Collective Receiver-initiated Multicast for Grid Applications

Mathijs den Burger and Thilo Kielmann

Abstract—Grid applications often need to distribute large amounts of data efficiently from one cluster to multiple others (multicast). Existing sender-initiated methods arrange nodes in optimized tree structures, based on external network monitoring data. This dependence on monitoring data severely impacts both ease of deployment and adaptivity to dynamically changing network conditions. In this paper we present *Robber*, a collective, receiver-initiated, high-throughput multicast approach inspired by the BitTorrent protocol. Unlike BitTorrent, Robber is specifically designed to maximize the throughput between multiple cluster computers. Nodes in the same cluster work together as a *collective* that tries to *steal* data from peer clusters. Instead of using potentially outdated monitoring data, Robber automatically adapts to the currently achievable bandwidth ratios. Within a collective, nodes automatically tune the amount of data they steal remotely to their relative performance.

Our experimental evaluation compares Robber to BitTorrent, to Balanced Multicasting, and to its predecessor MOB. Balanced Multicasting optimizes multicast trees based on external monitoring data, while MOB uses collective, receiver-initiated multicast with static load balancing. We show that both Robber and MOB outperform BitTorrent. They are competitive with Balanced Multicasting as long as the network bandwidth remains stable, and outperform it by wide margins when bandwidth changes dynamically. In large environments and heterogeneous clusters, Robber outperforms MOB.

Index Terms—High-throughput multicast, Load-balancing, Cluster computing

❖

1 INTRODUCTION

G RID computing serves high-performance applications by integrating multiple sites, ranging from single machines to large cluster computers, located around the world. Contrary to more traditional computing environments like clusters or super computers, the network characteristics between grid sites are both very heterogeneous and dynamically changing. Communication libraries need to take this heterogeneity into account to stay efficient in a world-wide environment.

A typical communication pattern is *multicast*: the transfer of a substantial amount of data from one site to multiple others. A common use case is the distribution of large input data of a parallel application before or during a run. For example, BLAST is a widely used application to perform queries on DNA and protein databases. A significant fraction of the runtime can be spent in distributing the database to compute nodes [1]. Another example is multimedia content analysis, processing huge amounts of image and video data [2]. Using Grids to store and analyze these data becomes increasingly popular, and needs efficient multicasting.

Traditionally, multicast has been implemented using a *sender-initiated* approach: the application nodes are arranged in one or more spanning trees over which the data are sent. The advantage of this approach is that once the trees have been set up, routing the data is easy. The hard part is deciding which trees to use. In static environments, like the network within a supercomputer, fixed tree shapes like binary or binomial trees can be used. In grid environments, however, this method can be very inefficient, as bandwidth between sites can vary significantly among network paths and also over time.

The completion time of large data transfers depends primarily on the bandwidth an application can achieve across the interconnection network. Commonly applied methods today [3], [4], [5] rely on monitoring information to construct optimal multicast trees. The disadvantage of these approaches is that (1) It assumes network monitoring systems to be deployed ubiquitously. (2) It assumes monitored data to be both accurate and stable during a multicast operation, which might not be the case in shared networks with variable background traffic. (3) Network monitoring systems monitor the network itself; it remains a hard problem to translate this data (e.g., *available bandwidth*) to information that is meaningful to an application or multicasting algorithm (e.g., *achievable bandwidth* [6]). Our previous work on *Balanced Multicasting* [7] belongs to this category.

More recently, *receiver-initiated* multicast has become popular in peer-to-peer networking [8], [9]. Here, the application nodes are arranged in a random mesh, and explicitly request data from their neighbors. Nodes update each other about which parts of the data they possess, and randomly exchange parts with each other. This way, nodes dynamically route the data over the mesh. The request-reply interaction between nodes automatically adapts the effective throughput to the available bandwidth, which handles heterogeneous and fluctuating WAN bandwidth very well.

Most current receiver-initiated multicast approaches are designed for peer-to-peer systems of individual and uncooperative nodes. In contrast, we apply this ap-

• *M. den Burger and T. Kielmann are with Vrije Universiteit Amsterdam.*
E-mail: mathijs@cs.vu.nl, kielmann@cs.vu.nl

proach to grid environments, consisting of multiple clusters of cooperative application nodes. In such systems, the set of nodes used by a grid application remains constant during a single run, i.e. there is no churn. (Different runs may use different sets of nodes.) Also, communication can be considered reliable, subject to the underlying transport protocol (e.g., TCP/IP). Any data loss will only affect the achievable bandwidth, which in turn is handled by our multicast algorithms.

Previously, we developed *MOB* [10], a *collective* receiver-initiated multicast approach specifically designed for clusters. MOB is inspired by the BitTorrent protocol [8], and lets nodes in the same cluster team up in a cooperative collective that requests data from other clusters as efficiently as possible. Each node in a collective is responsible for finding an equal part of all data remotely, which is distributed locally to other members of the same collective. MOB works well with small numbers of homogeneous clusters, but becomes inefficient in large grid environments or with heterogeneous clusters. Here, some nodes will be slower than others in terms of wide-area throughput they can achieve: they can have either slower network cards, or connections to other, slower clusters. Such slow nodes can degrade the overall throughput significantly.

This paper presents *Robber*, a successor to MOB that adds dynamic load balancing within a collective. Instead of using a static division of *work* (the data to request remotely, nodes that have become idle steal work from other nodes in the same collective. As a consequence, each node automatically performs an amount of work proportional to its relative speed in a collective. This avoids waiting for slow nodes to complete their share of work, greatly enhancing the overall throughput.

We have implemented Robber (as well as Balanced Multicasting, MOB, and the BitTorrent protocol) within our Java-based Ibis system [11]. We have experimentally evaluated the four approaches by emulating various wide-area networks in the DAS-3 multi-cluster system [12]. We show that both Robber and MOB outperform the original BitTorrent protocol by substantially reducing the load on wide-area links between clusters. In the case of stable wide-area bandwidth, they automatically achieve multicast bandwidth that is equivalent to Balanced Multicasting, yet without using any external monitoring data. They also adapt their behavior automatically when bandwidth drops or grows during a multicast operation, while Balanced Multicasting then either congests certain links or fails to exploit additional bandwidth. In large grid environments and with heterogeneous clusters, Robber automatically adjusts the work load of nodes in each cluster to their relative performance, resulting in much better throughput.

The outline of this paper is as follows. Section 2 discusses background and related work. Section 3 describes the Robber algorithm, proves its correctness and outlines its implementation. Section 4 evaluates the various approaches experimentally, and Section 5 concludes.

