in Algorithm 1. It includes mainly two parts. The first part (lines 1–13) is to determine the node to hold the transmission token, and the second part (lines 14–24) is to derive the window size. The main metric we consider in the algorithm is called *Expected Transmission Time*, *ETT* in the algorithm. It represents the time needed for each node to transmit all the held data. In the first part, our algorithm simply selects the node whose ETT is he maximum. The algorithm enumerates all the active jobs, and adds the transmission time to ETT (lines 4–8). $\frac{D_{ij}}{R_{iu}}$ represents the transmission time per node per job, where u is the receiver node, i.e., the host node of job j's reduce tasks. For the jobs that are close to their finish, we assign a higher priority by multiplying the transmission time with a parameter $\alpha > 1$.

In the second part, our goal is to assign multiple epoch to the selected node if possible. In practice, there is always protocol overheads while switching the transmitting node. We aim to avoid unnecessary switch by estimating the ETT at the end of each epoch. The main structure is a while loop where each round k is increased by one. Line 16 calculates the ETT of the selected node at the end of the epoch. Since it has transmitted for a epoch, w is deducted from its ETT. But we also need to consider the new data generated during the epoch. In lines 17–22, the algorithm checks all other nodes' ETTs and compare with the selected node. Once the selected node is no longer the best choice, the loop terminates and the value of k is returned.

Algorithm 1: Packet Scheduling Algorithm

```
    Input  : ${R_{ij}, D_{ij}, d_{ij}, w}$
    Output: The selected node $sel$ and its assigned
            window size $k$
 1  for each node $i$ do
 2      for each active job $j$ do
 3          $u$ is the receiver of job $j$'s shuffling data
 4          if job $j$'s progress $> \tau$ then
 5              $ETT_i = ETT_i + \frac{D_{ij}}{R_{iu}} \cdot \alpha$
 6          else
 7              $ETT_i = ETT_i + \frac{D_{ij}}{R_{iu}}$
 8          end
 9      end
10      if $ETT_i > max$ then
11          $max = ETT_i \; sel = i$
12      end
13  end
14  $k = 1$
15  while true do
16      $ETT_{sel} = ETT_{sel} - w + \sum_j \frac{d_{ij} \cdot w}{R_{iu}}$
17      for each node $i \neq sel$ do
18          $ETT_i = ETT_i + \sum_j \frac{d_{ij} \cdot w}{R_{iu}}$
19          if $ETT_i > ETT_{sel}$ then
20              break
21          end
22      end
23      $k = k + 1$
24  end
25  return $sel$ and $k$
```

Implementation. We develop a protocol between the master and slave nodes to implement the scheduling algorithm. At the beginning of every epoch, e.g., w seconds, each slave node pushes its report message to master node via a reliable TCP channel. The contents of these messages are constructed by the monitor module at each slave. We maintain these messages' sizes to be as small as possible and maximize the transmitting rate. At the server/master site, a dedicated thread collects these messages from the slaves with a proper and small period of time ($t < w$). To make sure all nodes sending their reports on time, the master then broadcasts a quick signal to (1) reset the timer of every node, and (2) if a node is having problems with its clock and has not reported during the current epoch, the broadcast signal will trigger this node to immediately line up its clock as well as submit its late report. Note, at this moment, the time left in this epoch will be equal to $w - t$. The rest of the protocol is as follows:

1) The master node will check whether there is an active algorithm (Packet Scheduling Algorithm). If it finds one that has been plugged in, the algorithm will be applied and gone through to compute the size of next time-window (time slot, the time needed to transfer output data) as well as determine a next sending node. In case that no algorithm is found, Round-Robin scheduling [17] will take place by default.
2) The time-window will be written into a packet as a token and only propagated to the selected next sending node.
3) As soon as a slave received the token which contains a time-window, it sets a timer for $k \cdot w$ seconds equal to the time-window and starts to transmit its shuffling data. During this period, no other nodes will be sending.
4) When this timer ends, the node signals ('finished') the master. At the same time, the data transmission is halted and any newly generated shuffling data are queued.
5) Once the master node receives the slave's 'finished' signal, it floods a message to every node requesting for fresh updates, and then it restarts the procedure. Note that this broadcast message is identical to the one we used to reset and correct the slaves' clocks as we describe above.

