2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing

Dynamic Load-Balanced Multicast for Data-Intensive Applications on Clouds

Tatsuhiro Chiba*‡, Mathijs den Burger†, Thilo Kielmann†, and Satoshi Matsuoka*§

*Dept. of Math. and Computing Sciences
Tokyo Institute of Technology
2-12-1 Oookayama Meguro-ku Tokyo, 152-8550, Japan
Email: chiba-t@matsulab.is.titech.ac.jp, matsu@is.titech.ac.jp
†Dept. of Computer Science
Vrije Universiteit
De Boelelaan 1083 1081 HV Amsterdam, The Netherlands
Email: {mathijs, kielmann}@cs.vu.nl
‡Research Fellow of the Japan Society for the Promotion of Science
§National Institute of Infomatics

***Abstract*—Data-intensive parallel applications on clouds need to deploy large data sets from the cloud's storage facility to all compute nodes as fast as possible. Many multicast algorithms have been proposed for clusters and grid environments. The most common approach is to construct one or more spanning trees based on the network topology and network monitoring data in order to maximize available bandwidth and avoid bottleneck links. However, delivering optimal performance becomes difficult once the available bandwidth changes dynamically.**

In this paper, we focus on Amazon EC2/S3 (the most commonly used cloud platform today) and propose two high performance multicast algorithms. These algorithms make it possible to efficiently transfer large amounts of data stored in Amazon S3 to multiple Amazon EC2 nodes. The three salient features of our algorithms are (1) to construct an overlay network on clouds without network topology information, (2) to optimize the total throughput dynamically, and (3) to increase the download throughput by letting nodes cooperate with each other. The two algorithms differ in the way nodes cooperate: the first 'non-steal' algorithm lets each node download an equal share of all data, while the second 'steal' algorithm uses *work stealing* to counter the effect of heterogeneous download bandwidth. As a result, all nodes can download files from S3 quickly, even when the network performance changes while the algorithm is running.

We evaluate our algorithms on EC2/S3, and show that they are scalable and consistently achieve high throughput. Both algorithms perform much better than having each node downloading all data directly from S3.

***Index Terms*—Cloud Computing, Multicast Algorithm, Amazon EC2/S3, Work Stealing**

I. Introduction

Computing cloud platforms such as Amazon Web Services [1], Windows Azure [2], and others provide great computational power that can be easily accessed via the Internet at any time without the overhead of managing large computational infrastructures. Traditionally, HPC applications have been run on distributed parallel computers such as supercomputers and large cluster computers. Especially data intensive HPC applications, such as BLAST [3], require very large computational and storage resources. For example, the ATLAS experiment [4], which searches for new discoveries in the head-on collisions of protons of extraordinarily high energy, will typically generate more than one petabyte of raw data per year. In addition, replicas of these data and derived simulation results will increase the data footprint even more. It is difficult for one domain or organization to manage and finance the resources to store and analyze such amounts of data. While clouds are currently mainly used for personal and commercial use (e.g. for web applications), their large number of storage and computational resources, high accessibility, reliability and simple cost model make them very attractive for HPC applications as well.

Parallel HPC applications often need to distribute large amounts of data to all compute nodes before or during a run. In a cloud, these data are typically stored in a separate storage service. Distributing data from the storage service to all compute nodes is essentially a multicast operation. The simplest solution is to let all nodes download the same data directly from the storage service, but that can easily become a performance bottleneck that dominates the total execution time of an application.

Many optimization methods and algorithms for multicast operations have been developed for different environments. One example are high-performance collective communication algorithms for parallel distributed systems (e.g. clusters and grids). Another example is data streaming and file sharing on P2P systems. Each of these approaches essentially performs a multicast operation, but has to take different assumptions and settings into consideration depending on the target environment.

While cloud platforms are similar to parallel distributed systems, they also share some characteristics with P2P systems. First, cloud computing services generally provide virtualized computing environments that are shared with other cloud users. The available bandwidth within a cloud can therefore change dynamically. Moreover, the underlying physical network topology and activity of other users is generally unknown. Multicast optimization methods used in parallel distributed systems will therefore be less applicable to clouds, and will result in sub-optimal performance.

In this paper, we present two efficient algorithms to distribute large amounts of data within clouds. The proposed algorithms combine optimization ideas from multicast algorithms used in parallel distributed systems and P2P systems to achieve high performance and scalability with respect to the number of nodes and the amount of data. Each algorithm first divides the data to download from the cloud storage service over all nodes, and then exchanges the data via a mesh overlay network. Furthermore, the second algorithm uses work stealing to automatically adapt the amount of data downloaded from the cloud storage service to the individual throughput of the nodes.

We have implemented the two proposed algorithms and evaluated their performance on a real cloud environment (Amazon EC2/S3). As a result, we have confirmed that proposed algorithms achieve high and stable performance.

In Section II we briefly discuss existing multicast methods for parallel systems, clusters, grids, and clouds. Section III outlines our target platform, Amazon's EC2/S3 clouds. Section IV presents our algorithms for multicasting in cloud environments, before we analyze their performance in Section V. Finally, Section VI summarizes our findings and outlines directions of future work.

II. Related Work

This section first characterizes various multicast methods for large amounts of data on parallel distributed systems and P2P networks, and then explains the specific issues with optimizing multicast operations on clouds.

A. multicast on parallel distributed systems

For parallel distributed systems, such as clusters and grids, there are many type of optimization methods for collective operations. In particular, optimization of multicast communication has been researched in message passing systems like MPI and their collective operation algorithms [5][6][7]. Target applications in this environment are mainly HPC applications, so these optimization techniques focus on making a multicast operation as fast as possible. The optimizations of multicast communication for parallel distributed systems makes several assumptions:

1) network performance is high and stable
2) network topology does not change
3) available bandwidth between nodes is symmetric

Based on these assumptions, optimized multicast algorithms generally construct one or more optimized spanning trees by using network topology information and other monitoring data [8][9]. The data is then forwarded along these spanning trees from the root node to all others. These multicast techniques are therefore sender-driven (i.e. *push-based*). For large amounts of data, some optimization algorithms try to construct multiple spanning trees that maximize the available bandwidth of nodes. The data is then divided into small pieces that are transferred efficiently to each node by using the 'pipelining' technique [10][11]. For example, Stable Broadcast [11] uses depth-first search to find multiple spanning pipeline trees based on estimated network topology information [12] of multiple clusters, and maximizes available bandwidth of all nodes by reducing the effect of slow bandwidth nodes.