2 BACKGROUND AND RELATED WORK

In a *multicast* operation, the *root* node is transmitting data to all other nodes of a given group, like the processes of an application. This is comparable to MPI's broadcast operation. For optimizing multicast, we are minimizing the overall completion time, from the moment the root node starts transmitting until the last receiver has got all data. As we are interested in multicasting large data sets, we optimize for high throughput. Section 4 will thus report our results as achieved throughput (in MB/s).

Before presenting our new collective multicast algorithm, Robber, we first discuss more traditional approaches to multicasting in grids and Internet-based environments. In this section, we also summarize our previous approaches Balanced Multicasting and MOB, as well as some other receiver-initiated multicast approaches, and discuss their performance limitations. We complete our discussion with some background on random stealing, which is used in our Robber algorithm.

2.1 Overlay Multicasting

Multicasting over the Internet started with the development of IP multicast, which uses specialized routers to forward packets. Since IP multicast was never widely deployed, *overlay multicasting* became popular, in which only the end hosts play an active role. Several centralized or distributed algorithms have been proposed to find a single overlay multicast tree with maximum throughput [13], [14]. Splitting the data over multiple trees can increase the throughput even further.

A related topic is the overlay multicast of media streams, in which it is possible for hosts to only receive part of the data (which results in, for instance, lower video quality). In [14], [15], a single multicast tree is used for this purpose. SplitStream [16] uses multiple trees to do distribute streaming media in a P2P context. Depending on the bandwidth each host is willing to donate, the hosts receive a certain amount of the total data stream. The maximum throughput is thus limited to the bandwidth the stream requires. In contrast, our multicast approaches try to use the maximum amount of bandwidth the hosts and networks can deliver.

2.2 Network Performance Modeling

Throughout this work, we assume networks as sketched in Fig. 2.2. Nodes are distributed among clusters. Within each cluster, nodes are connected via some local interconnect. Towards the WAN, each node has a network interface that is connected to a shared access link. All access links end at a gateway router (typically to the Internet). Within the WAN, we assume full connectivity among all clusters.

For optimizing multicast operations, we need to efficiently use the available network bandwidth where we distinguish, as outlined in [6]. *Bandwidth Capacity* is the maximum amount of data per time unit that a hop or
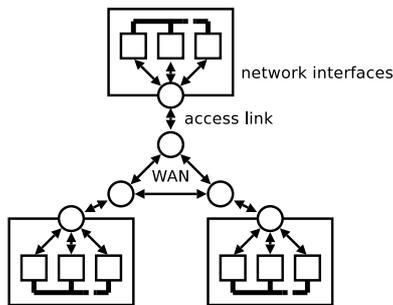
Fig. 1: Network model including clusters

path can carry. *Achievable Bandwidth* is the maximum amount that a hop or path can provide to an application, given the current utilization, the protocol and operating system used, and the end-host performance.

We are interested in maximizing the achievable bandwidth of all data streams used for a multicast operation. In multicasting, sharing effects can be observed whenever a single host is sending to and/or receiving from multiple other hosts. Here, the bandwidth capacity of the local network can become a bottleneck. This *local capacity* can be limited either by the network interface (e.g., a FastEthernet card, connected to a gigabit network), or by the access link to the Internet that is shared by all machines of a site. In Section 4 we refer to this setting as a *local bottleneck* environment, dominated by local bandwidth capacity. The opposite situation, where the bottleneck bandwidth is dominated by the achievable bandwidth across the wide-area network, we will call a *global bottleneck* environment.

In order to optimize multicast operations based on the given network characteristics, one has to rely on external network monitoring systems like the Network Weather Service [17], REMOS [18], or Delphoi [19]. Using such tools, however, has its own issues. First of all, the monitoring tools have to be deployed between all clusters in question. Frequently, this is an administrative issue. Second, network bandwidth is measured using active probes (sending measurement traffic) which can take significant amounts of time and scales only poorly to large environments as, for N clusters, $O(N^2)$ network paths need to be measured. Consequently, measurements are run in frequencies that are too low to properly follow dynamic bandwidth fluctuations. Finally, network monitoring tools measure the properties of the network paths themselves rather than the properties that are relevant to the applications, namely *achievable bandwidth*. Translating monitoring data to application-level terms is a hard problem [19].

2.3 Optimizing Sender-initiated Multicast

Optimization of multicast communication has been studied extensively within the context of message passing systems and their collective operations. The most basic approach to multicasting is to ignore network information altogether and send directly from the root host to

all others. MagPIe [5] used this approach by splitting a multicast in two layers: one within a cluster, and one flat tree between clusters. Such a flat tree multicast puts a high load on the outgoing local capacity of the root node, which often becomes the overall bandwidth bottleneck.

As an improvement, we can let certain hosts forward received data to other hosts. This allows to arrange all hosts in a directed spanning tree over which the data are sent. MPICH-G2 [20] followed this idea by building a multi-layer multicast to distinguish wide-area, LAN and local communication. As a further improvement for large data sets, the data should be split to small messages that are forwarded by the intermediate hosts as soon as they are received to create a high-throughput pipeline from the root to each leaf in the tree [21].

The problem with this approach is to find the optimal spanning tree. If the bandwidth between all hosts is homogeneous, we can use a fixed tree shape like a chain or binomial tree, which is often used within clusters [22]. As a first optimization for heterogeneous networks, we can take the achievable bandwidth between all hosts into account. The throughput of a multicast tree is then determined by its link with the least achievable bandwidth. Maximizing this *bottleneck bandwidth* can be done with a variant of Prim's algorithm, which yields the *maximum bottleneck tree* [13].

However, this maximum bottleneck tree is not necessarily optimal because each host also has a certain local capacity. A forwarding host should send data to all its n children at a rate at least equal to the overall multicast throughput t. If its outgoing local capacity is less than $n \cdot t$, it cannot fulfill this condition and the actual multicast throughput will be less than expected. Unfortunately, taking this into account generates an NP-hard problem.

The problem of maximizing the throughput of a set of overlay multicast trees has also been explored theoretically. Finding the optimal solution can be expressed as a linear programming problem, but the number of constraints grows exponentially with the number of hosts. In theory, this can be reduced to a square number of constraints, but in practice finding the exact solution can be slow and expensive [23]. Any solution thus will have to rely on heuristics to be applicable in real time.