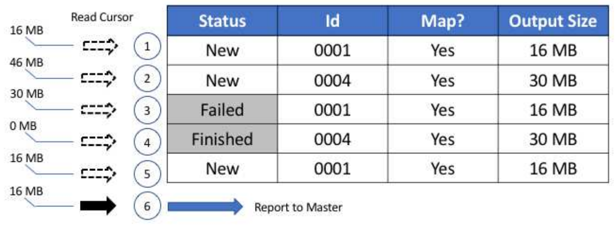
Time Synchronization. For every network scheduling solution, a common problem is the synchronization of the clocks. This condition is rarely achieved by just utilizing the machines' internal clocks. It is even more difficult for small IoT devices. Time shifting among the cluster's members consequently leads to ineffective scheduling decisions. We address this problem by placing a Centralized Scheduler inside our cluster, that is the master node. Other nodes act as workers and strictly follow the instructions from their scheduler for transmission duration. The workers block their outgoing traffic until a further instruction comes (the time-window token). A node can only transmit its output queue when it receives a time-window token from the scheduler. This node also notifies its master immediately after it spends the time-window. By this way, only one clock is ticking at a moment.

C. Monitor Module

Our monitor module consists of two major components, one is to monitor the run-time status of big data applications, and the other is to measure the wireless link qualities.

Hadoop Parameter Collector. This is the main implementation of monitor module that binds to Hadoop Framework core of every slave node. We inject our surveillance codes into the Shuffle Handler Channel as well as the Reducer Tracking Record of Hadoop to pullout useful information in runtime. The records to be kept track of will be store locally, such as job progress, reducer location, amount of output data that is ready to shuffle, have been shuffled and the mapper's output has been failed to send. Periodically, every worker keeps checking this module in background, reads the recorded information and reports the selected pieces to the Master (the Centralized Scheduler). The selected information is the amount of bytes needed to send or shuffle which was calculated based on the facts that some output data have already been sent, some were failed to be sent (time-out due to network congestion). Fig. 4 shows how the node calculates total size of data needed to be reported. Each slave node reads the recorded runtime information of Hadoop (read-cursor start from the last stopped point plus one) and accumulates the size of mapper's output.

If the status of current cursor's position is either "Finished" or "Failed", then this accumulating data size will be deducted by the data size in current position. The latest number is the selected information and will be reported. It is easy to see that at the end of a job, the number of new events in that job will be equal to the finished and failed events. The server collects these messages for computing and assigning time-windows to each slave. When the data have been sorted out, the slave node quickly notifies its master so that this server can cut short the assigned time-window and allows other nodes to take turn earlier.



Status	Id	Map?	Output Size
New	0001	Yes	16 MB
New	0004	Yes	30 MB
Failed	0001	Yes	16 MB
Finished	0004	Yes	30 MB
New	0001	Yes	16 MB

Fig. 4. An example of how the slaves read the recorded events, the read-cursor starts from the first record (at the beginning) or from the last stopped location (the following periods). Accumulated data size will be deducted in case the status is any other than"New"

Link Quality Measurement Unlike traditional clusters whose all member servers are close to each other, an IoT cluster is likely spreads over a much larger area with complex terrain. Link quality or bandwidth between a pair of node may be different with other pairs. Some have lower speed network links and would require more air-time to transmit than others. Meticulously measuring and updating the link quality of every pair to master node can increase the efficiency of the packet scheduling algorithm. Another critical benefit of this update is allowing the transmitter to balance the speeds between its virtual NIC and physical NIC. The architecture of using virtual NIC will be discussed in the next subsection. Basically, our system create a virtual NIC named VNIC0 that is the direct network interface to applications. At the beginning of a time-window, a node starts to transmit by just writing the data to its VNIC0. Since this action is much faster than the physical NIC speed. Therefore, at the end of such time-window, there often are hundreds to a thousand of packets are still in the egress queue of the physical NIC. Consequently the devices keep interfering the channel. We implemented this module to measure and update the speed between two nodes in packet per second. Thus the node can control how many packets can be carried out per second.