B. overlay multicast on P2P systems

Optimization of multicast communication is also studied within the context of P2P overlay networks. Examples include file sharing applications like BitTorrent [13] and data streaming protocols like SplitStream [14]. The target environment of these applications differs from parallel distributed systems:

1) network performance is very dynamic
2) nodes can join and leave at will
3) available bandwidth between nodes can be asymmetric

Consequently, the main focus of multicast communication protocols on P2P systems is being robust. They divide the data to multicast into small pieces that are exchanged with a few neighbor nodes. All nodes tell their neighbors which pieces they have and request pieces they lack. Multicast communication in P2P networks is therefore receiver-driven (i.e. *pull-based*).

Receiver-driven multicast can also improve throughput. For example, MOB [15] optimizes multicast communication between multiple homogeneous clusters connected via a WAN. The MOB protocol is based on BitTorrent, and takes node locality into account to reduce the number of wide-area transfers. Nodes in the same cluster are grouped into *mobs*. Each node in a mob steals an equal part of all data from peers in remote clusters, and distributes the stolen pieces locally. This way, each piece is transferred to each cluster only once, which greatly reduces the amount of wide-area traffic compared to BitTorrent. The incoming data is also automatically spread over all nodes in a mob, which works very well when the NICs of the nodes are the overall bandwidth bottleneck instead of the wide-area links.

C. multicast on clouds

Cloud services that provide virtualized application execution environments (e.g. Amazon EC2/S3) are becoming increasingly popular as a platform for HPC applications. For example, Deelman *et al.* [16] tried to evaluate the cost of running data-intensive applications on Amazon EC2/S3 in terms of different execution and resource provisioning plans. Their target application is Montage [17], which is a portable software toolkit for science-grade mosaics of the sky by composing multiple astronomical images. In this case, if users continuously use and access the data sets stored on cloud storage many times, it is good to store the generated mosaic, because the storage costs are cheaper than the CPU and the data transfer costs. Some works have shown the calculation performance of Amazon EC2 by using LINPACK and NAS Parallel Benchmark [18][19]. In these papers, it was reminded that it is unsuitable for communication-intensive applications since EC2 network performance is lower than other existing cluster systems.

Some works tried to evaluate basic performance and cost problems, and also revealed whether data-intensive scientific applications were suitable for clouds. For example, Garfinkel [20] evaluated the performance of EC2, S3 and Simple Queue Service (SQS) for a period of several months, focusing on S3's read and write performance from EC2 nodes, as well as on data availability and transactions per second for different objects sizes. In [20], it was indicated that S3 certainly provides a scalable and highly available service, however with variable performance, causing users to manually tune communication parameters like object sizes and numbers of simultaneous connections.

Palankar *et al.* [21] evaluated whether Amazon S3 is a feasible and cost effective alternative storage platform for science grids, based on experiments similar to the ones in [20]. It was found that S3 currently does not satisfy all requirements of scientific applications due to a lack of a usage model in which costs, data availability, durability, and accessibility can be taken into account simultaneously.

Cloud systems are based on large clusters in which nodes are densely connected, as opposed to P2P environments in which nodes are sparsely connected. Contrary to traditional clusters, the computational and storage resources provided by clouds are fully or partly virtualized. A multicast algorithm for clouds can therefore not assume anything about the exact physical infrastructure.

Similar to P2P environments, the network performance within clouds is dynamic. The performance of the uplink and downlink of a virtual compute node can be affected by other virtual compute nodes that are running on the same physical host. Routing changes and load balancing will also affect network performance. Furthermore, the data to distribute is often located in the cloud's storage service, which introduces another potential shared bottleneck.

Existing multicast solutions for parallel distributed systems, like the pipelined trees described in Section II-A, rely on monitoring data and topology information or a map of the available to optimize data transfers for the current network conditions. However, obtaining such information is tedious and hard, and keeping it up-to-date is almost impossible. Our multicast algorithms therefore apply solutions commonly used in P2P environments to handle the dynamic network performance in clouds. These solutions are much easier to deploy and more reactive to changes in network performance [22].

III. Amazon EC2/S3

Our target cloud environment is Amazon EC2/S3. In this section, we will first describe Amazon EC2/S3 and define our cloud usage model for data-intensive HPC applications. Second, we will evaluate the performance of Amazon EC2/S3 through some preliminary experiments and discuss our findings.

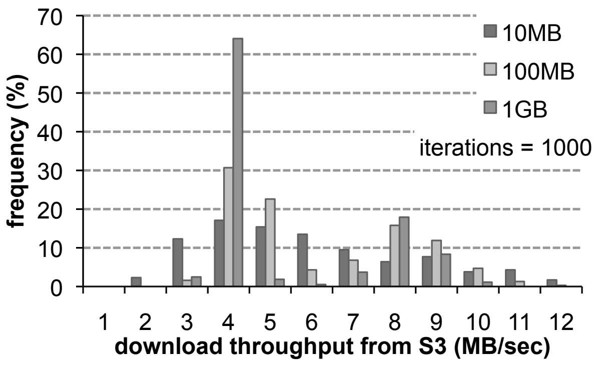

Fig. 1. download throughput frequency from S3 to one EC2 node

A. characteristics

Amazon Elastic Compute Cloud (EC2) and Simple Storage Service (S3) are cloud services provided by Amazon Web Services (AWS) [1]. Amazon EC2 provides virtualized computational resources that can range from one to hundreds of nodes as needed. The provided resources are virtual machines on top of Xen, and called *instances*. A user first chooses an OS image (called an Amazon Machine Image or AMI) that is stored in S3, and selects the type of instance to run the image on. Various instance types exist, with varying CPU power, memory size etc. [23]. Second, a user selects where to run the instances. All instances will then be booted immediately and become active and usable within several tens of seconds.