The multiple tree approach in [3] uses linear programming to determine the maximum multicast throughput given the bandwidth of links between hosts, but requires a very complicated algorithm to derive the set of multicast trees that would achieve that throughput. Therefore, the linear programming solution is only used to optimize the throughput of a single multicast tree.

The Fast Parallel File Replication (FPFR) tool [4] is implementing multiple, concurrently used multicast trees. FPFR repeatedly uses depth-first search to find a tree spanning all hosts. For each tree, its bottleneck bandwidth is "reserved" on all links used in the tree. Links with no bandwidth left can no longer be used for new trees. This search for trees continues until no more trees

spanning all hosts can be found. The file is then multicast in fixed-size chunks using all trees found. FPFR does not take the local bandwidth capacity of hosts into account, leading to over subscription of links, forming capacity bottlenecks. In consequence, depending on local capacities, FPFR may perform much worse than expected.

2.4 Balanced Multicasting

In previous work [7], we have presented Balanced Multicasting, improving over FPFR by also taking bandwidth capacity into account. An example is shown in Fig. 2(a), consisting of three hosts, each connected to the network by their access line. Routers connect access lines with the WAN. Access lines are annotated with their local capacity, e.g. the capacity of the LAN. Wide-area connections are annotated with their achievable bandwidth. For simplicity of the example, we assume all connections to be symmetrical in both directions. (The actual units for bandwidth are not relevant here.)

(a) Network example: x sends data to y and z

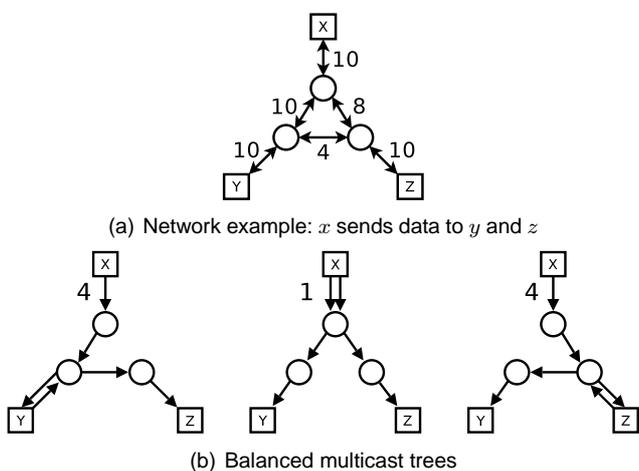

(b) Balanced multicast trees

Fig. 2: Example of Balanced Multicasting

In this example, Balanced Multicasting creates the three multicast trees shown in Fig. 2(b), with a total achievable bandwidth of 9. Together, these trees maximize the multicast throughput while not oversubscribing individual link capacities. Please note that the individual trees may have different bandwidth, in the example 4, 1, and 4. These different data rates are enforced by traffic shaping at the sender side. This process of balancing the bandwidth shares gave the name to the approach. If the sender would not balance the shares of the three trees, then the middle tree (using the LAN at x twice) would consume bandwidth that was intended for the other trees, resulting in a total bandwidth of $3 \times 10/4 = 7.5$, instead of the anticipated 9.

This example shows balanced multicast trees as they are computed by our algorithm published in [7]. Finding the optimal set of balanced multicast trees is an NP-hard problem. For this reason, our implementation is using heuristics to find solutions which we have shown in [7] to be close to the optimum.

When evaluating Robber, we compare to Balanced Multicasting as a (close-to) optimal solution that can be found with complete network performance information. Balanced Multicasting, however, like all other spanning-tree based multicasting strategies, is computing its optimized spanning trees based on the monitoring data available at the time when the multicast is started. Later changes in the network will *not* be taken into account.

2.5 Receiver-initiated Multicast

As explained so far, deriving optimized multicast trees is a hard problem. Especially in the case of dynamically changing network performance, carefully computed multicast trees can easily become inefficient. Therefore, several alternatives have been developed based on receiver-initiated communication, in which nodes explicitly request data from each other instead of forwarding it over trees.

Bullet [24] takes a hybrid approach to high-throughput multicasting. A Bullet network consists of a tree combined with a mesh overlay. The data is divided in blocks that are further divided in packets. Nodes send a disjoint subset of packets to their children in the tree, and request the remaining pieces from a set of disjoint peers in the system. The selection of those peers is based on random, orthogonal subsets of nodes distributed periodically by the RanSub algorithm. Bullet's additional mesh distribution layer yields significantly better throughput than traditional tree structures.

BitTorrent [8] is a peer-to-peer application, designed to distribute large files efficiently. The data is logically split in P equally-sized pieces, usually a few hundred kilobytes each. Nodes create an overlay mesh by connecting to a few peer nodes chosen at random, and tell each other which pieces they already possess. From then on, nodes constantly inform each other which new pieces they received. Nodes explicitly request pieces from their peers, which are randomly chosen from the reported ones. Each node always has R outstanding requests (we use $R = 5$) to get the 'pipelining' effect described in [8]. Which peers are allowed to request pieces is decided by the so-called *choking algorithm*. A node 'unchokes' only N peers at the same time (we use $N = 5$), thereby allowing them to download pieces. The decision to choke or unchoke peers is made every 10 seconds, and is based on the observed download rate. This results in an incentive to upload pieces, since uploading gives a higher chance of being allowed to download.

Chainsaw [9] uses a simplified version of the BitTorrent protocol, applied to live streaming of data. Nodes have a sliding window of interest, which they advertise to their neighbors. Packets that could not be found in time 'fall off' the trailing edge of the window, and are considered lost.

2.6 Clustering Nodes

Other work has already recognized that grouping receiver-initiated multicast nodes to clusters can increase

the overall throughput. *Biased neighbor selection* [25] proposes to group BitTorrent nodes by ISP (Internet Service Provider), which reduces the amount of costly traffic between ISPs. Robber is doing a similar grouping by cluster, but also adds teamwork among the nodes of a cluster to further improve multicast performance. In uncooperative peer-to-peer environments, this improvement would not be possible.

Another approach is followed by Tribler [26], a BitTorrent client that groups users in social clusters of friends. The amount of trust between nodes is increased using existing relations between people. Users can tag each other as a friend, indicating they are willing to donate upload bandwidth to each other by searching each other's pieces. Robber is essentially an automation of this technique applied to grid clusters, with all nodes in the same cluster being friends. However, the coordination of teamwork in Robber is much more efficient.

Robber's predecessor *MOB* [10] is based on the BitTorrent protocol. Nodes in the same cluster are grouped to '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. Although MOB achieves good throughput with a small amount of homogeneous clusters, its static load balancing approach fails in larger or more heterogeneous grid environments.

