SuperPub: An Efficient Tree Structured File Distribution System

Harrison Wong, Yu Cheng
Cornell University
{hcw29, yc489} @cornell.edu

ABSTRACT

In this paper, we look at various approaches to distribute large amounts of fairly large files within a datacenter. There are many use cases in which a file needs to be published from one node to many others. However, doing so in a timely, traffic efficient, and reliable manner is non-trivial. We propose a solution that establishes a tree like overlay structure mirroring a datacenter's topology to create and efficient multi-topic publish/subscribe system for file distribution within a datacenter. We minimize expensive cross-rack and cross-datacenter traffic, and provide robustness guarantees to handle failures.

1. INTRODUCTION

File distribution tasks in datacenter is a common task. A rapid file distribution system would provide significant improvement to VM provisioning[1] and binary roll-outs[2][3]. Historically, IP multicast and variations of BitTorrent were considered for this task, however each solution has its own problems. IP multicast relies on a large number of nodes in the network receiving the same data to be efficient. Moreover, multicast support itself is not typical in many cloud deployments. BitTorrent on the other hand, can have a large amount of inefficient traffic in large fanouts.

We introduce SuperPub, a solution that combines elements of both IP multicast and BitTorrent. SuperPub creates an overlay network in a similar fashion to BitTorrent, with a central controller that is analogous to BitTorrent's tracker. Its overlay network's logical topology is in the shape of a tree that mirrors the typical tree structure of switches found in a datacenter. Data is distributed from parent to child, and among peers, much like multicast. We form a hierarchy of levels corresponding to the nearness of nodes within each level, which allows us to minimize things such as cross-rack traffic and cross-datacenter traffic. SuperPub also includes a topic based subscription system that allows a client to subscribe to topics. A client will only receive messages from topics that it is subscribed to.

This work is to fulfill the project requirement for CS6452: Datacenter Networks and Services, Cornell University, Spring 2012.

2. DESIGN

2.1 System Components

SuperPub consists of two main components: a centralized controller and clients that subscribe to file updates.

2.1.1 Controller

To produce the global optimum that we are looking for, we need to hold a shared state between the different clients. We feel that this is best done by using a centralized controller to maintain this potentially complex state. This would be analogous to the tracker in BitTorrent that maintains state about the swarm. Just as in BitTorrent, all messages sent to this controller should be small and computationally inexpensive to allow for scaling.

The controller manages the state of what nodes are up and which are down, builds the physical topology layout using our datacenter structure discovery algorithm, and instructs each node the location of overlay topology. It also manages the topics that each node is subscribed to, and builds a overlay tree that is optimal for each topic in respect to the physical topology.

We do realize that the controller in this model becomes a single point of failure as the sole state maintainer, and can disrupt the distribution network. However, since the controller is not much different than a straightforward state machine, it would be very possible to establish a Paxos group using a commercially available solution such as ZooKeeper to maintain the state of the network to achieve higher fault tolerance guarantees. This would also allow the controller to service reads more efficiently as new publishers need to query the controller.

2.1.2 Clients

Each node starts a different client for each topic that it wishes to subscribe to. The client establishes connections to other clients depending on the location information given by the controller.

Each client in the tree is structured such that it has a parent, and potentially peers and children. The root client has no parent. The leaf clients (at the bottom of the tree) have no children. There are also cases where a client may have no children, such as the case when it is the only client in the rack. The detailed rules for overlay network construction will be discussed in the Overlay Topology section below.

2.2 Design Considerations

SuperPub aims for fast and efficient distribution of large files in a tree like manner with reliability guarantees. This is optimized for traditional datacenter networks where servers are typically connected using multiple tiers of switches. Because of the unique layout of a datacenter, we devised a set of considerations for our implementation of SuperPub system: We need to efficiently use network resources, achieve high throughput distribution and low latency, and have strong reliability guarantees.

2.2.1 Network Efficiency

To achieve our goal of network efficiency, we looked a typical datacenter layout with top-of-rack switches, aggregation switches and core switches. It is easy to see that bandwidth becomes more constrained near the higher levels of the hierarchy, while bandwidth within a rack has lower cost. This naturally leads us to design our system so that most traffic should reside inside a rack, and we would attempt to minimize cross-rack traffic, and further minimize cross-cluster or cross-datacenter traffic.