Amazon S3 is a cloud storage service that can be accessed via the Internet. Files can be uploaded and downloaded via standard GET, PUT and DELETE commands over HTTP or HTTPS that are sent through a REST or a SOAP API. S3 stores files as *objects* in a unique name space, called a *bucket*. Buckets have to be created before putting objects into S3. They have a location and an arbitrary but globally unique name. The size of objects in a bucket can currently range from 1 byte to 5 gigabytes. The S3 API allows users to access a whole object or a specific byte range of it. For example, one could access bytes 10 to 100 from an object with a size of 200 bytes. This API feature is important for our algorithms, and is described more precisely in Section IV.

The cloud computing business model is to provide as many resources as needed when needed to the user. A compute cloud gives the illusion of an infinite resource with relatively high SLA guarantees as if the user owned his machine. As an overall "shared" resource infrastructure, it tries to be as efficient as possible, overlaying multiple users to a single resource in order to control the resource consumption of the user while allowing effectively infinite resource allocation. EC2 and S3 provide a simple cost model based on *pay-as-you-go*. A user pays for each EC2 instance used, the total storage space occupied in S3 as well as the network bandwidth used by the EC2 instances and S3 objects.

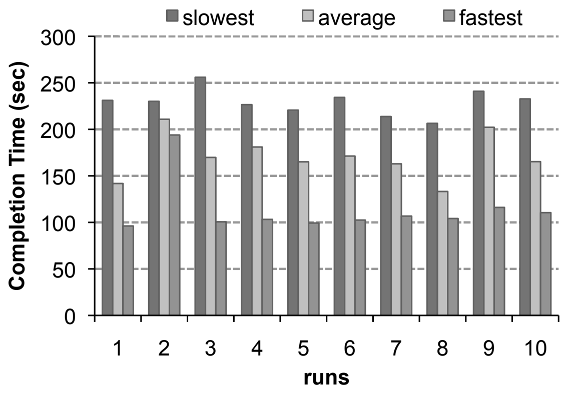

Fig. 2. download completion time with flat tree algorithm (8 nodes, 1GB)

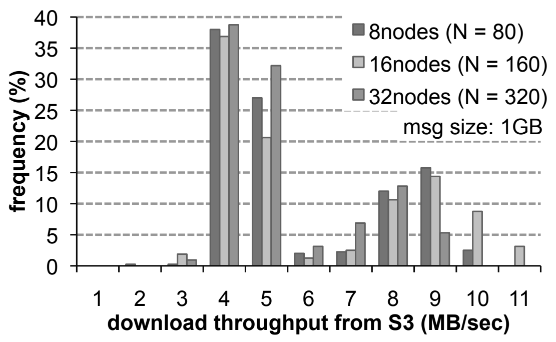

Fig. 3. download throughput frequency from S3 to 8, 16 and 32 EC2 nodes with flat tree

B. preliminary experiments

We will first evaluate the most simple algorithm to distribute data from S3 to multiple EC2 instances: have all the nodes simultaneously download all the data directly from S3 to their local storage. In the remainder of this paper we will call this algorithm, or equivalently the communication topology achieved with the algorithm, the *flat tree*.

We will first evaluate the download performance from S3 to a single EC2 node, and then to multiple EC2 nodes. In order to evaluate the multicast or download performance, we utilize download completion time and download throughput in the remainder of this paper; download throughput corresponds to the total incoming data size in bytes divided by the total download completion time which include protocol overhead, so it means application-level achievable throughput.

1) download performance to a single EC2 node: First, we evaluate the download performance for files of different sizes stored in S3 to a single EC2 node. Figure 1 shows the relative frequency ratio of download throughput to one EC2 node for different file sizes (10 MB, 100 MB and 1 GB) when a node repeatedly downloads the same file 1000 times. For example, for a 100 MB file, more than 60% of the all downloads from S3 achieves the throughput of approximately 6 MB/sec, while less than 30% achieves 8 MB/sec or more. We also observe that, with increase in the file size, we obtain two fairly diverse groups of high and low download performance. With 1GB files, for example, we see a peak at 4 MB/sec, but we also observe another peak at 8 MB/sec. This discrepancy turns out to be common across various other measurements, and is a crucial phenomenon to overcome, in order to achieve stable and fast multicast speeds.

Although we could not know clearly why these phennomenon occurred, we speculate that there exists network or I/O contention and bandwidth limitation caused by underlying cloud system.

2) download performance to multiple EC2 nodes: Next, we evaluate the download performance to multiple EC2 nodes using the flat tree. Figure 2 shows the maximum (slowest), average and minimum (fastest) download completion time when 8 nodes download the same 1 GB file simultaneously over 10 runs. Similar to the single node case, the download completion time (and accordingly the throughput) varies greatly between nodes as well as across successive experiment runs. Overall, the fastest node achieves a throughput of more than 10 MB/sec, while the slowest node only achieves about 4 MB/sec.

To observe the throughput difference amongst the nodes in more detail, Figure 3 shows the average download throughput frequency ratio with multiple nodes that all download the same 1GB file simultaneously. Here, we observe the same phenomenon as Figure 1: not only does the throughput change dynamically, but there are also two fairly diverse groups of high and low download performance.

Figure 4 shows the timewise trend of the aggregate throughput of 10 EC2 nodes as well as their breakdowns in achieved throughput over 10 runs downloading the same 1GB file. Here, we observe that the aforementioned download speed grouping does not necessarily correlate with the achieved throughput of the previous run. For example, the throughput of node 5 was about 10 MB/sec for the first run, but on the second run it was reduced to 5 MB/sec, and then recovered to 10 MB/sec for the third run, after that dropped to 5 MB/sec again for runs four and five. Moreover, the aggregate throughput of all the 8 nodes fluctuated around the average of about 50 MB/sec, although at times it was about 60 MB/sec while immediately after it dropped to under 40 MB/sec.

