Lark: Bringing Network Awareness to High Throughput Computing

Zhe Zhang*, Brian Bockelman*, Dale W. Carder†, Todd Tannenbaum†
*University of Nebraska-Lincoln
{zzhang, bbockelm}@cse.unl.edu
†University of Wisconsin-Madison
dwcarder@wisc.edu, tannenba@cs.wisc.edu

***Abstract*—High throughput computing (HTC) systems are widely adopted in scientific discovery and engineering research. They are responsible for scheduling submitted batch jobs to utilize the cluster resources. Current systems mostly focus on managing computing resources like CPU and memory; however, they lack flexible and fine-grained management mechanisms for network resources. This has increasingly been an urgent need as current batch systems may be distributed among dozens of sites around the globe like Open Science Grid. The Lark project was motivated by this need to re-examine how the HTC layer interacts with the network layer.**

In this paper, we present the system architecture of Lark and its implementation as a plugin of HTCondor which is a popular HTC software project. Lark achieves lightweight network virtualization at per-job granularity for HTCondor by utilizing Linux container and virtual Ethernet devices; this provides each batch job with a unique network address in a private network namespace. We extended HTCondor's description language, ClassAds, so users can specify networking requirements in the job submission script. HTCondor can perform matchmaking to make sure user-specified network requirements and resource-specific policies are fulfilled. We also extended the job agent, *condor_starter*, so that it can manage and configure the job's network environment. Given this important building block as the core, we implement bandwidth management functionality at both the host and network levels utilizing software-defined networking (SDN). Our experiments and evaluations show that Lark can effectively manage network resources within the cluster with low overhead. It provides the users with better predictability of their job execution and the administrators more flexibility in network resource consumption policies.

Keywords*—high throughput computing, HTCondor, bandwidth management, software-defined networking, network-aware scheduling.

I. Introduction

Cluster computing has become the workhorse powering scientific discovery and engineering research. At the heart of many compute clusters is a general-purpose workload management batch system such as PBS [1] or Grid Engine [2]. The goal of these batch systems is to ensure submitted jobs ultimately run to completion on cluster resources according to a supplied policy. Achieving this goal is facilitated by resource management mechanisms including scheduling, monitoring, accounting, and binding to specific jobs [3]. However, batch systems rarely treat the network as a first-class resource that can be directly managed and bound to jobs. SLURM [4] can perform topology-aware job placement to reduce communication overhead; however, it cannot interact with the network layer directly, thus leading to limited functionality. For example, policies that allocate or prioritize network access to different sets of jobs cannot be effectively monitored or enforced. Network-based tools on their own are of little help to enact job-specific policies because the network has no ability to distinguish between different jobs running on the same host.

The need for batch systems to manage the network is increasing as the size of data sets being processed grows, and as high-throughput computing clusters have become federated into wide-area international computing grids that connect hundreds of clusters across dozens of countries [5]. In these sorts of environments, as well as single cluster installations, HTCondor [6] is a widely deployed workload management system in both commercial and academic settings. Traditionally, the computing resources fully managed by HTCondor have been CPU, memory, and disk. Accordingly, we started the Lark project that aims to make networking a first-class managed resource of HTCondor. This paper describes our progress to date. We explain how the Lark software leverages recent functionality in the Linux kernel to implement a mechanism that provides a unique network identity for each compute job by binding jobs to unique network address. Given a one-to-one mapping of network addresses and jobs, HTCondor can now interact with and alter the network layer based on its internal policies. We also explain how Lark defines new job and machine attributes in a policy language, allowing users to describe networking requirements at job submission time and resource administrators to enforce network policies. The existing HTCondor matchmaking techniques can then match jobs to appropriate hosts.

To demonstrate the utility of these techniques, we describe our implementation of a bandwidth management application for HTCondor. This allows us to control the available bandwidth and perform network accounting for each individual job on the host, even when many jobs are all running on the same hardware. Furthermore, we explain how we integrate this with software-defined networking via a job-aware OpenFlow [7] controller to enable a complete network-level view of the running HTCondor jobs. In this way, we can incorporate application level information to perform flexible wide area network (WAN) traffic management.

While our implementation is specific to HTCondor, the basic approaches can be generalized to other software, and the mechanisms described could apply to any software running atop the Linux kernel.

II. Background and Related Work

A. Adopted Technologies

Two essential Linux kernel features leveraged by Lark are *namespaces* [8] and *virtual Ethernet pairs*. In Linux, a namespace is a set of processes that have the same view of a system resource. Unless namespaces are explicitly created, all processes are in the "system namespace." For example, processes in a mount namespace all see the same file system mount table, which may be a different set of mounts than on the rest of the system. HTCondor uses mount namespaces to give each job a separate `/tmp` directory. Different processes in two different jobs see a different set of files in `/tmp`. Jobs can also be run in separate PID namespaces, meaning processes cannot see or affect each other through the `kill` system call or via the `/proc` file system. Unlike traditional Unix process groups, to change a namespace requires elevated capabilities such as root-level access. Namespaces can be nested - resources in one namespace can be moved into a child namespace. One resource that can be managed by namespaces are network devices. Consider a host with two physical network devices, `eth0` and `eth1`. If the devices are placed in network namespaces A and B, respectively, then processes in namespace A could not send packets through `eth1` and processes in namespace B could not send packets through `eth0`.

A second kernel feature leveraged by Lark is the *virtual Ethernet device*. Pairs of these devices are similar to Unix socket pairs; data sent into one device, `veth0`, will come out through the other device, `veth1`. By placing the devices in the two namespaces respectively, we can link together two different network namespaces. The combination of virtual Ethernet devices and network namespaces are essential tools for Lark's mechanism described in the next section to provide a unique network identity for each job.