2.2.2 High Throughput and Low Latency

We attempt to achieve high throughput across the overlay network. This requirement leads to the same goals as for network efficiency as more utilization for the network within a rack and less utilization in between racks.

We also want our system to be very low latency compared to our competitors such as BitTorrent. We recognized that BitTorrent suffers from a relatively high startup cost, with tracker setup, distribution of a full file hash to peers and significant ramp up delay. In an effort to combat these, we propose an overlay network that is initialized prior to sending or generating data to be sent. This allows us to set up some of the overhead that we need in a non-time-constrained fashion.

To achieve high throughput and low latency, some software tricks as pipelining are also useful, since it allows for data to be pushed in periods of latency.

2.2.3 Reliable Sending

TCP itself has an end-to-end property to ensure that data gets delivered. This is done by sending ACKs to verify that the packet was received. Building an overlay network in this fashion however breaks this end-to-end property from the publisher to the subscribers. To rectify this, we need to reimplement this functionality to ensure that if a message gets dropped, it is resent in a timely manner.

2.3 Proposed Solutions

To address the considerations mentioned above, we propose the following solutions.

2.3.1 Datacenter Topology Finding

Before creating any overlay network, it is important to discover the datacenter topology.

We have two options for discovering the topology in our implementation. The controller could read in a file that stores the physical topology, which is used in cases when network administrator has the topology ahead of time; it is also useful in debugging and testing. Alternatively, we also have a physical topology finder that determines the distances to other nodes in runtime. This is done by measuring the time-to-live (TTL) timeouts between source and target. Using this, it would be possible to determine whether two nodes are in the same rack, cluster or datacenter in a simple fashion, and a physical topology profile can be created dynamically. A hybrid approach is also support for the system, where the administrator can specify only a part of the datacenter topology.

A client joining the network has two phases, a prejoin phase and a join phase. In the prejoin phase, the client notifies the controller of its name, IP and port. The controller then responds with a list of other nodes to find the distance to. If the controller already have physical location data of the client, it returns an empty list and the client goes directly to the join phase.

The list of nodes that a client is given in the prejoin phase are pinged in succession with increasing TTL timeouts. If a ping times out, we increase the TTL of the next ping and try again. The client does this for every node in the list, and returns a list of the minimum TTL setting where it is able to contact the corresponding node in the given list. The controller then interprets the results to determine the physical location of the node.

The client starts by setting the TTL to 2. If the client is able to ping a node with a TTL of 2, we know that the client and node are at least within the same rack. If it is unable to get a response from the remote node with a TTL of 2, it tries again with a TTL of 4. If it gets a response with this level, they must only have one switch between their racks. We can continue this to establish different levels of hierarchy between nodes by their network distance to each other.

The controller takes this list of network distances and determines where to place the node in the physical topology. It tries to place it in the same rack as an existing node. If it is unable to, it will create a new rack in the topology to place it.

Notice that with this approach, with a datacenter of n nodes, a node needs to ping on average $n/2$ other nodes in the topology discovery phase. However, instead of returning a plain list of previously seen nodes, the controller could instead return the current known physical topology, and a list of nodes that has not yet been inserted into the topology. With that information, client could figure out the closest node by discarding roughly $2/k$ of the remaining nodes in the topology tree with each ping, resulting $\log(n) + C$ number of pings in total, where k denotes the branch factor across all switches. This is possible since each ping will accurately reflect the common ancestor between the source and destination, and we would have probability $(k-1)/k$ of discarding $1/k$ nodes, and $1/k$ of discarding at least $(k-1)/k$ nodes.

This optimization is safe as it will always produce the correct result, and would save traffic during prejoining phase by a significant portion. Since the controller necessarily holds the physical topology regardless, it does not need to encode any data neither.

2.3.2 *Overlay Topology*

After acquiring the physical location of a node, the controller then adds the node to the overlay network for the corresponding topic it asked for, and returns the location information back to the node. This information consists of a node's parent, and optionally its peers and children.

The controller builds the overlay network in the following fashion: The overlay topology will essentially be superimposed onto the real physical topology. For each of the intermediate switches in the physical topology, the controller picks a random machine from all its decedents, and place it as the logical node for the switch. We call this process "promotion".