Figure 5 indicates the average total throughput of 10 runs for 8, 16, and 32 nodes, and then shows that overall the EC2/S3 system scales well, i.e. the aggregate bandwidth scales by and large linear to the number of EC2 nodes. With 32 nodes, it is still not saturated. This indicates that the fluctuation in bandwidth is not a scaling problem causing some straightforward congestion at the network traffic level as is the case in P2P systems, but more likely is due to some other resource control algorithm imposed within the cloud, e.g. Xen has some features of network and disk I/O QoS.

3) performance summary and problems: Overall, we observe that available download throughput from S3 to EC2 nodes dynamically changes according to the underlying state of the system where some undetermined and proactive band-

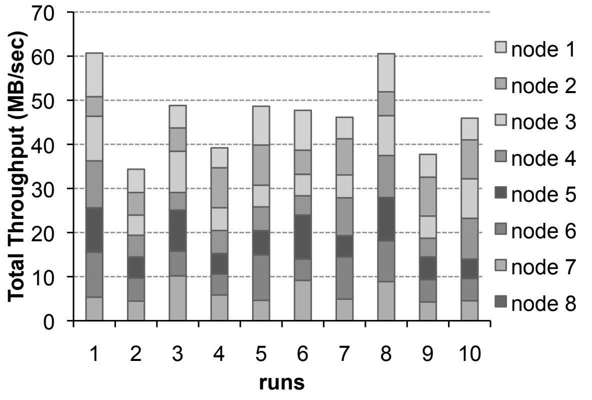

Fig. 4. sum of each node throughput from S3 (8nodes, 1GB, flat tree)

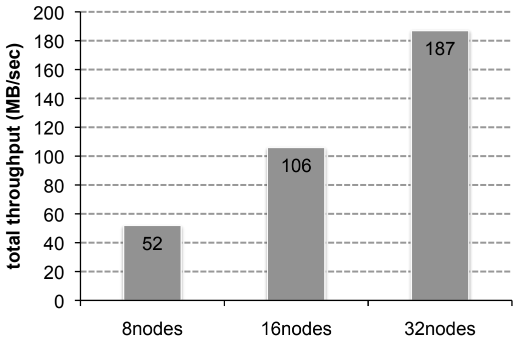

Fig. 5. average total throughput from S3 to EC2 node at 8nodes, 16nodes and 32nodes

width control is imposed by the resource management system of the cloud. As a result, we see very little stability or repeatability even if a node would download the same file successively. Some 'fortunate' nodes experience good speed for a while, but these are a significant but still relative minority (in our experiments, only 20-30% of all the nodes) and the situation can change rapidly. This indicates that we cannot make any (pseudo-)static assumptions about the optimal multicast topologies, as the performance differences are large enough to warrant some optimal multicast topologies but the underlying throughput assumptions are non-determinate, regardless of the number of nodes as Figure 3 indicates. On the other hand, we observe that the available throughput is by and large scalable with the number of nodes. This observation is promising for multicasts, as long as we can find effective ways of always utilizing most of the available bandwidth irrespective of the underlying instability.

C. the cloud usage model of data intensive applications

Our multicast algorithms are designed for effective hosting of data-intensive applications on clouds. We will assume that a typical data-intensive application would be started on multiple compute resources in a cloud in parallel, in order to process a large data set. The data set would be stored in the cloud's storage service for flexibility, cost and durability. Storing the data within the OS images themselves would be too inflexible, while storing the data at a user site and uploading it to each instance in the cloud would be too costly and slow, as well as reliability would be determined by the characteristics of the user's own storage. Hence, the ideal solution is to let each compute resource initially transfer the data set from the cloud storage service to its local storage before the application can start.

In this paper, we adopt Amazon EC2/S3 for the aforementioned usage scenario, although we believe our results would also apply to other clouds. Our objective is to distribute a large amount of data stored on S3 to all EC2 instances of an application, as fast as possible. To maintain the practicality of the usage scenario, we assume that EC2 instances and S3 objects are located in the same region, as with the current EC2/S3 infrastructure, transferring data from one region to another would be too costly and slow for processing large amounts of data.

IV. Our Proposed Multicast Algorithms for Clouds

We first define some general requirements that multicast operations on clouds should fulfill, and summarize the features of our proposed algorithms. Second, we describe our multicast algorithms in detail.

A. requirements

The general problems for multicast operations on clouds have been described in Section II. Section III has pointed out specific multicast performance problems with the flat tree algorithm on Amazon EC2/S3 (which could very well be general across different clouds). Consequently, we claim that a multicast operation on clouds has the following requirements:

- **maximized utilization of available aggregate download throughput from cloud storage**: the available throughput should scale according to the number of nodes, so the multicast algorithm achieves maximum utilization of the available aggregate download throughput despite any rapid changes in the underlying throughputs of individual nodes.
- **minimization of multicast completion time of each node**: usually all the nodes not only need to start to calculate as early as possible, but they also need to be able to commence the calculation by and large simultaneously to avoid the long-tail effect resulting in resource underutilization. A multicast operation should be stable, i.e. the data transfer finishes without any outliers even if the user's allocation scales to tens of thousands of nodes.
- **non-dependence on monitoring network throughput nor estimation of network topology**: various previous works we have mentioned usually tried to achieve the above policies by network monitoring and/or estimation of the physical network topologies. Such an approach is undesirable in clouds, as cloud compute resources are

TABLE I
FEATURES OF GENERAL AND PROPOSED MULTICAST ALGORITHMS ON EACH ENVIRONMENT

	cluster	P2P	clouds
multicast topology	spanning tree(s)	overlay	tree + overlay
communication type	push	pull	pull
network performance	high	low	middle
node proximity	dense	sparse	dense
node-to-node performance	homogeneous	heterogeneous	heterogeneous
storage-to-node performance	homogeneous	heterogeneous	heterogeneous
underlying network topology	stable	unstable	(un)stable
correspond to dynamic change	bad	good	good

dynamically provisioned by the underlying cloud system. Any measurement results would generate significant overhead due to extensive monitoring requirements while they would likely be not be applicable across runs.