In addition to Linux kernel extensions, our Lark work also relies on the software-defined networking (SDN) approach where aspects of the network are implemented in software rather than hardware. The most popular approach to SDN currently appears to be the use of the OpenFlow [7] protocol. In the OpenFlow model, a switch is split into a control plane and a data plane. The data plane consists of a table of simple rules for packet processing. In a typical Layer-2 switch, the control plane hardware makes forwarding decisions and inserts processing rules into the data plane. When a switch is OpenFlow-enabled, the hardware control plane is replaced by an external software controller. If the data plane does not know how to handle a packet, the packet is forwarded to the controller. The controller processes the packet, determines an action, and then installs new data plane flow rules in response. Because the controller is implemented in software and can manage multiple switches, there is more flexibility to "program" the network compared to hardware control planes. OpenFlow provides the abstraction model and protocol specification for communication between the switch and the controller. We utilize OpenFlow to program the network with respect to the HTCondor system, but there are many other applications, especially in cloud computing [9].

It is important to contrast this work with the use of virtual machines (often in the context of cloud computing). Container-based technologies are more lightweight - as they share the same kernel, startup and shutdown can occur in less than a second. The technologies used are less resource-intensive, especially with respect to memory usage. Finally, using containers allows us to "mix and match" pieces of the host environment; for example, we can reuse the host filesystem. In our anecdotal experience, scientific users rarely want to maintain their own userspace.

B. Related Work

1) Bandwidth Management: Previous work in HTCondor aimed to manage network bandwidth per subnet by measuring the sizes and locations of the job executable and focused on primarily managing the bandwidth used in the checkpointing of jobs [10]. A key limitation of this approach is that it is not possible to enforce the bandwidth usage, it requires the job to be relinked with a special library, and it is not possible to perform network management at the granularity of a particular job.

Seawall [11] can divide network capacity within the data center according to the weights of high-level entities (e.g. VMs). It requires a shim layer in the virtualization or platform network stack and is designed for cloud computing environments. In contrast, Lark operates on a stock Linux vendor kernel and does not require the use of virtual machines. Eschewing virtual machines is important to many in the scientific community who demand peak return from their cluster hardware and thus want their jobs running on physical hardware. This makes it possible to access non-virtualized co-processors and avoid incurring the additional administrative overhead of maintaining VM images.

2) Network Virtualization: Userspace-level network virtualization techniques involve using the POSIX `ptrace` method to capture all system calls performed by an application. The Parrot [12] software as part of the CCTools project is a good example of this technique. The system calls, once intercepted, can be re-implemented in user space for the purpose of implementing an alternate networking stack. This approach allows the host to do arbitrary transformations to the networking stack. Unfortunately, this method has a high overhead for some applications because every system call is transferred to a user level process and does not allow integration with the networking hardware. Thus, this technique does not integrate well in the presence of network traffic which is not managed by the intercept application.

In the ViNe system [13], one or more virtual addresses were added to a given host along with a few static routes. The traffic sent on these static routes is intercepted by a userspace application and encapsulated onto a virtual network. This allows precise control over the network usage across several nodes in almost-arbitrary environments. However, this does not provide job-level granularity nor does it provide control over traffic destined for the Internet.

3) Application-aware Network Management: Application-aware network management solutions are emerging in some high-level cluster programming frameworks (e.g. Hadoop and Spark). Such examples include Orchestra [14] and Varys [15]. However, a tremendous amount of computer cluster jobs are a "black-box" process tree that could range from compiled native

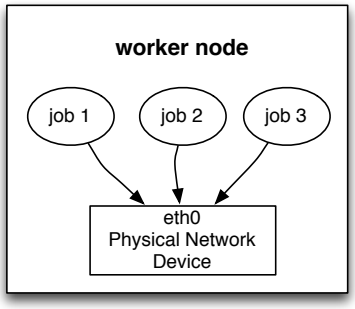


Fig. 1: Traditional network configuration for submitted jobs on an HTCondor worker node

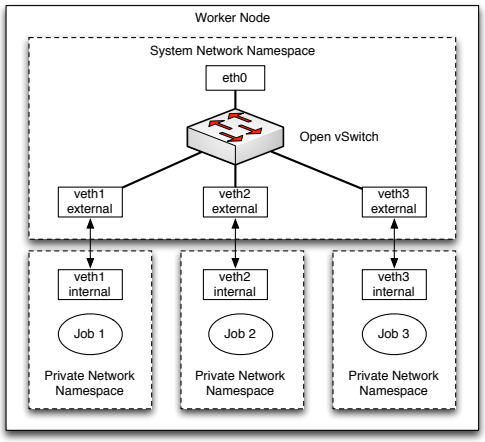


Fig. 2: Open vSwitch bridging configuration for different batch jobs. (Each dotted square represents a separate network namespace)

code to scripts to MATLAB interpreters. Lark enables network management even for black-box jobs that do not adhere to any specific programming model such as a MapReduce-style or distributed SQL model.