D. Packet Control Protocol

To optimize the effectiveness of our scheduling decisions and forward to minimize the wireless interferences, we designed our Packet Control Module that could guarantee only one host is transmitting at a time.Thus, such a way requires the machines to delay and hold their outgoing packets while waiting for the master's instructions to indicate a new transmission period. We satisfy this requirement by implementing the packet scheduling in OWLBIT using Virtual Network Interfaces, VNIC. This Linux's feature (Tun/Tap) [18] offers a way to bring networking to user space and thus user's programs cannot only see but also manipulate raw network traffic. We split the packet flows into two phases:

- Upon an outgoing packet from local processes (Hadoop-Yarn in this case) is ready, it is directed out via a virtual network interface (VNIC1) instead of a physical NIC. From here the packet is copied to machine's memory, in a linked queue, waits for a scheduled time slot to be actually transmitted out. We can call this interface a dummy outgoing port, because the packet has not wen anywhere yet. Intuitively, the outgoing traffic can be held or delayed just by this dummy port.
- Whenever a node receives the token from server that allows it to transmit, those packets in its memory will be delivered to physical NIC (WiFi NIC) in FIFO orientation via another virtual network interface, VNIC0. We will discuss the need of this VNIC0 right below.

Why VNIC0? Commonly, to send data to the remote destinations, a programmer would utilize sockets (e.g., TCP sockets) to setup the network channels. Since socket is point-to-point connection, there will be multiple pairs of socket for each pair of transmitters. On the other hand, more often, the linked queues carry a list of packets aim to different destinations, therefore, he or she, the programmer must also takes care the routing mechanisms. With the scale of IoT devices, a normal IoT cluster could contain hundred to thousand units, consequently, the routing and socket processing at user-level burden the limited resource of such small devices. Moreover, another problem needs to be accounted is fragmentations, Fig.5 demonstrates how a big data packet goes through the sockets. Very much like any physical interfaces, a virtual network interface distributes packets in MTU size. Recall that a packet got out of the dummy interface are still inside the device, soon or later, this packet must travel through a real NIC on reaching its remote destinations. Thus, such packet is likely to be sliced again into multiple chunks while carrying out the physical interface due to additional headers. Now, at the receiver site, if sockets are in used, the operating system decides to forward this packet to the socket corresponding to the sender. From here this packet will be redirected to VNIC1 i.e., the dummy interface since the Hadoop-Yarn's processes are binding to it (We cannot just send data to Hadoop-Yarn's processes). The programmer's codes should be smart enough to completely read every packet from the socket e.g., read multiple times per packet. Otherwise, incomplete packet will be discarded at VNIC1 by default, like every physical NIC. We offer a

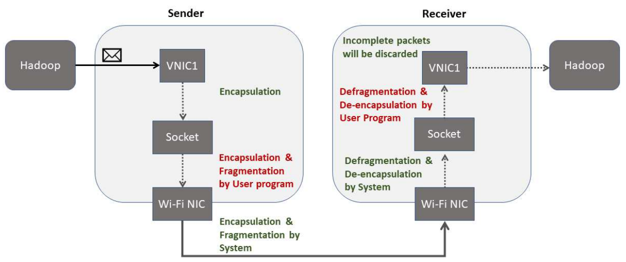


Fig. 5. Packet journey with sockets, some routing and fragmentation tasks needed to be processed at user-program level

way to ease these works and problem by creating another virtual network interface i.e., VNIC0. Fig.6 shows the packet being fast forwarded across the journey. All the packets in memory will be pushed to the Forwarding Information Base (FIB) by just writing them back to the VNIC0. A difference between the original packets and those who have been pushed back to FIB is the Input-interface which is the VNIN0. Using this conditional element, we can redirect the traffic out of an interface that we want (in this case, the physical wireless interface) using Policy Based Routing. The act of creating a loopback without changing the IP headers may trigger IP-Spoofing prevention, a common security feature in Linux that has another name Unicast-Reverse-Path Filtering (uRPF). Even if IP-headers have been changed, packets could still be dropped due to the lack of routing information. Disable this feature could be dangerous .Fortunately, the Tun/Tap interfaces are just internal; therefore, disabling IP-Spoofing will not create any vulnerabilities. On the other hand, Edge devices do not involve in any enterprise routing, in fact, they only connect with each other in the same rack. For these reasons, it is safe to turn-off uRPF. With a few additional tweaks in routing configuration we made sure that FIB knew where to forward these traffics. Thus the routing is delegated. As packets go down to lower layers of the kernel's network, reaching remote hosts, the fragmentations if any. will be well take-care by operating systems as usual. More importantly, the kernel at receiver site is smart enough to send this packet to Hadoop-Yarn directly. Hence fragmentation will not border us anymore. And lastly we can replace multiple sockets implementation (even raw sockets) with just a few lines of codes.