Table I shows the different requirements and characteristics of multicast algorithms for clusters, P2P systems, and clouds. Multicast algorithms for clusters can achieve high performance by using an expensive algorithm that requires monitoring data to construct a high-throughput spanning tree, but they cannot adapt well to dynamic and unstable changes in network throughput. Multicast algorithms for P2P systems are scalable and can adopt well to network performance changes. However, it is difficult to use P2P algorithms to achieve high performance because they overly assume that not only the network but also the availability of the data and the nodes is unstable, resulting in significant overhead. Our proposed algorithms achieve both high performance and scalability by combining the advantages of multicast algorithms for clusters and P2P systems. In particular, we do not perform network monitoring, and balance the network load by using the workstealing technique.

B. the 'non-steal' algorithm

The multicast algorithm proposed by van de Geijn *et al.*[3] is a well known algorithm for clusters and multi-clusters environments. It achieves high performance multicast operations, and is often used in efficient MPI collectives implementations [5][6]. The algorithm consists of two phases: *(1) the scatter phase* and *(2) the allgather phase*. In the scatter phase, the root node divides the data to be multicast into blocks of equal size depending on the number of nodes. These blocks are then send to each corresponding node using a binomial tree. After all the nodes have receives the divided blocks, they start the allgather phase in which the missing blocks are exchanged and collected by using the recursive doubling technique.

Our proposed *non-steal* algorithm is inspired by this algorithm. It also consists of a scatter phase and an allgather phase. All nodes cooperate to download and forward data from S3 to each EC2 node. Initially, none of the nodes has any parts of the data stored in S3, so S3 corresponds to a multicast root node. Figure 6 depicts the two phases the proposed algorithm:

- **phase 1 (non-steal):** the file to distribute is logically divided into P fixed-sized pieces, e.g. 32KB each, numbered from 0 to $P-1$. When the number of nodes is N, each node i is assigned a range of pieces:
$$(\frac{iP}{N}, \frac{(i+1)P}{N} - 1) \qquad (1)$$
Node i then downloads all the pieces in its range from S3. Once a node finishes downloading the assigned range of pieces, it waits until all the other nodes have finished too.
- **phase 2:** after constructing a full overlay network between the all the nodes, each node contineously exchanges information with its neighbors in the mesh about which pieces they already obtained, and fetches missing pieces from them until all pieces are downloaded.

As an example of our non-steal algorithm, consider three nodes (A, B and C) that download the same 300 MB file from S3. Node B has a fast connection to S3 (10 MB/sec), while A and C have a slow connection to S3 (2 MB/sec). The file will first be logically split into 9600 pieces of 32KB each (9600 * 32KB = 300 MB). Initially, each node requests the assigned 100 MB from S3 (i.e. node A, B, and C request pieces 0-3199, 3200-6399 and 6400-9599, respectively). After approximately 10 seconds, node B will finish downloading its range. Node A and C, on the other hand, achieve slower throughput and will finish after 50 seconds. Since all nodes wait until everybody has finished, the total completion time of phase 1 is 50 seconds.

Once phase 1 is finished, all nodes start phase 2 and exchange the pieces using a BitTorrent-like protocol. Each node connects to some neighbor nodes and sends a possession list indicating which pieces it has. For example, node i sends its possession list to node j. If the list contains piece p that node j has not yet obtained, node j requests node i to send it piece p which is then returned by node i. After node j obtains piece p, node j informs all its neighbors that it now has piece p. The informed nodes then update their possession list of node j. All nodes communicate this way until they have obtained all pieces.

C. the 'steal' algorithm

In the non-steal algorithm, all nodes globally synchronize with each other once they have download their assigned range of pieces from S3. As shown in Figure 4, however, the completion time may vary greatly among the nodes due to significant and unpredictable variance in the download throughput. Our second algorithm resolves this issue by *stealing*: when a node

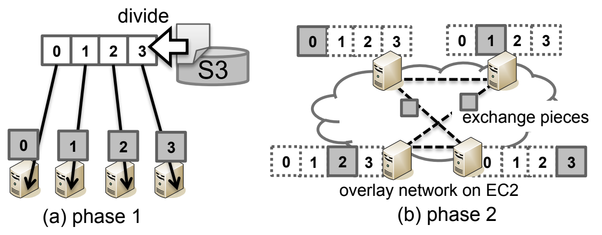

Fig. 6. abstract of proposed algorithm

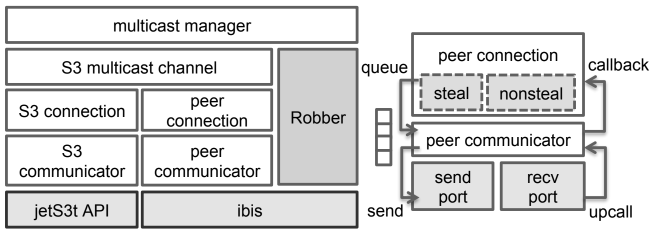

Fig. 7. overview of the software layers (left) and details of messaging implementation (right)

has downloaded its assigned range of pieces, it actively asks other nodes whether they still have pieces to download. If the node asked is still downloading, it splits its own assignments, and returns some of its pieces. This approach is similar to *work stealing*, in that a fast node 'steals' some download work from a slower node. To make the steal algorithm efficient, we adapt the *amount* of stolen work to the download bandwidth from S3 observed by both nodes.

The 'steal' algorithm consists of two phases:

- **phase 1 (steal)**: similar to the non-steal algorithm, the file to distribute is logically split into P equally-sized pieces, and each node i is assigned a range of pieces as shown in Equation 1. When node i has finished downloading its pieces, it asks other nodes whether they have any work remaining and reports its own download throughput B_i for the download just completed. Now assume that node j has W remaining pieces, and its download throughput is currently B_j. Node j then divides W into W_i and W_j such that:

$$W_i = \lceil W * B_i/(B_i + B_j) \rceil \tag{2}$$
$$W_j = \lfloor W * B_j/(B_i + B_j) \rfloor \tag{3}$$

  Node j then returns W_i pieces to node i. Node i and j can then concurrently download W_i and W_j pieces, respectively. Hence, the amount of work they download is proportional to their download bandwidth.
- **phase 2**: similar to the non-steal algorithm, each node exchanges pieces within EC2 by using a BitTorrent-like protocol until all the pieces have been obtained.