This approach would build a natural tree rather efficiently, but there is possibility of interweaving nodes from the same rack, that is, a node might become a non-intermediate decedent of a node from the same rack (A->B->C, where A and C are from the same rack). If that is the case, any file pushed through this channel will need to go into the rack at least twice, causing unnecessary traffic. Also, it is non-trivial how to pick a replacement node in case of failures.

Due to these constraints, instead of trying to either solve the hard global optimization problem incurring a rather huge reconfiguration cost in failure cases, or potentially withstand a bad local solution, we propose a slightly less efficient topology maintaining the property that any block can only enter the rack at most twice. It also has the nice property that it is easy to implement, and has relative high resource utilization.

We introduce the concept of "reserve node", where the m is a reserve node for node n in a given overlay topology if m is in the same rack as n, and is also subscribed to the topic of this overlay topology. The "reserve pool" for a node is the collection of reserve nodes for that node.

Following the same procedure as before, where we select some decedent of the switch to "promote" and take the switch's position in the overlay topology. But during "promotion", we also take the reserve pool of that promoted node and elevate it up as its children. Specifically, we would take all remaining nodes in the same rack that are subscribed to this topic, and make a branch for them. For example as in Figure 1, node 1,2,...,8 are in the same rack, and 9,10,11,12 are in a different rack. When 1 is promoted as the root of the tree, nodes 2,...,8 are elevated to one child of the node 1. In a similar fashion, 10, 11, 12 are elevated to the children of 9.

We will see that with the round-robin fashion of publishing blocks described below, a blaock cannot enter the rack more than twice. In fact, a block would enter any rack on average at most $(2k-1)/k$ times where k again denotes the branch factor across all switches. This topology also enables us to easily find replacement nodes with minimum impact on the overlay network, and handling nodes joining dynamically. These topics will be described later in the Node Failure Handling section.

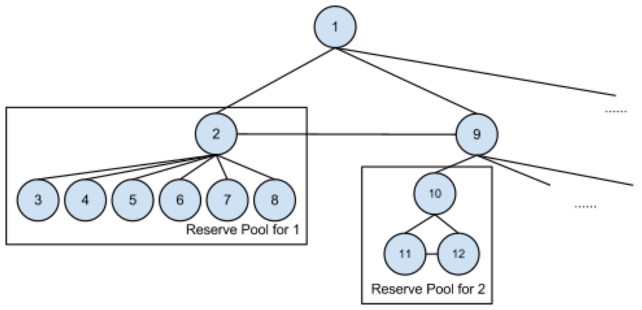

Figure 1: *Overlay Topology: Demonstration of reserve nodes placement*

2.3.3 *Publishing files*

To publish a file to the network, a publisher will first query the controller to find the root node. The publisher can then send files to this node in one of two ways. Firstly it can send messages in an asynchronous manner which simply blocks until the file is initially sent, or it can send it in a synchronous manner, which waits until the entire tree acknowledges the recipient of the file.

By the construction of our overlay network, a node will receive data from two sources: both its parent and its peers. If it receives a block from its parent, it replicates that block to all its peers, and sends the block to a single child. If it receives a block from its peers, it just sends it to a single child. The node selects which child to send in a round-robin fashion, in order to fully utilize the richer peer links instead of parent links.

2.3.4 *Node Failure Handling*

When a working node discovers that another node has failed by seeing a disruption in the connection between them, it complains about that node to the controller. The controller will aggregate information, and if a certain node is complained by a lot of other nodes, it will remove the failed node from the network and reconfigure the state of the overlay network. It does this by promoting a child from the reserve pool of the failed node to take the place if needed, or simply removing a leaf node from the network entirely.

When the reserve pool is empty, the controller knows that the whole rack lost interest in the topic, and will then simply promote a leaf node from another rack to take over. Notice that during this process, the overlay tree properties still need to be satisfied, namely, the reserve pool of that node will also need to be moved. However, this process do not need to break existing inter-rack connections, as to nodes in the rack, only the leader changes its parent and peers, all other connections and peer relationships are kept the same.