III. System Architecture

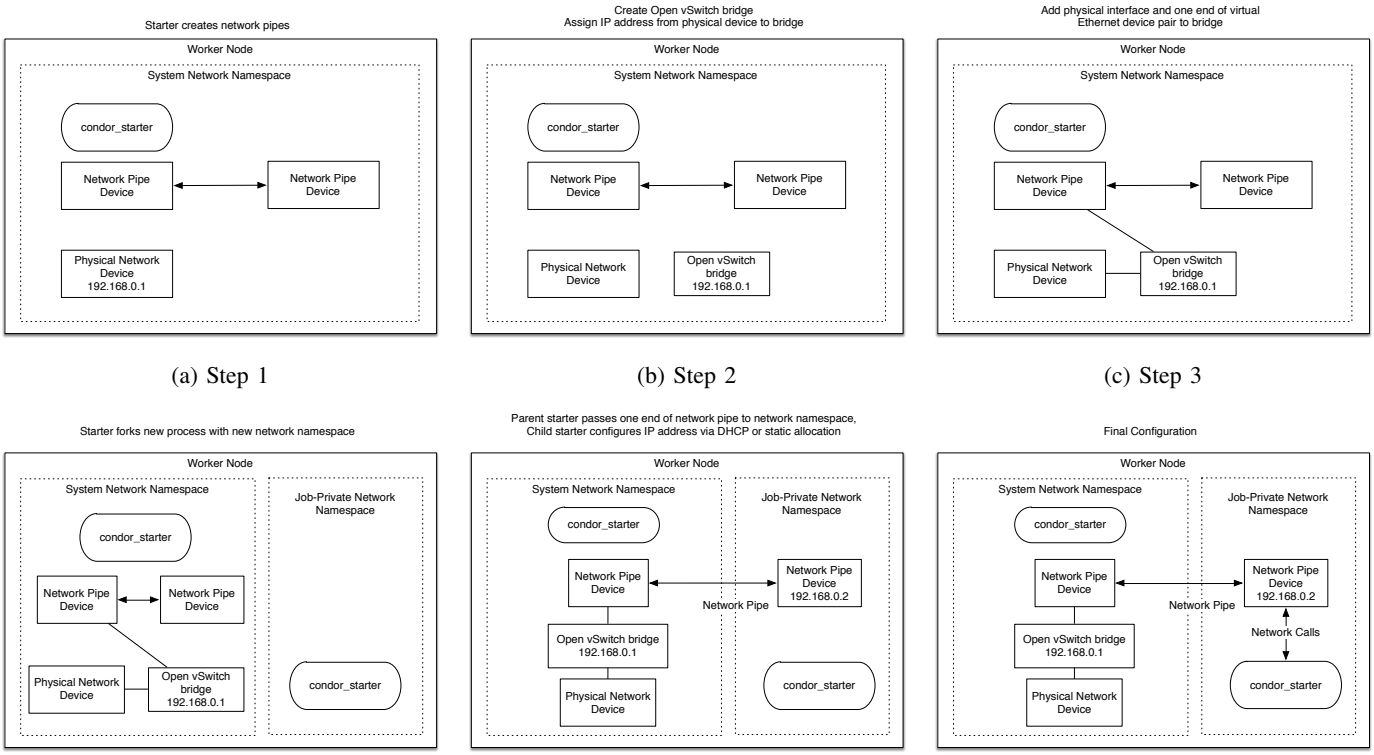
HTCondor runs several processes on a cluster's worker node. There is a single *condor_startd* process which represents the machine resources and one-or-more *condor_starter* processes, each of which manages the state and environment of a running job. Previously, all running HTCondor jobs share the network resource of the worker node. On Linux, this means jobs can use BSD sockets to communicate with the network through a network interface. However, there is no isolation or encapsulation among different running jobs in terms of network resource multiplexing such as bandwidth and addressing. Figure 1 illustrates this configuration. Because all jobs share a physical network device and are represented by a single network address with potentially several ephemeral port numbers in use, the external network has no way to implement a per-job policy.

In Lark, we have extended the *condor_starter* to have a per-job network address. When the starter forks to launch a job, it will create a new network namespace and a pair of virtual Ethernet devices. The parent is in the system network namespace and the child process is in the new network namespace. The parent process keeps one of the virtual Ethernet devices (the *external device*) and passes the other end to the child (the *internal device*). The starter can then integrate the child process into the external network in one of several ways, depending on the configured policy. Thus, each job has its own network interface and each job can isolate their network activities from the others. According to the user's requests and system policy, Lark can apply different network configuration policies on the host machine. Currently, there are three options provided in Lark: Linux bridge, Open vSwitch [16] bridge and NAT (network address translation).

Figure 2 demonstrates the network configuration in Open vSwitch bridging mode. The Linux bridge device has a similar configuration. Open vSwitch provides more flexible APIs for topology management and device configuration compared to the bridge device. Further, Open vSwitch supports software-defined networking (e.g. OpenFlow [7] protocol) which we will utilize in Section VI to better integrate the HTCondor job into the network.

In this configuration, an Open vSwitch bridge `ovsbr0` is first created, then a physical network device `eth0` is added as a port of the bridge. The IP address and default routes are moved from `eth0` to `ovsbr0`. When a batch job is scheduled to execute on this machine, a pair of virtual Ethernet devices (connected to each other with a network pipe) are created. The external device is also added as a port to the bridge. The internal device is moved into a new network namespace that is only visible to the job. The internal device can retrieve an IP address via DHCP or static configuration. Finally, the job is moved to the new network namespace. This results in a private network namespace to the batch job and each batch job can have an unique IP address. Figure 3 illustrates the whole process step by step in a more visual manner for readers' better understanding. Compared with the traditional network configuration, each job now has better network isolation and there is a one-to-one correspondence between the IP address and network flows from the job, making it possible to implement network scheduling policies based on fine-grained job information.

As an alternate to bridge mode, Figure 4 demonstrates the configuration of NAT. Compared with bridging mode, there is no Layer-2 bridging among devices. Each job is assigned a pair of virtual Ethernet devices and a private network namespace. The packets from the external device are NAT'd by the system `iptables` configuration. Therefore, each batch job shares the IP address of the physical Ethernet network device as the source IP address and the external network would only see the host's IP address, not the job's. This configuration is ideal for batch jobs that only require outbound connectivity but not inbound connectivity as the job has no publicly routable IP address on the network. This is a useful outcome in IPv4 where addresses are scarce, but not a pressing concern with IPv6. This mode also prevents the external network from performing any special per-job customization, thus any policies must be enforced by the local host.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

Fig. 3: Network namespace manipulation in Open vSwitch mode step by step

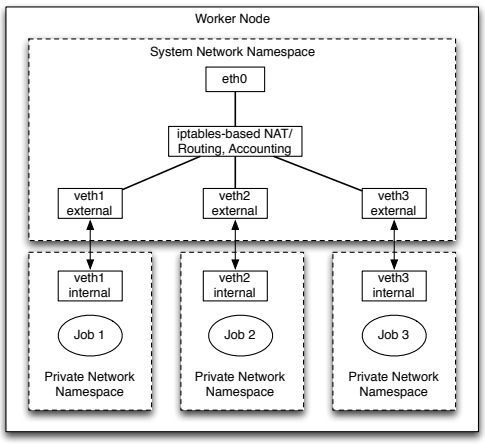


Fig. 4: NAT configuration for different batch jobs. (Each dotted square represents a separate network namespace)