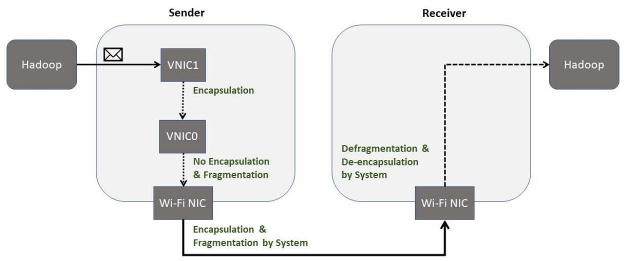


Fig. 6. Packet journey with VNIC0, delegate most of the complex network tasks to the kernel

E. Inter-process Messaging

As our implementation is split into several small modules, each unit in the cluster running an instance of these module. Yet requires an effective mechanism for these modules to communication with each other. We adopted ZMQ a light weight, fast, simple and reliable messaging library. Its allows us to be able to push hundreds or thousands of hundred-byte messages per second within the cluster. Although these message exchanging happen very fast compared with the data shuffling, we still want to assure they do not interfere with the data transmission. By having the messaging traffic flows in a different WiFi channel, we eliminate the interference caused by inter-process communication if/any.

V. Performance Evaluation

We evaluate OWLBIT on two dedicated clusters of IoT devices. Table II shows the detail setup and specifications of these clusters. Each cluster is comprised of nine (9) Raspberry Pi 3 model B with identical hardware and network configurations. They are all running Raspbian Jessie Lite as the operating system (Linux 4.9.13-v7), the minimal image based of Debian Jessie. We attach an additional WiFi NIC to each Raspberry Pi and use this interface as our main network channel in OWLBIT. Since the build-in WiFi NIC is less flexible in term of configuring and monitoring, we assign them for the inter-process communication discussed in the previous section. In addition to keeping the network private and avoiding middle hops in packet transmitting, we use wireless ad-hoc mode in stead of managed mode (usually comes with access points).The maximum measured throughput we can get with these configurations is, roughly 23 Mbps. We target our solution to achieve this number. Because IoT devices mainly use wireless connections to communicate with each other, we place a wireless sniffer near the clusters, and by this way, we are able to collect the experiment results.

TABLE II. Cluster Configuration

Category	Configuration
Processors	Quad Core 1.2GHz Broadcom ARM BCM2837 64bit CPU
Memory	1GB RAM
First Wireless Network	EW-7811Un USB dongle wireless LAN
Second Wireless Network	BCM43438 Build-in wireless LAN
IOs	4 USB 2 ports, HDMI
Other Network	Ethernet LAN 10/100/1000
Number of nodes Per Cluster	09
Wireless Sniffer	a commodity computer with Ubuntu 16

TABLE III. Hadoop Configuration

Setting	Master	Slaves
Map Task Memory	none	256MB
Map Task CPU	none	1 vCore
Reduce Task Memory	none	512MB
Reduce Task CPU	none	1 vCore
Number of Reducers	none	1
AM memory	128MB	128MB
AM CPU	1CPU	1CPU
Replicas	8	8
BlockSize	16MB	16MB