Note that in phase 1, we seem to use the heuristic that the download throughput is stable over a short period of time, yet the instability we have demonstrated in our earlier experiment would render such a heuristic ineffective. However, we claim that it is an effective strategy for the following reason: even if the new bandwidth B_i turns out to be slower than the previous B_i, it would be relatively OK: if W_i is small it will quickly terminate in any case, and if W_i is large it could be subject to further split by some alternative node that will have finished its work, i.e. it will be effectively in the position of node j.

Let us now apply the 'steal' algorithm in the same example scenario we described previously in Section IV-B. Initially Node A, B and C are assigned ranges 0-3199, 3200-6399, and 6400-9599. From the start of the algorithm, after approximately 10 seconds the fast node B will finish its work.

TABLE II
MESSAGE PROTOCOLS OF STEAL AND NON-STEAL ALGORITHM

Message	Information
give work	opcode, double(throughput value)
no work	opcode
work list	opcode, integer(start index), interger(end index)
done	opcode

Node B will then request more work (i.e. 'steal') from, for example, node A. By then, node A will have downloaded about 20 MB of its assigned total 100MB (10 seconds * 2 MB/sec = 20 MB). This is equivalent to 640 pieces, and leaves 2560 remaining pieces to download from S3. Node B will then steal work from node A according to equations (2) and (3), i.e. $\lceil 2560 * 10/(2 + 10) \rceil = 2134$ pieces (indices 1066-3199). These pieces are returned to node B as new *work*. Node A then restarts downloading $\lfloor 2560 * 2/(2 + 10) \rfloor = 426$ pieces (numbers 640-1065). This process is repeated until all pieces have been downloaded from S3. Finally, the nodes exchange all downloaded pieces with each other in phase 2 of the algorithm.

Our steal algorithm can generally apply not only to the Amazon EC2/S3, but also to the other cloud systems (e.g. Windows Azure Storage), moreover, it is possible to apply to the some parallel distributed file systems (e.g. Lustre and GFS [24]) in order to get higher throughput. This is because bandwidth flactuation per connection potentially will occur in these environments. For example, GFS divides file into a lot of fixed chunks and distributes these chunks to multiple chunk servers, so user can achieve high I/O throughput to/from chunk servers by aggregating each I/O connection. When some nodes access to the same chunk server simultaneously, however, I/O contention occuring access or I/O contention to same chunk server, however, it is confirmed that bandwidth performance in some links decreases [24].

D. implementation

We have implemented our algorithms on top of Ibis [25], a Java-based grid programming environment. The left hand side of Figure 7 gives an overview of the software layers in our implementation. Using the registry service provided by Ibis, each node is notified of the identity of new nodes that join the Ibis network, as well as its unique rank assigned by Ibis.

The steal and non-steal algorithms are implemented as an extension to the S3MulticastChannel object. An

S3MulticastChannel manages an S3Connection object and a number of PeerConnection objects. The S3Connection object downloads pieces from S3, while the PeerConnection objects send and receive messages to and from other nodes.

In phase 1 of the algorithms, each node starts to download its assigned pieces from S3 using the S3Communicator object. Files on S3 are located using the jetS3t API[26] (a Java based implementation to access Amazon's cloud service) and then downloaded from S3 by using the REST API.

After downloading all the assigned pieces, a node sends a 'give work' message to other nodes through the corresponding PeerConnection object for the steal algorithm, whereas it will send a 'done' message for the non-steal algorithm. The right hand side of Figure 7 shows the details of the implementations of the steal and non-steal protocols. Table II lists the messages used in the protocol. The PeerCommunicator object uses *send port* and *receive port* primitives provided by Ibis to send and receive each message in the protocol. All the messages sent are first put into a queue, after which each message is obtained from a queue one by one by a dedicated thread to avoid deadlocks. Incoming messages are received asynchronously using *upcalls* provided by Ibis, and are handled by callback functions in the PeerConnection object.

After all nodes have downloaded all pieces from S3, the algorithms switch into phase 2 an use BitTorrent-like communication among the nodes as described earlier. The actual implementation uses the Robber [22] system implemented on top of Ibis. Robber achieves high-throughput multicast in bandwidth-constrained environments such as multi-clusters connected by a WAN. Robber can automatically adapt its performance to the underlying network environment without any network monitoring, similar to our steal algorithm. The indices of all received pieces are passed to Robber, which then delivers all pieces to all nodes.

V. PERFORMANCE EVALUATION

To evaluate the performance of our proposed algorithms, we compare the multicast performance of the flat tree, non-steal and steal algorithms on Amazon EC2/S3. We measure the completion time, stability and scalability of each algorithm, and also investigate the internal performance breakdown of the phases in our algorithms. We only use EC2/S3 sites located in Europe, with a *small* EC2 instance type. Best and worst achievable raw network performance between two EC2 node with *small* instance are 808Mbps and 363Mbps, which is measured over 2000 runs by Iperf. Moreover, we just set 32KB for piece size in this experiment, but it is not certain whether 32KB is the most suitable. Conducting benchmarks with more sites, instance types, and different piece size are considered future work, and may actually amplify our performance advantage (precise node performance of each instance type is described in [23]).

A. *completion time and stability*

In our first experiment, we distribute a 1 GB file stored in S3 to 8 EC2 nodes using the non-steal and steal algorithms. We repeat this 10 times to analyze the stability of both algorithms. Figure 8 and 9 show the completion times of the non-steal and steal algorithms, respectively. For each run, they show the slowest (worst), fastest (best) and average overall completion time over all nodes.