IV. Network Policy

The network configuration technique described in Section III becomes more interesting if the configuration process can be made aware of system and job policy. One of the objectives of Lark is to actively alter the network layer based on the user request when they submit the batch jobs. For each running job, there are two relevant HTCondor ClassAds [17], the *job ad* and the *machine ad*. These ads contain key-value pairs which describe the user's job policy and the resource's policy, respectively. We augment the existing ads with network-related attributes, which describe users' demands for network configuration and network related attributes at the worker node. Then, policies can be set and matchmaking can be performed through the existing ClassAd mechanisms. For example, new expressions can be added to the job `Requirements` attribute to prevent the job from running on a host that cannot provide sufficient bandwidth.

These two ClassAds are given into the network policy component of the *condor_starter*. The network policy component converts the user's requests to the appropriate network configuration on the host. Table I and Table II list the two sets of network-related ClassAd attributes for the job and machine, respectively, and their corresponding meanings. The machine ad attributes are evaluated within the context of both the machine and job ad, then given to the policy component for network configuration.

When the users submit batch jobs, any ClassAd attributes from Table I can be specified in their job description. For example, if the job requires HTCondor IPv4 connectivity for the worker node, the following may be added to the submit file:

```
+IPProtocol = "IPv4"
Requirements = TARGET.IPProtocol =?= \
    MY.IPProtocol
```

Job ClassAd Attributes	
IPProtocol	Preferred IP protocol (IPv4 or IPv6)
RequestBandwidth	Requested guaranteed host bandwidth.
NetworkAccounting	Whether or not to perform network usage accounting for the job.
InboundConnectivity	Whether or not inbound connectivity for the job is desired.
OutboundConnectivity	Whether or not outbound connectivity for the job is desired. Note that setting InboundConnectivity and OutboundConnectivity to false will effectively disable all networking for the job.
RequestedVLAN	Determines which VLAN the users want their jobs to run in.

TABLE I: Network related job ClassAd attributes

Machine ClassAd Attributes	
IPProtocol	The available IP protocol (IPv4 or IPv6).
AvailBandwidth	Indicates the negotiated speed of the host network (Mbps).
VLAN	The VLAN ID of the host network interface.
NetworkType	The type of Lark network the host can configure: NAT, bridge or ovs_bridge.
AddressType	Method for determining the network address for the job (static or DHCP).
BridgeInterface	The name of a local Ethernet interface to add to the bridge.
ExternalInterface	The name of the system-level virtual interface corresponding to the starter.
InternalInterface	The name of the internal starter virtual interface.
NetworkAccounting	Whether network accounting is available.

TABLE II: Network related machine ClassAd attributes

Here the `=?=` operator checks if the left hand side operand is identical in both type and value to the right hand side operand, returning `TRUE` when they are identical. Since the policy expression for `Requirements` is too long for the space here, we use `\` to break it into multiples lines (`\` serves the same purpose in the following policy expressions). The `Requirements` line above will prevent the job from matching hosts with IPv6 connectivity. Since HTCondor IPv4/IPv6 mixed mode is still a work in progress, here we assume that HTCondor runs either in IPv4 or IPv6 mode. Other sample policies include:

- Suppose the user specifies the InboundConnectivity and OutboundConnectivity attributes, and the machine ad specifies the following NetworkType:

  ```
  NetworkType = ifThenElse( \
   TARGET.OutboundConnectivity, \
   ifThenElse( \
    TARGET.InboundConnectivity, \
    "bridge", \
    "NAT"), \
   "null")
  ```

  The `ifThenElse` operator takes three operands. The first operand is a conditional expression. If it is evaluated to be `TRUE`, the operator evaluates and returns the value given by the second operand; if it is evaluated to be `FALSE`, the operator evaluates and returns the value given by the third operand. When evaluated by the *condor_starter*, the NetworkType will be set appropriately according to the user policy.

- Suppose the job ad sets RequestedVLAN to "physics" and the machine ad sets VLAN to "chemistry". Then, the user and machine ad will also need the following requirements, respectively:

  ```
  Requirements = TARGET.VLAN =?= \
    MY.RequestedVLAN
  Requirements = \
    TARGET.RequestedVLAN is null || \
    TARGET.RequestedVLAN =?= MY.VLAN
  ```

  Without the mutual requirements, it would be possible to run the job on this host, resulting in the incorrect VLAN for the running job.

- If network accounting capability is desired, but not required, the user would specify

  ```
  +NetworkAccounting = true
  ```

  By not adding a `Requirements`, the job can still run on worker nodes without this capability.

If the declarative ClassAd-based matching is insufficient for the final network configuration, the *condor_starter* can invoke an arbitrary script. The script is given both job and machine ad via stdin and HTCondor will use the resulting stdout as the new machine ClassAd. This mechanism, while allowing additional customization, cannot be utilized for matchmaking. The prior VLAN example is a case where this mechanism is applicable. After successful matchmaking, the *condor_starter* will still evaluate the VLAN attribute to "chemistry"; however, an integer must be specified for the VLAN ID. The callout script can be used to translate the user-friendly name "chemistry" to an actual VLAN ID before the Lark code is invoked.

These policies currently are still simple and "first generation". With the further development of Lark, we would like to provide the users with a richer set of network related ClassAd attributes and more complicated policies to alter the network layer.

V. Bandwidth Management

One example application of the techniques in Sections III and IV is host bandwidth management and monitoring, available in the Open vSwitch bridging mode. The machine-level daemon, *condor_startd*, can be instructed to treat bandwidth as an allocatable resource. At startup, the current worker node bandwidth is determined by calling out an executable. Currently, this executable simply reads out the negotiated link speed for an Ethernet device. The corresponding result is advertised by the machine, and the machine's resource capacities are updated accordingly so jobs requesting more bandwidth than available cannot match.