A. Methodology

We choose a Hadoop-Yarn version that supports ARM chipsets as the Big Data Framework (Hadoop 2.7.2). One raspberry pi node in each cluster is assigned as the master, in charge of Hadoop Jobtracker and Namenode services. The remaining nodes are the slaves or workers, running the Hadoop's services such as Tasktracker and Datanode. Due to the limitation of hardware's resources, we cannot reuse the default configuration provided by Hadoop. For example, the default requirement of 4 GB of RAM for each map-task, is fours times larger than the total available RAM of a Raspberry Pi 3. Moreover, We also realize that such IoT devices have much slower CPU speed compared with cloud-site servers. Therefore, large Hadoop Distributed File System (HDFS) block size will be impractical in term of computational time. Table III summarizes our version of Hadoop's default configurations that are applied to the experiments. For input data sets, in this paper, we only focus on the shuffle stages of MapReduce, thus input data sets are distributed to HDFS before hand with the number of replicas equal to the number of slaves, i.e., eight replicas. This intensional setup maximizes the benefits of data locality feature. Hence the network traffics are mainly data shuffle.
To evaluate OWLBIT against native Hadoop, we setup one Raspberry Pi cluster with the configurations in Table III while the other cluster, with the same setup, has OWLBIT

installed as the difference. We use terasort and wordcount benchmarks from Hadoop distribution as well a HIPI based images transcoding application [19] to evaluate the two clusters. HIPI is an open source library for image processing which is designed to adopt the advantage of Hadoop MapReduce parallel computing.

Hadoop Benchmarks. We choose terasort as the main Hadoop benchmarking since the intermediate data size generated by this program are naturally at least the size of the input. To test OWLBIT with various size of data sets, we ran teragen to generate multiple sets of data, this is a data generator program that's also included in Hadoop distribution. For wordcount, we have a bash script that can generate large text files of none duplicated words, this type of text files make wordcount emits as much intermediate data as the input. We divide these Hadoop benchmark workloads into three categories, based on the size of input data: small, medium and large. Small jobs are jobs with the input of 256 MB or less, medium jobs have input size of 512 MB and 1024 MB are the large jobs.

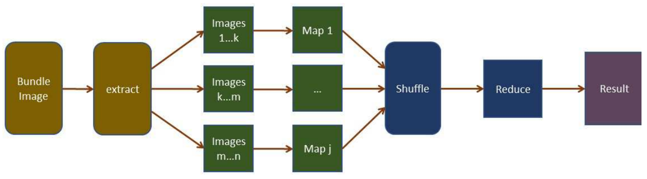


Fig. 7. Architecture of a HIPI MapReduce program

Image processing with HIPI. In this experiment, we simulate a real-life scenario where a private cloud of IoT devices can assist traditional cloud by pre-processing data. We deploy HIPI, a handful open source library to our clusters. As described in [19], HIPI facilitates efficient and high-throughput image processing with MapReduce style parallel programs typically executed on a cluster. It provides a solution for how to store a large collection of images on the HDFS and makes them available for efficient distributed processing. Fig. 7 illustrates every step of a MapReduce program that uses HIPI library. The input data is an image bundle object (HIB) which is a collection of pictures in a folder (any type of image format). This object can be easily created by the provided handful tools. At the first stage, input file (HIB object) will be extracted to a series of selected images that satisfied the predefined filtering. Then these selected images will be assigned to map tasks proportionally to perform some common prepossessing such as cropping, color space conversion and scaling. Map tasks then emit and shuffle intermediate data to reduce phases for next computation tasks, e.g., compressing, feature extracting or format converting. For this experiment, we adopt one of the HIPI's example codes, i.e., HibToJpeg, this application will convert pictures of various format to jpeg extension. Since the original source code does not have reduce phase, we modified the codes so that the program will run some extra activities at reduce phases hence there must be the shuffle sub stages.

B. Results

Improvement of Job Execution Time: We first show the advantage of OWLBIT via the Job Execution Time. For each job type and workload .e.g., terasort with 256MB input data, wordcount with 512MB input data, HibToJpeg and 256MB of input pictures etc., we take the difference between job execution time associates with native Hadoop cluster and the one under OWLBIT to show the improvement. We average each job over five iterations.