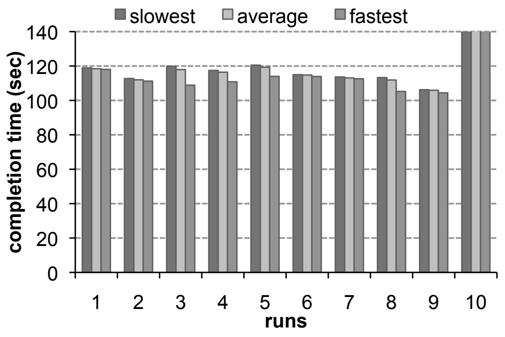

Fig. 8. completion time with non-steal algorithm (1GB, 8 nodes)

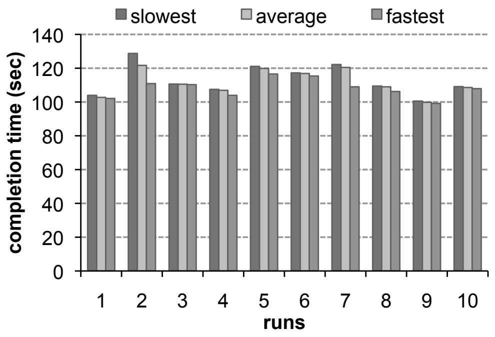

Fig. 9. completion time with steal algorithm (1GB, 8 nodes)

Compared to the flat tree performance in Figure 2, the best completion time is similar across all algorithms: about 100 seconds for this experiment. However, when we compare the slowest completion time, at least one node in the flat tree takes approximately 200 seconds, but none of the nodes in either the non-steal or steal algorithms takes more than 140 seconds. Overall, the best and worst completion time of the flat tree algorithm differ drastically, whereas it is fairly stable for both the non-steal and steal algorithms. In fact, the throughput of all the nodes with the non-steal and steal algorithms was about 1.7 times higher compared to that of the flat tree.

Comparing the non-steal and steal algorithm, we further observe that the latter is noticeably more stable and achieves higher performance. For example, in our experiment the throughput achieved by the non-steal algorithm suddenly drops in the tenth run, whereas we see no such perturbation with the steal algorithm. The reason for this sudden performance loss is that one of the nodes suffered a random slowdown in throughput, which prevented all the nodes to proceed to phase 2. The steal algorithm would adapt automatically to such a case by balancing the load, and achieve stable transfer

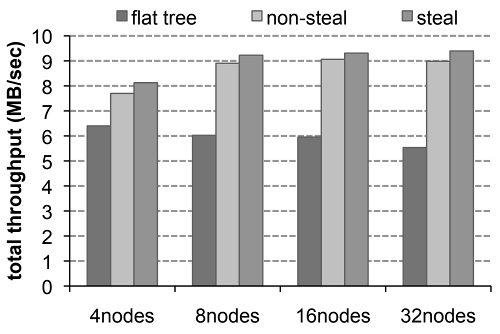

Fig. 10. total average throughput for the number of nodes (1GB)

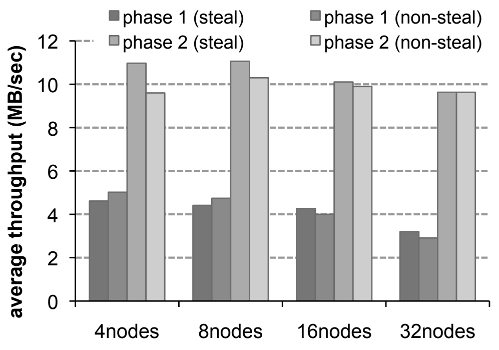

Fig. 11. average throughput details of phase 1 and phase2 with steal and non-steal algorithm

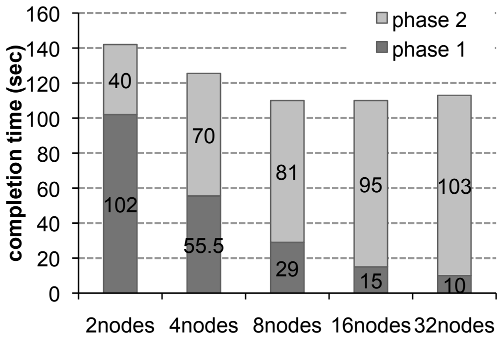

Fig. 12. completion time ratio of phase 1 and phase 2 (steal, 1GB)

throughput.

B. node scalability

Next, we evaluate the scalability of each algorithm with respect to the number of nodes. In a second experiment, we distribute a 1 GB file to 4, 8, 16, and 32 nodes using all three algorithms. Figure 10 shows the overall throughput of each case.

The first observation is that with the flat tree algorithm, throughput gradually decreases as we increase the number of nodes. In contrast, our proposed algorithms allow all the nodes to obtain higher bandwidth and achieve about 9 MB/sec.

As shown earlier Figure 3, only a limited set of EC2 nodes can achieve about 10 MB/sec throughput. The majority of nodes can only achieve about 5 MB/sec throughput. With more nodes, the performance of the flat-tree algorithm will therefore be more and more likely degrade to about 5 MB/s, whereas our algorithms maintaining stable performance regardless of the number of nodes. With 32 nodes, the non-steal and steal algorithm achieve 1.6 and 1.7 times more throughput than the flat tree, respectively. These results shows that our algorithms adapt well to changes in the underlying bandwidth, and achieve higher performance, both in transfers from S3 as well as within EC2.

C. Finer analysis of our proposed algorithms

Finally, we investigate the finer performance details of our proposed algorithms. Figure 11 shows the average throughput in each phase of the steal and non-steal algorithms. For both algorithms, the achieved throughput of phase 1 is approximately 5 MB/sec or less, irrespective of the number of nodes. The throughput of phase 2 is approximately 10 MB/sec or less. The throughput of phase 1 decreases when the number of nodes increases, which is caused by the fact that with more nodes each node downloads relatively less data from S3. Requesting more data at once increases the individual througput, but can also decrease the overall throughput when the individual throughput turns out to be low. The performance in phase 2 is high and stable because of the BitTorrent-like exchange algorithm.

Figure 12 shows how much time the steal algorithm spends in phase 1 and phase 2 for different numbers of nodes. It can be seen that, as the number of nodes increases, the time spent in phase 1 decrease while the time spent in phase 2 increases, but the total time spent in phase 1 and phase 2 remains nearly constant. Compared the completion time of 16 nodes with 32 nodes, 32 nodes take slightly much time than 16 nodes. That is the reasons steal negotiation overhead in our algorithm is larger with many nodes, and also it can not gain high download throughput in phase 1 because first allocated file block size is too small against the number of nodes. With many nodes, the overall completion time is dominated by phase 2 and becomes constant irrespective of the number of nodes. This indicates that our steal algorithm is effectively scalable, although the difference with the non-steal algorithm becomes smaller.