2.3.5 *Topics*

We introduces the concept of topic, common in the pub/sub domain, where users could subscribe to multiple topics, and would only get updates published to his interested topics. This is implemented as building multiple logical overlay networks on top of the same physical topology.

Creating separate overlay networks for different topics allows us to not have to burden nodes who do not need the blocks for a different topic. Sharing the same physical topology data structure allows us to only discover the topology once.

When a node wishes to join multiple topics, it simply starts multiple clients on the same node. The controller recognizes the node uniquely and the node does not have to go through the prejoin phase again. The controller then assigns each client to the corresponding overlay networks according to the topics.

2.3.6 Reliable Sending

Reliable sending requires a way to acknowledge that data has been sent to the destinations. We reimplement the TCP based acknowledgements as a tree based ACK, where each node ACKs only if all of its children have already ACKed. This places each node as the responsible party for its children, and ensures that when the final ACK is delivered to the root node, the entire tree has received the file already.

By using this system of ACKs, we can ensure that blocks do not get lost. If a message does not get ACKed within a certain timeout, the parent node will resend the file to the child missing the block until it gets it. If the child node does not respond to the ACK due to failure, the parent node will complain to the controller, wait for a reconfiguration, and resends the data to the new child node instead. If there is no new child node (current node is already a leaf node) it marks it as sent and continues.

3. EVALUATION

3.1 Implementation

We implemented the proposed design using Python 2.7. Network communication, RPC and data transferring are handled using Apache Thrift. The whole code base is roughly 1300 lines of code, excluding thrift schemas. The full implementation can be found at `https://github.com/HarrisonW/SuperPub`.

3.2 Setup

Running multiple instances of the same client would essentially allow for infinite bandwidth across the loopback interface if they are on the same computer. Because of this, we decided to run a series of VMs on two different machines to achieve isolation properties among network interfaces between the different VMs.

We set up our VMs using VMware Workstation 8.0.3 running on Windows 7 x64. Each VM is set up with 1 CPU core, 512MB of ram and a 100Mbit virtual bridged interface. We connected the physical computers running the VMs to a dedicated 100Mbit router to allow for data to be routed to both computers. The router is running DHCP which allows it to assign a unique internal IP to each VM and VMMs. These IPs are in the same subnet.

We tested data transfer between the VMs on the different physical computers to verify the proper network characteristics were met. Data transfer between VMs on the same machine took place at 99% of the 100Mbit virtual network interface, with no network traffic flowing through the router. Data transfer between VMs on different machines took place at 99% of the 100Mbit virtual and physical network interfaces. This allows us to simulate each of our physical computers acting as a rack of machines attached to a top-of-rack switch, with a core switch connecting the two racks. In our tests, CPU or memory usage never became a bottleneck, which validates the effectiveness of testing using VMs.

Figure 2 demonstrates the physical setup of our simulation network. Figure 3 demonstrates the overlay topology SuperPub will build on top of the physical topology.

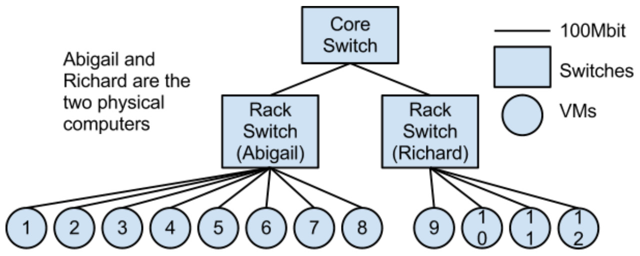

Figure 2: *Evaluation setup: Machines and connections*

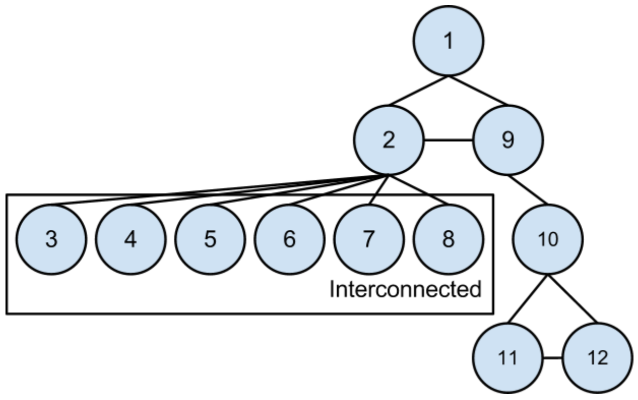

Figure 3: *Evaluation setup: Logical topology*

3.3 Evaluation Results

We began by sending 200MB files over topology described above, varying the block size for each send. We then simulated node failures in the network, and measured how fast SuperPub reacts and changes its overlay network to adjust for this failure.