The amount of WAN traffic between clusters of BitTorrent nodes can also be decreased using network coding [27]. Nodes then exchange linear combinations of pieces, which increases the probability that a peer in the same cluster has data of interest. However, the complexity and computational overhead of network coding have limited its practical use. The work division in MOB and Robber is much more lightweight and also minimizes the WAN traffic between clusters.

2.7 Random Work Stealing

Random work stealing is a well known load balancing technique used in various distributed computing systems. It can be applied when multiple nodes are solving a computational problem by dividing it into a number of smaller problems. All problems assigned to a node are called its *work*. Each node starts solving all problems assigned to it. When a node becomes idle, it will attempt to *steal* some work from a randomly selected peer, repeating steal attempts until it succeeds. This way, faster nodes will eventually process more work than slower nodes. Robber uses this technique to dynamically spread the bandwidth demand of a multicast operation over nodes in the same cluster.

3 ROBBER

In this section we present *Robber*, a multicast algorithm based on collective data stealing. Robber distributes data over a random mesh by letting nodes 'steal' pieces from other nodes. In addition, nodes in the same cluster team up in collectives that together try to steal pieces from nodes in other clusters as efficiently as possible. The total set of pieces a collective C has to steal from nodes in other clusters is called its $work(C)$. Initially, each node $n \in C$ is assigned an equal share of work, denoted as $work(n)$. Each stolen piece is exchanged locally between members of a collective, such that, at the end, all nodes will have received all data. When a node has no more work left, it attempts to *steal work* from a randomly selected local peer. This way, faster nodes download more pieces to a cluster than slower nodes, which prevents the latter from becoming the overall bandwidth bottleneck.

Which nodes are located in which clusters is assumed to be globally known, and often provided by the runtime system (in our case, Ibis). For each node, we will call nodes located in the same cluster *local* nodes, and nodes in other clusters *global* nodes. The number of nodes in a collective C is denoted as $|C|$. Each node $n \in C$ has a 'collective rank' $r(n, C)$, ranging from 0 to $|C| - 1$.

We will first describe the algorithm used by Robber nodes to connect to and communicate with each other. Second, we will prove the correctness of the Robber algorithm. Finally, we will outline our implementation.

3.1 Algorithm

The Robber multicast algorithm consists of four phases.

Phase 1: Each node $n \in C$ chooses $max(|C| - 1, N)$ (i.e. up to N) *local peers* uniformly at random from all other nodes in the same collective, and initiates a connection to them. Throughout this paper we use $N = 5$, which is also used in BitTorrent as a reasonable amount of peers to saturate a node's local capacity. Section 3.3 explains why $N \geq 3$ to guarantee Robber's correctness.

Each connection provides bidirectional communication. Incoming connections from other local nodes are always accepted, and these nodes will be added to n's local peers, too. If two nodes choose each other as local peers, only one connection is set up.

Phase 2: After enough local peers are found, each node x in collective C_x creates a set G_x of potential *global peers*. For each collective $C_y \neq C_x$, G_x contains one node $y \in C_y$ with collective rank $r(y, C_y)$ such that

$$\left\lfloor r(x, C_x) \cdot \frac{|C_y|}{|C_x|} \right\rfloor \leq r(y, C_y) < \left\lceil (r(x, C_x) + 1) \cdot \frac{|C_y|}{|C_x|} \right\rceil$$

Node x then selects $max(|G_x|, N)$ nodes in G_x uniformly at random as its global peers, and initiates connections with them in the same manner as in phase 1.

Fig. 3 illustrates the peer selection process. With equally-sized clusters, potential global peers have the

same collective rank. With different-sized clusters, potential global peers are selected uniformly at random from an equal share of all nodes. This strategy ensures that the connections between nodes in different clusters are well spread out over all clusters and their nodes.
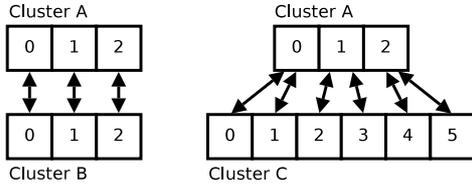

Fig. 3: Examples of peer selection between two clusters of the same size (A and B) and different sizes (A and C). Potential global peers are connected by an arrow.

Phase 3: After all connections are set up, data can be transferred. The data is logically split into P equally-sized pieces, numbered 0 to $P - 1$. At the start of a multicast operation, each node n provides a set of the indices of the pieces it already possesses, denoted as $possession(n)$. In a regular multicast operation, one root node possesses everything and the other nodes nothing. Other variations are also possible, like multi-root multicast (where multiple nodes have all data) or striping (where each node in a collective has a part of all data). As long as there is at least one *'root collective'* in which all nodes together possess all pieces, all nodes will receive all data (this is proven in Section 3.3).

The set of piece indices denoting the pieces a Robber node n wants to steal from a peer p is called its $desire(n, p)$. From local peers, a node desires all pieces it does not have. From global peers, a node n only desires the pieces that are part of its $work(n)$. Initially, the work of each node n in collective C consists of an equal share of all pieces i such that:

$$\left\lfloor r(n, C) \cdot \frac{P}{|C|} \right\rfloor \le i < \left\lfloor (r(n, C) + 1) \cdot \frac{P}{|C|} \right\rfloor$$

For example, with $P = 6$ total pieces and a cluster of 3 nodes (x, y, and z), $work(x) = \{0, 1\}$, $work(y) = \{2, 3\}$, and $work(z) = \{4, 5\}$. A node's work can change at runtime; once it has stolen all pieces that are part of its work from global peers, it tries to steal more work from one of its local peers. At any time, exactly one node in a collective is responsible for stealing a certain piece i remotely. By using this scheme, Robber transfers each piece to each cluster exactly once.

Phase 4: When a node has received all P pieces, it joins a final synchronization phase. Here, each node keeps serving requests from its peers until this is no longer necessary, so every node is able to finish. A node starts the synchronization phase by sending a *'done'* message to all its peers. Whenever it receives such a message from a peer, it remembers the peer is done. When a node and its peer are both done, they send a final *'stop'* message to each other. This *'stop'* message is the last message sent to a peer in a single multicast operation. When a node receives a *'stop'* message from a peer, it stops listening to it. A node finished a multicast operation once it stopped listening to all its peers.