When a job matches the machine and requests a bandwidth guarantee, the *condor_startd* will decrement the advertised available bandwidth by the requested amount. This is done in a greedy manner in HTCondor (similar to CPU-core and memory-based matching) and can result in an inefficient allocation of bandwidth. We do not currently tackle this issue further. Once all the bandwidth is allocated to jobs, HTCondor will not start additional jobs until a running one completes and releases the network bandwidth resource.

When setting up the Open vSwitch bridge, the virtual port connected to the external device is configured with the ap-

propriate QoS rate limiting (a built-in Open vSwitch feature). Linux's firewall, `iptables` is set up with a chain per job slot so all packets to and from the internal network device are monitored. The Linux kernel will count the packets and bytes that match each firewall rule. Furthermore, these counters are periodically polled to get the total bytes in and out of the job. This can be used to derive the average network rates, which are reported centrally to the HTCondor pool together with the total traffic information.

The Open vSwitch rate limiter will prevent the job from using more than the requested bandwidth. By crudely partitioning based on the negotiated link rate, we have a simple bandwidth management scheme. This approach has some limitation for network traffic that needs to go through a WAN as the bandwidth for the local area network is less likely to be constrained than the bandwidth available off-site for a cluster. To coordinate bandwidth availability across clusters, we need to interact with the network itself, which is explained in SectionVI.

Section VII goes into details about our set of experiments designed to evaluate the effectiveness of this and other bandwidth management features.

VI. Software-defined Networking Integration

Software-defined networking (SDN) techniques allow for a decoupling of the software control-plane from the data-plane of the network. Figure 5 shows a traditional data center network topology where virtual machines are bridged onto a hierarchical network, and each network device functions autonomously.

Through abstractions made available for SDNs (e.g. OpenFlow), it is now possible to manipulate the network programmatically from a centralized controller application. In addition, these same abstractions can be applied to virtual network switches running on individual hosts via Open vSwitch. Figure 6 on page 6 shows a simplified network topology where the network's control-plane is centrally administered by an SDN controller. This centralized control layer in software-defined networking opens the network up for innovations in scheduling and traffic management.

We have implemented a custom OpenFlow controller in order to centrally administer the Open vSwitch instance on each host as well as OpenFlow capable physical network devices. The controller receives a copy of the job and machine ClassAds when the job starts, allowing us to schedule the use of network resources and apply policy to the network dynamically from this central vantage point. This controller is based on the POX [18] controller with the addition of an HTCondor-specific module.

The *condor_starter* can invoke an arbitrary script at startup and shutdown of the job's network namespace. The script is given a copy of the job and machine ClassAds. The ads are sent to the controller listener through a TCP socket.

The controller maintains an in-memory database of ClassAds for all running jobs in the pool. The machine ad includes the job's IP address, allowing the controller to maintain a mapping between IP addresses and job ads.

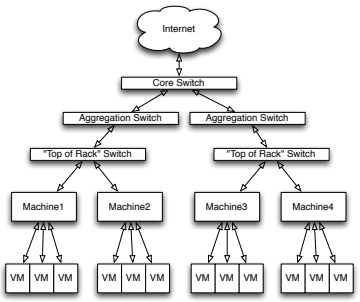


Fig. 5: Traditional data center network topology

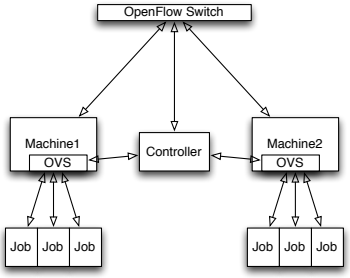


Fig. 6: Network topology with OpenFlow controller and Open vSwitch integration

In addition to the job ID, the network controller has access to all job attributes such as the user name or requested bandwidth. The controller behaves similarly to a Layer-2 switch. When the controller receives a *PacketIn* event from the switch it can look up a job ClassAd associated with the source or destination IP address. If a job is associated with the IP address, it can use the corresponding ClassAd and configured policy to determine an appropriate action. Otherwise, it will apply normal Layer-2 switching rules. In most cases, the OpenFlow controller will install a new flow rule in the switch so future packets to or from this job do not need to be handled by the controller. The controller's policies are meant to be heavily customizable via a configuration file.

With these network policies applied, the isolation among different HTCondor jobs is greatly improved.

Currently, the OpenFlow controller performs no routing actions; we have mostly focused on alterations of the existing Layer-2 network. We have designed a few sample policies to demonstrate the power in integrating the network with HTCondor. Current policies include:

- Drop all HTCondor job traffic for users specified in the HTCondor configuration file.
- Create a network slice per user; that is, drop any traffic between jobs owned by different users.
- Drop all network traffic going to the WAN port for a given set of users or, alternately, send the traffic to an alternate WAN port (for example, if those users have purchased access to a particular high-speed network).

Extension for Core Switch

Besides host-level network policing, we also implemented bandwidth management for HTCondor related network traffic at the core switch level. The core switch is the switch that connects the data center to the WAN. At the core switch, it is difficult to do per-job bandwidth management because of the large number of HTCondor job traffic flows going through the same interface. Instead, we would like to achieve bandwidth management for a bundle of traffic flows within the same HTCondor accounting group.

Currently, OpenFlow 1.0 does not provide a mechanism for manipulating QoS policies (such as dynamically creating bandwidth rate limiting); however, it allows the switch to specify which QoS policy to apply to a given packet. This, along with the third policy above (per-user WAN policy), can be used to apply specific WAN bandwidth shaping policies. A configuration file specifies the mapping between HTCondor accounting group and the QoS queue previously created by the network administrator. We hope to have our controller automatically create these queues when OpenFlow 1.3 hardware is more broadly available.