3.3.1 Propagation Time

We measured and plotted the time between publishing a new file and all nodes in the overlay receiving the file. Figure 4 demonstrates this propagation time with respect to block size. We see that the time is increased when the block size is too large or too small. This makes intuitive sense because when we have smaller blocks, the time spent on processing overhead of the blocks and sending ACKs becomes more significant. On the other hand, as the block size gets larger, pipelining will not work as effectively. We believe that under our network setup with 100MBits links, 1MByte block size is ideal. As the bottleneck of the system is at the network level, we should be able to scale up the block size as we have

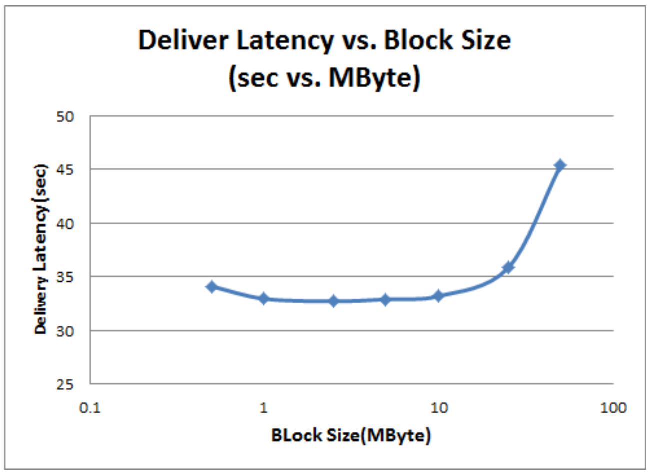

Figure 4: *Evaluation result: Message delivery latency of 200MB file of various block sizes*

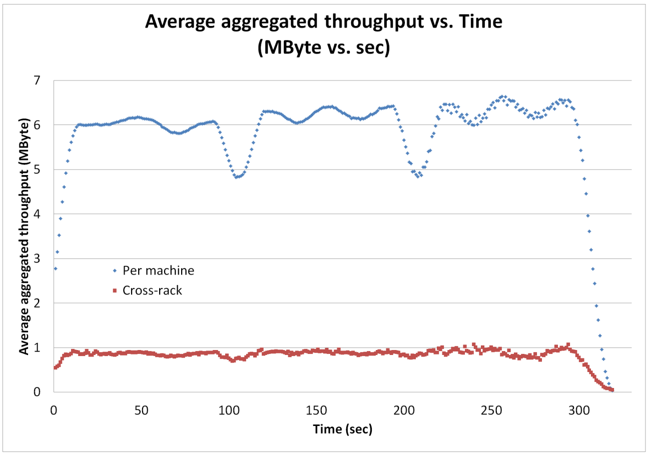

Figure 6: *Evaluation result: Aggregated average throughput per machine and cross-rack throughput during delivery of 200MB file with 0.5MB, 1MB, and 2MB block sizes.*

higher speed network links. (Notice that for binary rollouts, it might also be beneficial to limit the bandwidth to avoid interference to the regular workload, and under those circumstances, dynamically adjusting the block size when distributing the file would be a more optimal solution for distributor)

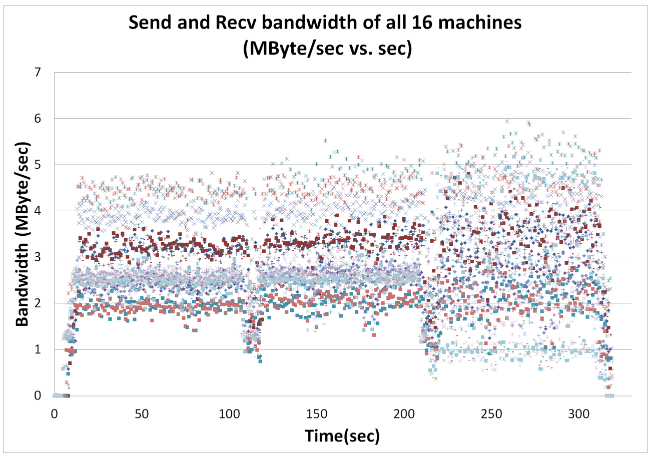

Figure 5: *Evaluation result: Bandwidth per machine during delivery of 200MB file with 0.5MB, 1MB, and 2MB block sizes.*