In phase 3, nodes communicate with their peers using a variant of the BitTorrent protocol [8] using only *bitfield*, *have*, *request* and *piece* messages for stealing data. We add *desire*, *steal*, *work* and *found-work* messages for stealing work. Table 1 summarizes the format of each message.

TABLE 1: Format of messages used by Robber

Message(s)	Format
steal, found-work, done, stop	opcode (byte)
have, request	opcode (byte), piece index (integer)
bitfield, desire, work	opcode (byte), piece indices (list of booleans)
piece	opcode (byte), data (list of bytes)

A node n starts a multicast operation by sending a *'desire'* message to each peer p, telling p which pieces it wants. The answer is always a *'bitfield'* message, which tells n which pieces of its desire p has. Nodes remember the desire of their peers. After receiving the *'bitfield'* message, a node n starts sending *'request(i)'* messages to peer p instructing it to return piece i, which is always honored by p. The pieces to request are selected randomly from the set $desire(n, p) \cap possession(p)$. Whenever a node has received a new piece i, it informs all its local peers about it by sending a *'have(i)'* message. Global peers are only informed if piece i is part of their desire.

Let $pending(n)$ be the set of piece indices a Robber node n requested from peers, but has not received yet. During a multicast operation, n can be in three states:

Working: $|work(n)| > 0$
Unemployed: $|work(n)| = 0$
Retired: $|pending(n)| + |possession(n)| = P$

When a working node requests a piece from a peer, it removes it from its work. This happens until the node becomes unemployed or retired. A retired node starts the final synchronization phase. An unemployed node tries to steal work from local peers. Each node maintains a set of local peers that presumably have work, called its *local labor force*. A node n steals work by randomly selecting a peer p from its local labor force, and sending it a *'steal'* message. In reply, p removes the first $\lceil |work(p)| / 2 \rceil$ pieces from its own work and returns them to n in a *'work'* message. If p does not return any work, n knows that p is unemployed too. Node n then removes p from its local labor force, and retries stealing from another local peer. When none of n's local peers has any work, its local labor force becomes empty and the steal attempts stop to avoid needless polling. However, if a peer p does return some work, this becomes n's new work. The nodes then start the *new work sequence*:

1) Both n and p send their new desire to all their global peers, since n will now desire more pieces and p will desire less pieces.
2) Node n requests some newly-stolen pieces from each global peer if possession information is available from previous *'bitfield'* messages.

3) Node n sends a *'found-work'* message to all its local peers, who add n to their local labor force. If such a local peer is still unemployed, it restarts stealing.

The new work sequence ensures that n will eventually request at least one piece from a global peer after stealing work. The pieces n will learn about after step 1 can either be requested directly in step 2, or (if n's work is already stolen again before the *'bitfield'* messages arrive) after a successful steal later on. To bootstrap the local labor forces, each node sends a *'found-work'* message to all its local peers just after the initial *'desire'* message.

3.2 Benefits of load balancing

Robber, like MOB, divides the work over nodes in the same cluster. This alleviates the load on the bandwidth bottleneck in case of two common network scenarios, which we will call *'global bandwidth bottleneck'* and *'local bandwidth bottleneck'*. In the global bandwidth bottleneck scenario, the achievable bandwidth of the wide-area links between clusters is the overall bandwidth bottleneck. In this case, it is necessary to minimize the amount of wide-area communication. In Robber, each node will only inform a global peer p of a received piece if it is part of p's work. Only one *'have'* message per piece will therefore be sent to each cluster, and each piece will be transferred to each cluster only once. Together, this greatly relieves the load on the wide-area network.

In the local bandwidth bottleneck scenario, the overall bandwidth bottleneck is the local capacity of the wide-area network interfaces (NICs) of the nodes in a cluster. In Robber and MOB, nodes only receive data through their wide-area NIC that is part of their work. The bandwidth demand is thereby automatically spread over all wide-area NICs in the same cluster. This was already observed in Balanced Multicasting [7], but the setup there depended on network monitoring information; Robber and MOB achieve the same result automatically.

Both bandwidth bottleneck scenarios occur in the real world, even simultaneously. New technology keeps shifting common bottlenecks, but they never disappear. Currently, the dawn of optical interconnects is moving the bandwidth bottleneck from the wide-area links to the local capacity of individual nodes.

Unlike MOB, Robber uses work stealing to cope with slow nodes that achieve a lower incoming bandwidth from the WAN than other nodes in the same cluster. There are two reasons for having slow nodes. First, a node's wide-area NIC can be slower. As an example, imagine an old cluster of nodes with a FastEthernet NIC that is expanded with new nodes that all have Gigabit Ethernet cards. The former will at most receive 100 Mbit/s from global peers, whereas the latter can reach 1 Gbit/s. Second, a node can have slower connections to global peers than other nodes. This becomes more likely when the number of clusters increases, and nodes in the same collective have different sets of global peers. MOB's static equal work distribution causes the

node with the smallest incoming throughput t_{min} in a collective C to limit the total incoming throughput of that collective to $|C| * t_{min}$. The overall throughput of a multicast operation to all clusters will then be limited by the slowest collective. In Robber, however, fast nodes will steal work from slow nodes. The amount of data each node steals remotely is therefore automatically adapted to its relative speed within a collective.

3.3 Proof of Correctness

In this section, we will prove that the Robber algorithm is correct, i.e. it delivers all data to all nodes in a finite amount of time. We first prove this for a single collective, and then for the case of multiple collectives. In each case, we will also identify the initial distribution of the data over all nodes that is required to guarantee correctness.

3.3.1 Single Collective

Robber's communication network can be modeled as an undirected graph, in which vertices represent nodes and edges correspond to bidirectional connections between nodes. We will call this graph the *communication graph*.