VI. Conclusion

In this paper, we have addressed cloud characteristics and summarized different multicast algorithms for distributed parallel and P2P systems. We have focused on Amazon EC2/S3 which is the most commonly used cloud platform, and revealed some multicast performance problems on there when using the most simple algorithm by which all EC2 compute nodes download files from S3 directly. There are two main types of problems with optimizing multicast performance on clouds:

(1) the network performance within clouds is dynamic. It means the network performance changes not only between compute nodes but also especially between the cloud storage and each compute node. (2) keeping network monitoring data and topology information up-to-date is almost impossible. It means it is difficult to know the underlying cloud's physical infrastructures and construct one or more available bandwidth maximized spanning trees, which is well known-known optimization technique for cluster systems.

Based on these findings, we have presented two high performance multicast algorithms. These algorithms make it possible to transfer large amounts of data stored in S3 to multiple EC2 nodes efficiently. The proposed algorithms combine optimization ideas from multicast algorithms used in parallel distributed systems and P2P systems, and they have three salient features; (1) they can construct an overlay network on clouds without network topology information, (2) they can optimize the total throughput between compute nodes dynamically, and (3) they can increase the download throughput from cloud storage by letting nodes cooperate with each other, resembling work stealing techniques.

We have implemented the two proposed algorithms and evaluated their performance on Amazon EC2/S3 by comparing with the simple, flat tree algorithm. As a result, it was confirmed that these algorithms constantly achieve higher available throughput, about 1.7 times of the flat tree algorithm, regardless of the number of EC2 nodes. Moreover, the steal algorithm achieved higher throughput than the non-steal algorithm since the steal algorithm could adapt to the dynamic changes of the download throughput from S3.

As future work we plan to improve our proposed algorithms in the following two ways: (1) merge the two phases into one phase so as to overlap the S3 to EC2 and intra-EC2 transfers to allow early dissemination of already fetched data, and (2) use multiple connections from each EC2 node to S3 for satisfying each node's total available download throughput. The resulting multicast performance should be improved even further. We also plan to conduct further performance comparisons with other multicast algorithms such as using pipelined spanning trees. Lastly, we plan to test our algorithm on larger and/or more divergent sets of cloud platforms to confirm the scalability of the result.

Acknowledgments

This work is partly supported by Grant-in-Aid for JSPS Fellows (20·8911) and Grant-in-Aid for Scientific Research (18049028) from Ministry of Education, Culture, Sports and Technology, Japan. The authors gratefully acknowledge a grant given by Amazon's *AWS in Education* program that gave access to the EC2/S3 platform.

References

[1] Amazon Web Services, "http://aws.amazon.com/."

[2] Windows Azure, "http://www.microsoft.com/windowsazure/."

[3] M. Barnett, L. Shuler, S. Gupta, D. G. Payne, R. A. van de Geijn, and J. Watts, "Building a high-performance collective communication library," in *Supercomputing*, 1994, pp. 107–116. [Online]. Available: http://citeseer.ist.psu.edu/140591.html

[4] A Toroidal LHC ApparatuS Project (ATLAS), "http://atlas.web.cern.ch/."

[5] M. Matsuda, T. Kudoh, Y. Kodama, R. Takano, and Y. Ishikawa, "Efficient mpi collective operations for clusters in long-and-fast networks," in *IEEE International Conference on Cluster Computing (cluster 2006)*, 2006.

[6] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," in *International Journal of High Performance Computer Applications*, vol. 19, no. 1, 2005, pp. 49–66.

[7] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang, "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999, pp. 131–140.

[8] R. Izmailov, S. Ganguly, and N. Tu, "Fast parallel file replication in data grid," in *Future of Grid Data Environments workshop (GGF-10)*, 2004.

[9] M. den Burger, T. Kielmann, and H. E. Bal, "Balanced multicasting: High-throughput communication for grid applications," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[10] T. Chiba, T. Endo, and S. Matsuoka, "High-performance mpi broadcast algorithm for grid environments utilizing multi-lane NICs," in *7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007, pp. 487–494.

[11] K. Takahashi, H. Saito, T. Shibata, and K. Taura, "A stable broadcast algorithm," in *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2008, pp. 392–400.

[12] T. Shirai, H. Saito, and K. Taura, "A fast topology inference - a building block for network-aware parallel processing," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. ACM, 2007, pp. 11–22.

[13] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[14] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: High-bandwidth multicast in cooperative environments," in *In 19th ACM Symposium on Operating Systems Principles*, 2003. [Online]. Available: citeseer.ist.psu.edu/article/castro03splitstream.html

[15] M. den Burger and T. Kielmann, "Mob: Zero-configuration high-throughput multicasting for grid applications," in *16th IEEE International Symposium on High Performance Distributed Computing (HPDC '07)*, 2007, pp. 159–168.

[16] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The Cost of Doing Science on the Cloud: The Montage Example," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1–12.

[17] Montage Project, "http://montage.ipac.caltech.edu."

[18] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?" in *UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. New York, NY, USA: ACM, 2009, pp. 17–20.

[19] E. Walker, "benchmarking amazon ec2 for high-performance scientific computing," *LOGIN*, pp. 18–23, October 2008.

[20] S. L. Garfinkel, "An evaluation of amazon's grid computing services: Ec2, s3 and sqs," Center for Research on Computation and Society School for Engineering and Applied Sciences, Harvard University, Tech. Rep., 2007.

[21] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for science grids: a viable solution?" in *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*. New York, NY, USA: ACM, 2008, pp. 55–64.

[22] M. den Burger and T. Kielmann, "Collective receiver-initated multicast for grid applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2010.

[23] Amazon EC2 Instance Types, "http://aws.amazon.com/ec2/instance-types/."

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *19th ACM Symposium on Operating Systems Principles (SOSP-19)*, 2003.

[25] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a flexible and efficient Java based grid programming environment," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 7-8, pp. 1079–1107, June 2005.

[26] jetS3t, "http://jets3t.s3.amazonaws.com/."