3.3.2 Overall bandwidth of the system

We measured and plotted the send and receive bandwidth for each node, and plotted them onto the same figure. Figure 5 shows the send and receive bandwidth. This data is taken during sending with block size 0.5MB, 1MB and 2MB with each series representing either the sending or receiving bandwidth of an individual node. From the graph we notice that with lower block size, the bandwidth across system is more stable: Most nodes in the same level share similar bandwidth consistently, where the nodes higher up in the overlay topology tree have in general lower bandwidth as they have less connections. As the block size goes higher, there are more variability across bandwidths between nodes.

3.3.3 Average bandwidth and cross-rack bandwidth

We also measured and plotted the overall aggregated bandwidth for the whole system and averaged over 12 nodes. This is done by taking the previous send and receive bandwidth for each node, summing them together, dividing them by 2 to compensate for duplications, and dividing it by 12 to get the average. On the same graph, we also plotted the cross computer traffic in our set-up, modeling the cross-rack traffic in a real scenario. Figure 6 demonstrate this average overall aggregated bandwidth and cross-rack bandwidth.

The average bandwidth shows noticeable peaks and pits in each distribution cycle. But on the same plot, we notice that a dropping in average bandwidth commonly occurs during high cross-rack bandwidth. And thus we suspect the pits in the aggregated bandwidth are caused by the system facing a bottleneck on cross-rack data transfers.

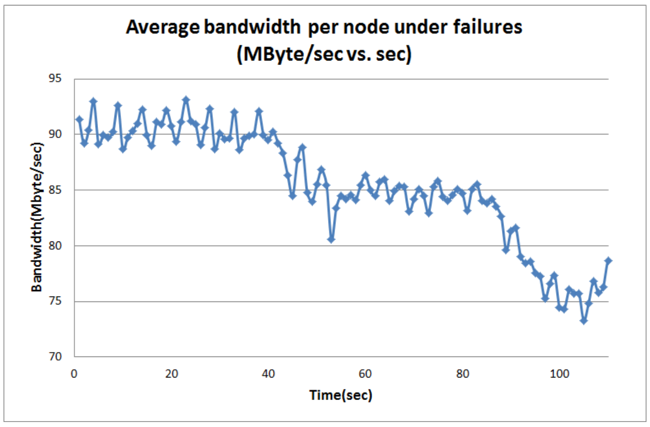

Figure 7: *Evaluation result: Aggregated average throughput in node failure cases. One leaf node failed at time 40, and one non-leaf node failed at time 80.*

3.3.4 Bandwidth in node failures

We measured and plotted the overall aggregated bandwidth for nodes under failures. Figure 7 demonstrate this total throughput. In the beginning of the experiment, we allow the system to converge to a steady state, with roughly 7.6 MBytes/sec bandwidth per each node. We manually killed a leaf node at time 40, and a non-leaf node at time 80.

We notice that at time 40, the aggregate bandwidth suddenly drops to 80 MBytes/sec, but recovers relatively fast to 85 MBytes/sec, which is roughly same 7.6 Mbytes/sec for 11 nodes, as the controller reconfigures its parent and peers to cease distributing to the node. At time 80, we also see the bandwidth dropping, but it recovers slower than in the case of leaf node failure, as more nodes need to be reconfigured to adjust for the changes. Eventually, the aggregate bandwidth converges to 76 Mbytes/sec, which is the normal bandwidth for 10 nodes system.

3.3.5 Corrupted blocks and resend

We also verified that the resend functionality works when the clients receive a corrupted block. As a future direction, it would be beneficial for the controller to automatically adjust the wait time to resend a block when ACKs are not received within that time frame. This might involve optimizations using the network topology, historical communication patterns, and block sizes. For all the experiments we performed, the resend logic was not invoked.