Lemma 1. *Within a single collective, Robber eventually creates a connected local communication graph.*

Proof: In Robber, each node connects to 5 distinct other nodes chosen uniformly at random from the other nodes in the same collective. The resulting communication graph corresponds to a random m-orientable graph [28]. In this model, a digraph $DG_m^{(n)}$ starts with n separate vertices. For each vertex v, m distinct arcs (v, w) ($v \neq w$ and $1 \leq m < n$) are then randomly chosen and added to $DG_m^{(n)}$, such that each of the $\binom{n-1}{m}$ possible sets of arcs has the same probability of being chosen. The undirected graph $G_m^{(n)}$ is obtained from $DG_m^{(n)}$ by ignoring the orientation of the arcs. The connectedness of $G_m^{(n)}$ is predictable as shown in [29]:

$$\text{For } m \geq 2, \ \lim_{n \to \infty} Prob(G_m^{(n)} \text{ is connected}) = 1 \quad (1)$$

In Robber, $m = 5$, making it even more likely that the communication graph is connected. However, in practice $n \ll \infty$. It is easy to see that for $n < 12$, $G_5^{(n)}$ is always connected. To the best of our knowledge, no exact formula is known for calculating this probability for $n \geq 12$. However, the simulations in [29] demonstrate that it is very high in practice (i.e. larger than 0.9999). All nodes in the same cluster therefore repeatedly generate a random communication graph in memory that is tested for connectedness before initiating the actual connections. Due to the high probability of connectedness, usually only one graph needs to be generated. □

Theorem 1. *All pieces possessed by nodes in a collective will be transferred to all nodes in that collective.*

Proof: All local peers will report all pieces they have to each other using *'bitfield'* and *'have'* messages. Each

pair of local peers will exchange pieces until they both have all each other's pieces too. Since the local communication graph is connected (Lemma 1), there exists at least one path $n_0, n_1, \ldots, n_x$ between any nodes n_0 and n_x in a collective. Each node in this path can request all pieces from its neighbors in the path. As a result, n_0 and n_x will be able to exchange all their pieces, either directly or via intermediate nodes in a path between them. Consequently, all nodes in a collective will be able to receive all pieces available in the collective. □

3.3.2 Multiple Collectives

To proof Robber's correctness in the case of multiple collectives, we first define the *global communication graph*, in which vertices represent collectives and edges denote connections between global peers in these collectives.

Lemma 2. *The global communication graph is connected.*

Proof: Since each collective consists of at least one node, the minimum degree of each vertex in the global communication graph is 5. Like the local communication graph, the global communication graph therefore already has a high probability of being connected. To ensure connectedness, it is also first generated in memory and tested for connectedness before initiating the connections between global peers. □

Definition 1. A *root collective* is a collective in which all nodes together have all data.

If there is no root collective, the data to distribute is scattered over multiple nodes in different collectives. In that case, a deadlock situation can occur. Fig. 3.3.2 shows an example deadlock situation. The global communica-
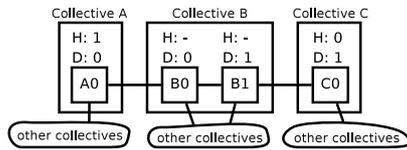

Fig. 4: Example deadlock situation without a root collective.

tion graph in this example contains two bridges: (a_0, b_0) and (b_1, c_0). The multicast data consists of two pieces (0 and 1). The nodes in collectives A, B, and C are annotated with the pieces they initially have ('H') and desire ('D'). The nodes in all other collectives have no pieces at all. Both nodes in collective B desire a piece that none of their global peers has, which creates a deadlock.

We will now prove that having *at least one* root collective guarantees Robber's correctness, since all its neighbors in the global communication graph will then eventually become root collectives too.

Theorem 2. *All pieces possessed by all nodes are transferred to all other nodes if there is at least one root collective.*

Proof: If there is at least one root collective C_x, it follows from Theorem 1 that each node $x \in C_x$ will eventually possess all data too. Initially, all global peers

y of x will therefore be able to request all the pieces they desire, either from x itself or from some other global peers. When y's desire has become empty it will become 'unemployed'. Before it can request more pieces from x, y must find new work by stealing it from its local peers in collective C_y. If there is work left in C_y, all nodes in C_y are either 'working' or 'unemployed' (the presence of 'retired' nodes implies that all work has been done, and y will receive any remaining pieces via its local peers). If y has at least one working local peer, it will steal some work from it and continue requesting pieces from x. If all local peers of y are unemployed, y has to wait until they have found new work.

In general, each unemployed node $n \in C_y$ will try to steal work from its local peers. If any of them has work left, n will steal at least one new piece of work. If none of n's local peers has any work left either, they will be trying to steal work from their local peers as well.

Since the local communication graph is connected (Lemma 1), there will be a path $n_0, n_1, \ldots, n_x$ between any unemployed node n_0 and some working node n_x, possibly via intermediate unemployed nodes $n_1, \ldots, n_{x-1}$ that propagate half of the work over each hop in the path with each successful steal. If $|work(C_y)| \geq |C_y|$, the work stealing will spread out the work over all nodes in C_y and no node will be unemployed. In this case, node y can keep requesting pieces from node x

For the last pieces of work, $|work(C_y)| < |C_y|$. In that case, there will always be some unemployed node trying to steal work. Each remaining piece of work w will then perform a *random walk* over all nodes until some node does the work and steals piece w remotely. If all pieces are stolen remotely this way, no work will be left in C_y and it becomes a root collective. If some work w would never be done, it would wander around in the collective forever. However, since the local communication graph is finite, w will eventually reach each node [30]. Each unemployed node is therefore always able to find work.

One of the unemployed nodes that will eventually find new work is node y with its global peer x in root collective C_x. Robber's 'new work sequence' then ensures that y always makes progress. Only after step 3 of the new work sequence, unemployed local peers may steal some of y's work. However, by then either y's knowledge of the possession of its global peers has increased or will increase (in step 1), and/or y requested one or more pieces from its global peers (in step 2).

Since y will eventually find new work and make progress with it, node y will always continue requesting data from x in root collective C_x. All pieces will therefore eventually be transferred to C_y (either by y itself or by another node in C_y) and all nodes in C_y will eventually receive all data too (Theorem 1). Consequently, all neighbors of a root collective in the global communication graph will eventually become root collectives too. From Lemma 2 then follows that all nodes will eventually receive all data. □

3.4 Implementation

Fig. 5 shows a high-level view of the software layers in our implementation, and Robber's implementation in more detail. We have implemented Robber, as well as BitTorrent, MOB, and Balanced Multicasting (BM) on top of Ibis [11], our Java-based Grid programming environment. We used the SmartSockets library [31] to emulate different clusters inside a single cluster by providing configurable custom routing of data, which we used to evaluate our algorithms (see Section 4.1 for details).