As WAN bandwidth is often the limiting factor, we believe this can provide effective bandwidth allocation and prioritization when the WAN resource is scarce. The mapping of HTCondor groups to QoS policy and the configuration of the QoS policy must be configured separately and manually in the OpenFlow controller and physical hardware. The detailed experimental results are presented in Section VII.

VII. Experiment Results and Evaluation

In this section, we demonstrate the effectiveness of our bandwidth management functionality, analyze the scalability and performance overhead brought by Lark compared with regular HTCondor software, and we also discuss a real-world use case which shows that Lark can provide easy fair sharing or prioritization on network resources in the multi-tenancy scenario.

A. Effective Bandwidth Control

1) Host Level Bandwidth Management: To demonstrate the functionality and utility of the host level bandwidth management, we have designed an experiment which highlights the new capabilities. The network and cluster topology is outlined in Figure 7. These jobs all share the worker node's available network bandwidth (1000Mbps to the local switch) and process files from an HTTP server.

We submit jobs from three users, A, B and C. Jobs from user A request 100Mbps of bandwidth and download from HTTP server A; jobs from user B request 300Mbps of bandwidth and download from HTTP server B; and jobs from user C request 600Mbps of bandwidth and download from HTTP server C. As each user has equal priority, one job from each can run at a time. After all three jobs (one from each user) start to execute on this worker node, no more new jobs can be assigned to it as there is no network resource available. These new jobs are either assigned to other available worker nodes or wait in the job queue until a worker node is available and provides enough bandwidth resource as the new jobs request.

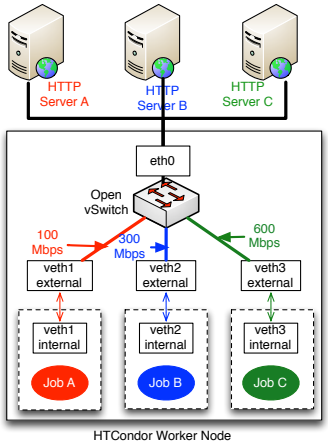


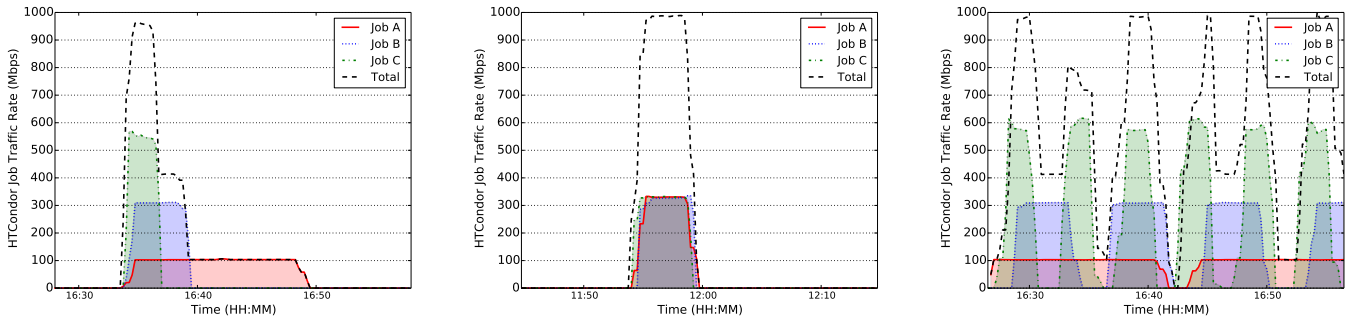
Fig. 7: Network topology configuration for host level bandwidth control

It is worth noting that the host level bandwidth management is ideal for network communication within a single cluster since the network is typically homogeneous. These HTTP servers are located on the same switch as the worker node.

Figure 8 shows the resulting network usage with bandwidth management enabled. We produce the plot by graphing the network traffic from the HTTP servers A, B and C (targetted to be 100, 300, and 600Mbps, respectively). Figure 8a shows the corresponding network rates during each job's execution. The incoming traffic rates of these jobs are restricted to their bandwidth requests. Figure 8b shows a second set of experiments, identical to the prior case except without bandwidth management. As TCP provides fairness and each of the three running jobs uses one TCP stream, we expect the jobs to split the bandwidth evenly - as demonstrated in the graph. Finally, Figure 8c demonstrates the network traffic generated by running several jobs of each type sequentially. As expected, the jobs stay within their bandwidth allocations. The spikes shown in the traffic graphs are due to sampling intervals and limitations of the QoS policy in Open vSwitch. Compared to Figure 8b, we demonstrate the ability to give jobs of type C a shorter execution time than what was previously achievable in HTCondor.

2) WAN Bandwidth Management: With the combination of HTCondor, network namespaces, and OpenFlow-enabled switches, we can classify flows on the network according to the attributes of the associated jobs. We use this to apply site policy to manage the scarce WAN bandwidth resource.

To demonstrate the usefulness of our techniques applied to managing WAN bandwidth, we have performed FTP transfers between Nebraska and Wisconsin test nodes as part of HTCondor jobs. These have similar network characteristics of grid jobs running with remote data access. The network topology of our test setup is illustrated in Figure 9. We use a high-end server node with Open vSwitch installed as the core switch, connecting the test nodes at Nebraska site to the WAN through Internet2 at a rate of 1Gbps. Using Open vSwitch, we create three QoS queues at the Ethernet port `eth0` that connects to the external network. *q0* is the default queue, which has the



(a) With host level bandwidth management (b) Without host level bandwidth management (c) Multiple jobs from each user

Fig. 8: Network traffic graph for HTCondor jobs with and without host level bandwidth management