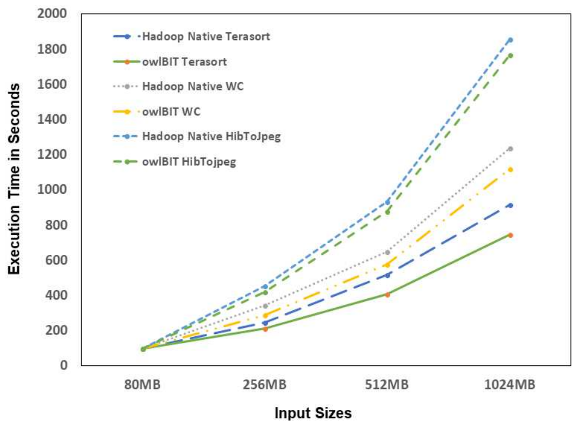


Fig. 8. Job Execution Time Comparison

In Fig. 8, we plot the differences in job completion time of all types of jobs and data sizes. The plot shows that under OWLBIT, the execution times of every job are shorter compare with native Hadoop. Terasort job in OWLBIT cluster with 256MB input data has the shortest execution time about 212 seconds and this trend continues to the greater data sizes (512MB and 1024MB). OWLBIT generated big distances from the corresponding trend in native Hadoop with the peak reached 168 seconds. The figure also shows other jobs have longer completion times than terasort , for examples, the image conversion jobs (HibToJpeg) require heavier computation, while wordcount needs to distribute the result at the end. However all jobs associate with OWLBIT have shorter execution time than their Hadoop native counterparts, the differences are from 33 to 116 seconds earlier. These achievements are because of the improvements in quality of wireless network, but there is another factor that expedite the job completion. Recall that our scheduling algorithms granulate network traffic to intermediate data unit which is a block size of data, as OWLBIT tries to deliver a full block per time-window. The arrival time of the first intermediate block must be sooner than the case in native Hadoop where output data blocks transmitting in waves (send multiple at a time). As soon as receiving a whole intermediate block, the reducers start the computation and thus the execution time of reduce phases must be shorter.

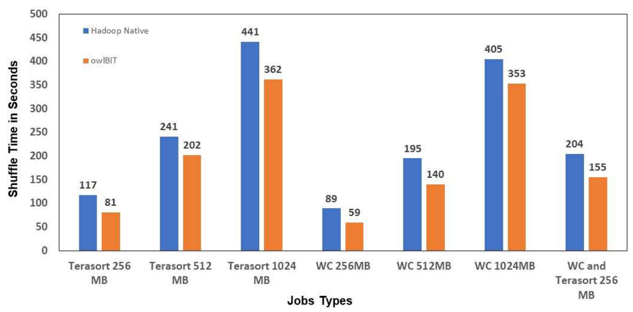


Fig. 9. Hadoop Benchmarks Shuffle Time Comparison

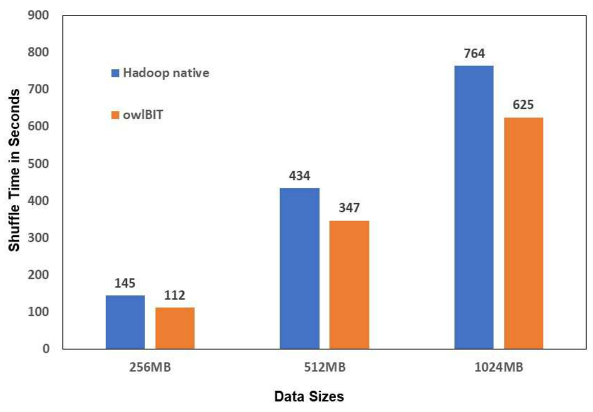


Fig. 10. HibToJpeg Shuffle Time Comparison

Accumulated Shuffle Time : As mentioned, shuffle stage is our main target which usually dominates a large portion in job completion time. The results in Fig. 9 show that Hadoop benchmarks achieved significant improvement in shuffling time, from 12% to 33% faster than the Hadoop native. These achievements re-appear in the image processing experiments with reductions of shuffling time are from 10% to 15% (Fig. 10). Our scheduling solution reduce a great deal of interference by assuring only one host can transmit at a time. By extracting the information from the sniffer, we collect WiFi traces and plot the transmissions of all nodes through time. We note that the rate adaptation is barely seen in the traces, hence the overall throughput in shuffle stages are nearly maximum speeds (23 Mbps). Our piloting in Fig. 11 hardens this conclusions by showing the clarity of different transmission sessions. Each chunk of connected bars with the same color represents the shuffling traffic of a node during a time-window.