Ibis provides all nodes with the names and ranks of all other nodes and the names of their clusters, which is all the input data Robber and MOB need. Balanced Multicasting also needs network monitoring data information. For the experiments in Section 4.2, we use the input for the emulation itself as monitoring data.
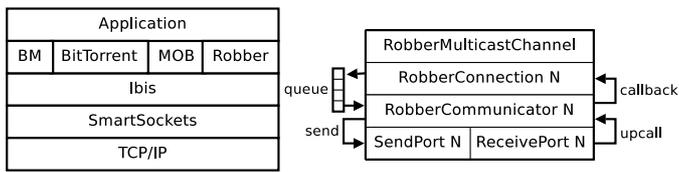

Fig. 5: Software layers and details of Robber's implementation

Each of the four multicast methods is encapsulated in a MulticastChannel object. Fig. 5 shows the implementation of the RobberMulticastChannel object. All connections to peers are created in the constructor. Each connection is abstracted in a RobberConnection object. The underlying RobberCommunicator object sends and receives protocol messages using Ibis' *send port* and *receive port* primitives. All received messages are processed in asynchronous upcalls provided by Ibis, and translated to individual callback functions in the RobberConnection object on top. Outgoing data is put into a queue and sent in a separate thread per connection to avoid deadlocks. Data can be multicast by invoking the same method on a RobberMulticastChannel object on each node, which returns when all data has been received. Received data can only be altered after calling RobberMulticastChannel.flush(), which waits until all peers have received all data too.

4 EVALUATION

We have evaluated Robber by comparing its performance to that of BitTorrent, MOB, and Balanced Multicasting in four test cases. The first two test cases consist of several *'global bottleneck'* scenarios of various dynamics and size. The second two test cases are examples of *'local bottleneck'* scenarios, and show that Robber achieves the same optimized throughput between clusters as Balanced Multicasting, without needing any external monitoring data. Finally, we have analyzed the communication overhead and computational overhead of Robber and MOB.

4.1 Emulation Setup

We emulated the various WAN scenarios in all test cases within one cluster of the Distributed ASCI Supercomputer 3 (DAS-3) [12]. Each node in the DAS-3 is equipped with two 2.4 GHz AMD Opterons and a 10 Gbit Myrinet network card for fast local communication.

Using emulation enabled us to precisely control the environment and subject each multicast method to exactly the same network conditions without any interfering background traffic. This ensured a fair comparison and reproducible results. The emulation only concerns the network performance. All application nodes run the real application code (Ibis and one of the four multicast protocols on top).
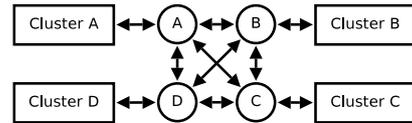

Fig. 6: Emulation setup with four clusters. The hub nodes A, B, C, and D route data between clusters and apply traffic control.

Fig. 6 shows the setup of four emulated clusters, as used in the first test case. The other test cases use an identical setup, except for the amount of clusters. Nodes in the same emulated cluster communicate directly, but traffic between nodes in different emulated clusters is routed via two special *'hub'* nodes using SmartSockets. Besides routing inter-cluster traffic, the hubs also emulate the wide-area bandwidth and delay between clusters using the Linux Traffic Control (LTC) kernel module [32] to slow down outgoing Myrinet traffic. Bandwidth is emulated using HTB qdiscs [33], while one-way delay is emulated using Netem qdiscs [34]. The qdiscs use a default maximum queue length of 1000 packets. The hubs apply simple end-to-end flow control to slow down a source node in case of 'congestion'. All qdiscs together emulate incoming and outgoing capacity of nodes and clusters, and delay and bandwidth between clusters (i.e. all arrows in our network model Fig. 2.2).

Fig. 7 shows a detailed example of two emulated clusters A and B. Each cluster contains one application node (x and y) and one hub node. All nodes are connected by SmartSockets connections, shown as solid lines. The LTC qdiscs used in all nodes for slowing down outgoing traffic are drawn as dotted lines. For clarity of presentation, only two clusters and two nodes are shown; with more clusters or nodes, there would be multiple versions of the qdiscs marked with an asterisk, one for each destination cluster or destination node. A packet sent from node $x \in A$ to node $y \in B$ passes several qdiscs that emulate all the network characteristics between nodes x and y. The packet first passes a qdisc on node x that emulates x's outgoing capacity. It is then sent to the hub node of cluster A, where it passes three qdiscs that emulate the outgoing capacity of cluster A, the bandwidth from A to B and the delay from A to B. The packet is then sent to the hub node of cluster B, where it passes two
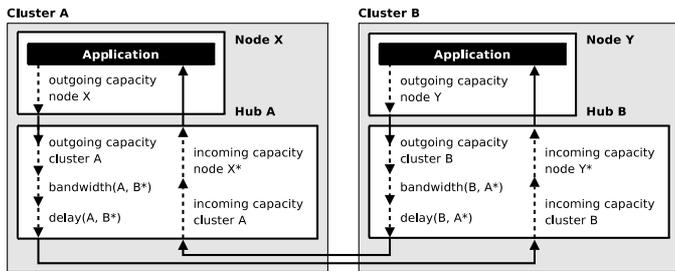
Fig. 7: Detailed view of two clusters A and B, and the qdiscs (dotted lines) and SmartSockets connections (solid lines) over which data between two global peers $x \in A$ and $y \in B$ is sent.

qdiscs that emulate the incoming capacity of cluster B and the incoming capacity of node Y. Finally, the packet is delivered to node y.

4.2 Global Bottleneck Test Cases

The first two test cases evaluate the performance of BitTorrent, MOB, Robber and Balanced Multicasting in various *'global bottleneck'* environments. In the first test case, we compare the performance of all multicast methods under fluctuating wide-area bandwidth. The second test case demonstrates the emergence of slow nodes in larger environments, and shows the benefit of Robber's load balancing strategy.

4.2.1 Test Case 1: Dynamic Wide-area Bandwidth

The first test case consists of five WAN scenarios with different dynamics:

1) **fast links**: the WAN bandwidth on all links is stable and set according to Fig. 8(a) (all links are annotated with their emulated bandwidth, in MB/s).
2) **slow links**: like scenario 1, but with the bandwidth of links $A \leftrightarrow D$ and $B \leftrightarrow C$ set to 0.8 MB/s (indicated by the dotted lines in Fig. 8(b)).
3) **fast $\rightarrow$ slow**: like scenario 1 for 30 seconds, then like scenario 2, emulating a drop in throughput on two WAN links
4) **slow $\rightarrow$ fast**: like scenario 2 for 30 seconds, then like scenario 1, emulating an increase in throughput on two WAN links
5) **mayhem**: like scenario 1, but every 5 seconds all links randomly change their bandwidth between 10% and 100% of their nominal values, emulating heavy background traffic. The random generator is always initialized with the same seed to ensure that this scenario uses identical fluctuations every time.