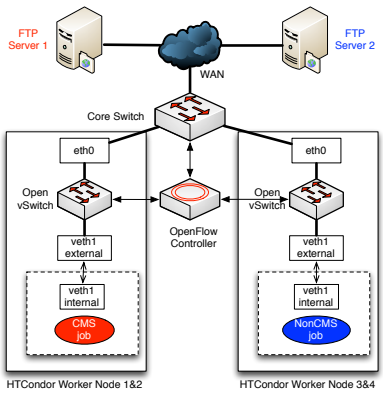
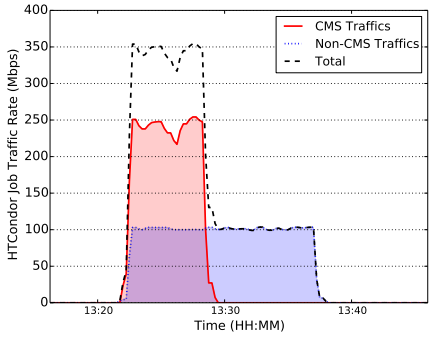
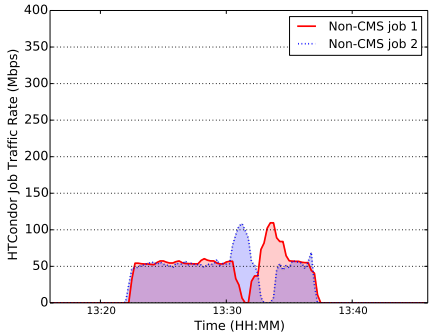


Fig. 9: Network topology configuration for WAN bandwidth control



(a) Aggregate traffic graph for CMS and Non-CMS jobs



(b) Non-CMS related network traffic graph

Fig. 10: WAN bandwidth control management for HTCondor

default maximum network bandwidth of that physical port. *q1* and *q2* are created to have network bandwidth 250Mbps and 100Mbps respectively. *q1* is for CMS traffic (the name of a local project which purchased a large local cluster whose jobs are considered high-priority) and *q2* is for lower-priority, non-CMS traffic. Two FTP servers at Wisconsin site are set up for file uploading. One is on *wisc-lark01*, the other is on *wisc-lark02* for CMS and non-CMS uploads, respectively. We submit four HTCondor jobs (two CMS jobs and two non-CMS jobs), each of which uploads a 5GB file to FTP servers. We indicate the type of job and FTP destination in the submission scripts to HTCondor. Figure 10a illustrates the corresponding aggregate network traffic graphs for these two groups. We can observe that the aggregate traffic obeys the networking policy as the high-priority jobs do not share the bandwidth equally with the low-priority jobs. Figure 10b further looks into the detailed bandwidth usage for each of the two jobs within the non-CMS group. It demonstrates that the jobs within the same accounting group behave as expected according to TCP fairness.

These experiments clearly demonstrate the ability to manage the network based on application-layer data. Without assigning each job its own address on the network or sending the application-layer data (the job's ClassAd) to the OpenFlow controller, we wouldn't have been able to differentiate the network flows by the owner in the HTC layer.

B. Performance Overhead

To provide network management, we have introduced several additional components such as Linux network namespaces, Open vSwitch on the host, the POX controller, and OpenFlow

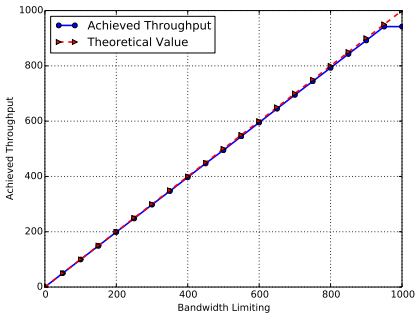


Fig. 11: Achieved throughput by Open vSwitch after bandwidth limiting

on the central switch. Each new component comes with a particular cost compared to plain HTCondor. We are particularly interested in:

- The overall overhead for setting up and tearing down private network space for each job and performing necessary networking configuration.
- Network performance degradation by using Open vSwitch.
- The overhead in the use of SDN techniques for network management on Open vSwitch.

1) Startup Overhead: We record the timestamps at the beginning and end of the batch job to calculate the extra overhead from creating and configuring the network namespaces (creating the virtual Ethernet device, adding it to the network bridge, and the Layer-2 configuration). Compared with batch a job running without Lark, the average extra overhead is about 1 second, including the time it takes for the virtual Ethernet device to perform network configuration with the DHCP server. By caching the DHCP release on the worker node between jobs, this overhead is reduced to 0.4 second. This extra overhead varies mostly with the time to recieve the DHCP offer from the server; the other kernel operations have little observed variability. As the typical HTC job execution time is measured in hours, we consider this overhead negligible.

2) Open vSwitch: Open vSwitch forwards packets between the namespaces in the kernel. When the QoS bandwidth limiting is applied, we are interested in the actual throughput that can be achieved between the virtual Ethernet device in the network namespace and the other hosts within the cluster. Using `iperf`[19] to do a 30 second transfer, Figure 11 shows the achieved throughput with different levels of bandwidth allocation.

The observed throughput is comparable to the configured bandwidth limiting. In practice, for CMS jobs, we have found that the bottleneck is more often the IO subsystem or the single-stream TCP performance over the WAN, not the Ethernet device.

Using the standard `ping` utility, we measured the end-to-end latency between end points with and without Open vSwitch. We found that the end-to-end latency difference is negligible.

3) OpenFlow Controller: In our setup, when a new flow is encountered on the OpenFlow switch, its packets are forwarded to the local controller until a new rule is installed. Again, by measuring latency with `ping`, we observe that the time taken to forward the packet, look up the corresponding HTCondor job, and install a new rule is approximately 50ms. After these initial packets, the extra latency is negligible.

4) Scaling Limitations: HTCondor clusters tend to run single-core jobs, meaning that a fully-loaded worker node can expect to have a network namespace and IP address per core. In addition to the startup and teardown times of each job being negligible compared to job runtime, we have observed no problems when running many jobs per worker node. Locally, the factor limiting the use of our techniques across the production clusters is a shortage of IPv4 addresses. Until IPv6 becomes more widespread across the grid and we implement IPv6 support in this software, we plan on NAT'ing the majority of jobs. Unfortunately, the wider network cannot distinguish flows by individual jobs in NAT mode, preventing us from applying a uniform network management policy.