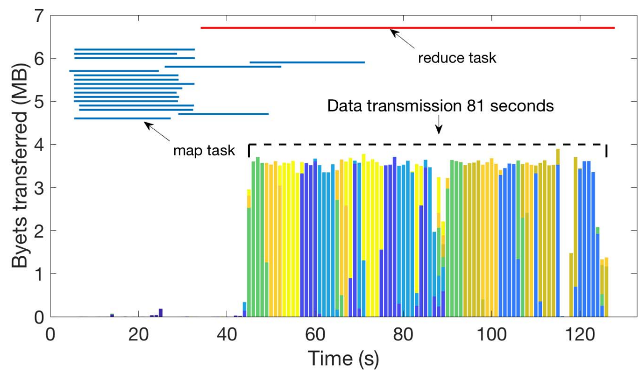


Fig. 11. An example of sorting 256MB data with OWLBIT

VI. Conclusion

In this paper, we present a framework that supports big data platforms running on IoT devices. The main design intuition is to utilize application-layer information to guide the data packet scheduling at the link layer. We implement all the modules in state-of-art devices and platforms (Raspberry Pis and Hadoop-Yarn). Our evaluation is based on real system experiments, and the results show significant performance improvement.

References

[1] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[2] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[3] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, March 2010.

[4] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 17–24, Nov 2010.

[5] M. Hammoud, M. S. Rehman, and M. F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 49–58, June 2012.

[6] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, 2014. USENIX Association.

[7] Vivek Shrivastava, Nabeel Ahmed, Shravan Rayanchu, Suman Banerjee, Srinivasan Keshav, Konstantina Papagiannaki, and Arunesh Mishra. Centaur: Realizing the full potential of centralized wlans through a hybrid data path. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, 2009.

[8] M. Kim, S. Han, and M. Lee. Demand-aware centralized traffic scheduling in wireless lans. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 144–152, May 2016.

[9] J. Manweiler, N. Santhapuri, S. Sen, R. Roy Choudhury, S. Nelakuditi, and K. Munagala. Order matters: Transmission reordering in wireless networks. *IEEE/ACM Transactions on Networking*, 20(2):353–366, April 2012.

[10] Ramana Rao Kompella, Sriram Ramabhadran, Ishwar Ramani, and Alex C. Snoeren. Cooperative packet scheduling via pipelining in 802.11 wireless networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design and Analysis*, E-WIND '05, pages 35–40, New York, NY, USA, 2005. ACM.

[11] P. Garg, R. Doshi, R. Greene, M. Baker, M. Malek, and Xiaoyan Cheng. Using ieee 802.11e mac for qos over wireless. In *Conference Proceedings of the 2003 IEEE International Performance, Computing, and Communications Conference, 2003.*, pages 537–542, April 2003.

[12] A. Banchs, M. Radimirsch, and X. Perez. Assured and expedited forwarding extensions for ieee 802.11 wireless lan. In *IEEE 2002 Tenth IEEE International Workshop on Quality of Service (Cat. No.02EX564)*, pages 237–246, 2002.

[13] C. Crespo, J. Alonso-Zarate, L. Alonso, and C. Verikoukis. Distributed point coordination function for wireless ad hoc networks. In *VTC Spring 2009*, pages 1–5, April 2009.

[14] Young-Jae Kim and Young-Joo Sun. Adaptive polling mac schemes for ieee 802.11 wireless lans. In *The 57th IEEE Semiannual Vehicular Technology Conference, 2003. VTC 2003-Spring.*, volume 4, pages 2528–2532 vol.4, April 2003.

[15] M. Barry, A. T. Campbell, and A. Veres. Distributed control algorithms for service differentiation in wireless packet networks. In *IEEE INFOCOM 2001*, 2001.

[16] owlBIT. https://github.com/bboycoi/RPi-Hadoop/.

[17] Rasmus V. Rasmussen and Michael A. Trick. Round robin scheduling - a survey. *European Journal of Operational Research*, 188(3):617–636, August 2008.

[18] TunTap kernel description. https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[19] HIPI hadoop image processing interface. http://hipi.cs.virginia.edu/.