4. RELATED WORK

There had been various approaches regarding file distribution in a datacenter set-up. Traditionally, people tried single distributor model [3], where clients would pull from the single distributor for updates. This inevitably causes a severe traffic bottleneck on the distributor, and introduces the distributor as the single point of failure of the system. Due to the undesirable scaling properties, it is rarely used in any large datacenter nowadays.

More recent approaches apply P2P concepts into the model. A well known and used system is BitTorrent. SystemImager, a rapid VM provisioning toolkit for high performance computing cluster, uses a straightforward BitTorrent system for distributing VM images[1]. Twitter introduced Murder[3], a BitTorrent system for roll-outs with simple optimizations, and claim to have a 200x speed boost compared to the traditional single distributor approach. Facebook also applied similar BitTorrent concepts, but introduced rack-awareness in the tracker, and thus had better data locality than normal BitTorrent systems[2].

Multicast was also studied for this scenario. There is a lack of support and reliability to multicast, which causes it to be overlooked often times in datacenter setups or even disabled. Multicast has been requested on Amazon's EC2 since at least 2007[4] but is still not enabled today. There have been attempts to use multicast for smaller scale using UDP Cast[5] for example. There are other attempts to make multicast tractable in a datacenter by limiting its use to cases where it is needed the most such as in Isis2[6]. However, such attempts have yet to catch on, and it is rare for a system to implement multicast effectively.

We based our system off of a publish/subscribe system with a simple interface much like Redis[7]. RabbitMQ has a good outline of potential configurations of a publish/subscribe system[8]. Redis uses a topic based publish/subscribe system where clients can either publish or subscribe to a topic on the Redis server. However, Redis has a single point of failure with the server, and all load is directed to the single server. There is work on Redis Cluster[9] that aims to correct this single server model using bloom filters and replication for the pub/sub system, however it is not production ready.

Amazon has allowed for public use its Simple Notification Service[10] (SNS). SNS is a scalable publish/subscribe messaging system. However, it does not seem to be able to scale well to larger message sizes. It is designed to send small messages like text messages (SMS) to phones, emails or over HTTP.

5. CONCLUSION

During our construction of SuperPub we have encountered many problems that we did not foresee. To make it model the tree structure of the network, we had to do a system of node promotions where nodes act like a switch in the network. To handle failures and dropped data, we had to add a system of acknowledgements. The state finding algorithm in the controller is moderately complex to account for different network configurations. Despite these problems, we feel that SuperPub is a working piece of software that can rapidly distribute files from a publisher to many subscribers in a datacenter. Our evaluation benchmarks demonstrate a high rate of throughput from using the system, while utilizing the constrained cross-rack links as little as possible.

6. ACKNOWLEDGMENTS

We would like to acknowledge our professor Papa Gün (Emin Gün Sirer) at Cornell for his help with this project and general awesomeness. His class *CS 6452: Datacenter Networking and Services* was invaluable to our understanding of current work in the field.

7. REFERENCES

[1] "Systemimager v4.1.6 manual." `http://www.systemimager.org/documentation/systemimager-manual-4.1.6.pdf`, Feb 2008.
[2] Paul, "Exclusive: a behind-the-scenes look at facebook release engineering." `http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/`, Apr 2012.
[3] Gadea, "Murder: Fast datacenter code deploys using bittorrent." `http://engineering.twitter.com/2010/07/murder-fast-datacenter-code-deploys.html`, July 2010.
[4] jaydsi, "Multicast support on ec2?." `https://forums.aws.amazon.com/message.jspa?messageID=52742`, Jan 2007.
[5] "Udpcast." `http://www.udpcast.linux.lu/`, Dec 2011.
[6] Birman, "Isis2 cloud computing library." `http://isis2.codeplex.com/documentation/`, 2012.
[7] "Pub/sub redis." `http://redis.io/topics/pubsub/`.

[8] “Rabbitmq.”
`http://www.rabbitmq.com/getstarted.html`, 2012.

[9] “Redis cluster specification (work in progress).”
`http://redis.io/topics/cluster-spec/`.

[10] “Amazon simple notification service (amazon sns).”
`http://aws.amazon.com/sns/`.