The SDN controller is an additional level of complexity to the HTCondor system that the *condor_starter* must communicate with, reducing overall system stability. Finally, current OpenFlow hardware implementations have a finite limit on the number of rules (often around 4096). As each new job requires at least one rule, the hardware provides another bound on the number of jobs we can run.

C. Use Case Analysis

As WAN bandwidth is a scarce resource, it's possible for resource contention to lead to priority inversion. To illustrate the utility of bandwidth management, we have a real-world use case which compares high-priority CMS jobs running at Nebraska and uploading 1GB of output to a FTP server at Wisconsin with identical low-priority non-CMS jobs. Starting one CMS job and N non-CMS jobs simultaneously (varying N from 1 to 15; one TCP stream per job), we compare the speed of the high-priority job with bandwidth management to the same experiment without. When bandwidth management is enabled, we allocate 900Mbps bandwidth to the CMS job and 100Mbps bandwidth to non-CMS jobs. This bandwidth allocation scheme is to show that the CMS job is treated to be the high priority job with more resource allocation. When bandwidth management is not enabled, the CMS job and non-CMS jobs compete for network resources equally. Figure 12 illustrates the variations of average network throughput for the CMS job when the number of non-CMS jobs increases. When there is bandwidth management, the throughput for the CMS job is pretty stable and not affected by the increasing number of non-CMS jobs because all non-CMS job traffic has been forwarded to the low-bandwidth queue. Without bandwidth management, the throughput of the CMS job degrades accordingly when the number of non-CMS jobs increases. Table III shows the job execution speed up of the CMS job when using bandwidth management. This use case reveals that users can have better predictability with their submitted jobs in terms of execution due to the network resource allocation and isolation. Similarly, administrators can perform fair sharing and/or prioritization on network resources among different sets of users by simply modifying the relevant configuration file.

No. of non-CMS jobs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Execution Speed up	1.8	2.7	3.6	5.7	6.5	7.6	7.9	10.6	11.2	12.9	13.5	17.6	22.1	22.7	25.2

TABLE III: Job execution speed up for the CMS job from using WAN bandwidth management

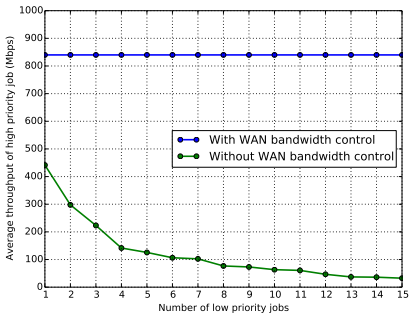


Fig. 12: WAN bandwidth management use case analysis

VIII. Conclusions and Future Work

There is a stubborn belief in the false dichotomy of batch computing versus resource provisioning. That is, in order to tightly control resources such as CPU, memory, or the network, cluster usage must be based on virtual machine technology. HTCondor has successfully used technologies from Linux containers to manage CPU and memory; the Lark project demonstrates that similar techniques can be applied to manage the host's network. By combining Linux network namespaces with OpenFlow, we can manage the cluster network resources, including WAN bandwidth. Our management techniques are more lightweight than virtual machines. Additionally, we can reuse the host's OS environment and the network namespace can be fully set up in less than a second.

The greatest impact of this work is permitting the network to associate application-level job attributes with network flows. While we mainly classified flows by the job owner, more subtle policies are possible. For example, we could also prioritize flows according to the job's priority for each owner. In the future, we hope to apply similar management techniques to other applications such as GridFTP. GridFTP transfers have a wealth of application data (current logged-in user, current file being transferred, etc.) that network policy could benefit from. We aim to integrate as many applications as possible until all network activities of a project on a cluster can be explicitly managed according to site policy. We acknowledge that the work is still in its preliminary state and the proposed network policies are simple, thus we plan to deploy the Lark code across our production clusters as OpenFlow hardware support matures to thoroughly assess the robustness and performance of the system at scale with more realistic workloads.

IX. Acknowledgements

We gratefully acknowledge funding from NSF ACI-1245864 in support of this work. This work was completed utilizing the Holland Computing Center of the University of Nebraska-Lincoln and the Center for High-Throughput Computing at the University of Wisconsin-Madison.

References

[1] R. Henderson and D. Tweten, "Portable batch system: External reference specification," NASA, Ames Research Center, Tech. Rep., 1996.

[2] W. Gentzsch, "Sun grid engine: Towards creating a compute power grid," in *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, ser. CCGRID '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 35–36.

[3] J. P. Jones, D. Lifka, B. Nitzberg, and T. Tannenbaum, "Cluster Workload Management," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, Oct. 2001.

[4] M. Jette, , M. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[5] "The worldwide LHC computing grid," http://home.web.cern.ch/about/computing/worldwide-lhc-computing-grid.

[6] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[8] "Linux containers," Available from http://linuxcontainers.org, 2014.

[9] "Openstack networking," Available from http://www.openstack.org/software/openstack-networking/, 2014.

[10] J. Basney and M. Livny, "Improving goodput by co-scheduling CPU and network capacity," *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 220–230, 1999.

[11] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX Association, 2011.

[12] D. Thain and M. Livny, "Parrot: An application environment for data-intensive computing," *Scalable Computing: Practice and Experience*, vol. 6, no. 3, pp. 9–18, 2005.

[13] M. Tsugawa and J. A. Fortes, "A virtual network (ViNe) architecture for grid computing," in *20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 98–109.

[15] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proceedings of the ACM SIGCOMM 2014 Conference*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 443–454.

[16] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[17] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[18] J. Mccauley, "Pox: A python-based openflow controller," Available from https://openflow.stanford.edu/display/ONL/POX+Wiki.

[19] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The TCP/UDP bandwidth measurement tool." [Online]. Available: http://sourceforge.net/projects/iperf/