In all five scenarios, the one-way delay of the wide-area links is set to 10 ms. We emulate four clusters of 16 nodes each. One root node in cluster A sends 600 MB to all other nodes, using four different multicast methods (BitTorrent, MOB, Robber, and Balanced Multicasting) in the five scenarios described above. In each scenario, Balanced Multicasting uses the exact *initial* emulated bandwidth values as input for its algorithm. We also compute the theoretical maximum throughput in each scenario by converting all 16 possible multicast trees

(a) scenario 1: A-D and B-C are fast links
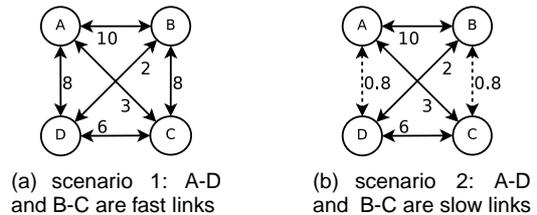
(b) scenario 2: A-D and B-C are slow links

Fig. 8: The two static WAN scenarios in test case 1. The wide-area links are annotated with their emulated bandwidth in MB/s.
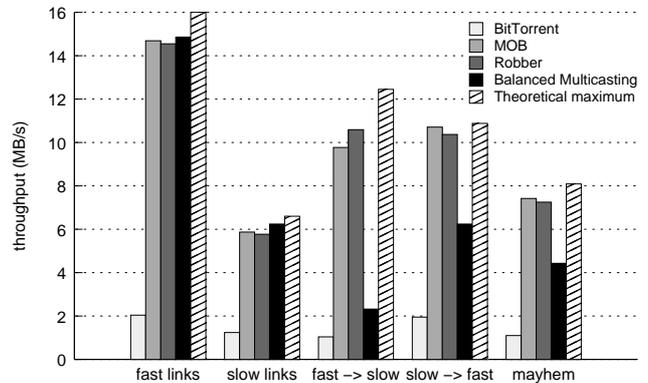

Fig. 9: Multicast throughput between four clusters of 16 nodes each. The root node sends 600 MB to all others using four different methods.

between the four clusters to a linear program. For the maximum throughput in the dynamic scenarios, we assume that the theoretical algorithm can adapt instantly to the optimal strategy in each new situation.

Fig. 9 shows for each scenario the throughput of each multicast method, calculated as 600 MB divided by the time passed between the moment the root started the multicast and the moment the last node received all data. Table 4.2.1 shows these throughput values as a percentage of the theoretical maximum throughput in each WAN scenario. It can be seen that BitTorrent always performs worst, which is caused by the overhead it creates by sending duplicate WAN messages. The difference between MOB and Robber shows the small overhead of Robber's extra load-balancing communication (which is not really needed in this case, since all nodes in each cluster are equally fast). Robber and MOB perform similar to Balanced Multicasting in the static scenarios, and outperform it in all three dynamic ones. In the first dynamic scenario *'fast $\rightarrow$ slow'*, Balanced Multicasting overuses the links that became slow and

TABLE 2: Percentage of the theoretical maximum throughput each multicast method achieves in the five WAN scenarios.

|	BitTorrent	MOB	Robber	BM
fast links	13%	92%	91%	93%
slow links	13%	92%	91%	93%
fast $\rightarrow$ slow	8%	78%	85%	19%
slow $\rightarrow$ fast	18%	98%	95%	57%
mayhem	14%	92%	90%	55%

does not adapt, for which it is heavily penalized. In contrast, Robber and MOB adapt and use the other WAN links to distribute the data, resulting in much higher throughput. In the second dynamic scenario *'slow → fast'*, Balanced Multicasting does not use the extra achievable bandwidth that became available on two WAN links due to its sender-side traffic shaping. It therefore achieves the same throughput as in the *'slow links'* scenario, whereas Robber and MOB greedily use the extra bandwidth that became available. In the last dynamic scenario *'mayhem'*, the average throughput of all links is 55% of that in the *'fast links'* scenario. Balanced Multicasting actually achieves 30% of its throughput in the *'fast links'* scenario, since it continuously uses the same WAN links and the throughput of the bottleneck link in each of its multicast trees determines its overall throughput. Robber and MOB are much more adaptive and far better in using all the available wide-area bandwidth, achieving 50% of their throughput in the *'fast links'* scenario. Their throughput is also close to the theoretical maximum, even in the dynamic scenarios.

4.2.2 Test Case 2: Slow Nodes

In our second test case, we emulate 4 to 16 clusters of four nodes each, located in four different 'continents': Europe, America, Asia and Africa. Clusters in the same continent are connected by high-performance links of 5 MB/s and 10 ms one-way delay. Clusters in different continents are connected by slower connections of 500 KB/s and 100 ms one-way delay. In the experiment, one European node sends 100 MB to all others using BitTorrent, MOB, Robber and Balanced Multicasting.

Fig. 10(a) shows that the throughput achieved by MOB, Robber and Balanced Multicasting increases with the number of clusters per continent. This is logical, since more clusters allow for more intercontinental connections which results in more available bandwidth. For BitTorrent, the throughput remains almost constant since the amount of communication there increases linearly with with the total number of nodes. In Robber and MOB, the amount of communication increases linearly with the total number of clusters, which scales much better. With more clusters, Robber increasingly outperforms MOB. The larger the environment, the more likely it is that nodes in the same cluster have a different number of peers in clusters on other continents. Nodes with more of these 'foreign' peers will therefore be slower than others, and limit the overall throughput of MOB. Robber's work stealing mechanism overcomes this imbalance, which results in higher throughput.

The winner in this test case seems to be Balanced Multicasting, which outperforms all other methods. This is not surprising, since Balanced Multicasting does not exchange any additional meta-data like the other methods do. However, its good performance depends heavily on accurate monitoring data to determine the optimal routing. To illustrate this dependency, we conduct a second experiment in which all links between Europe and
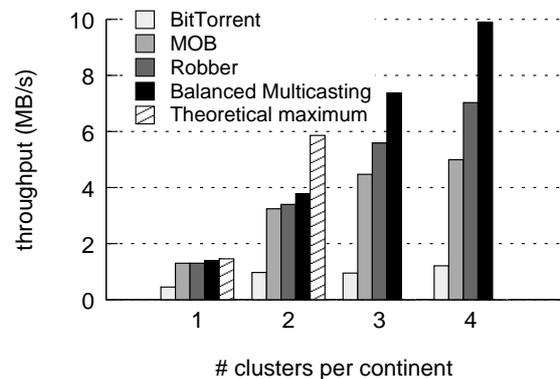
(a) Links between continents are 500 KB/s
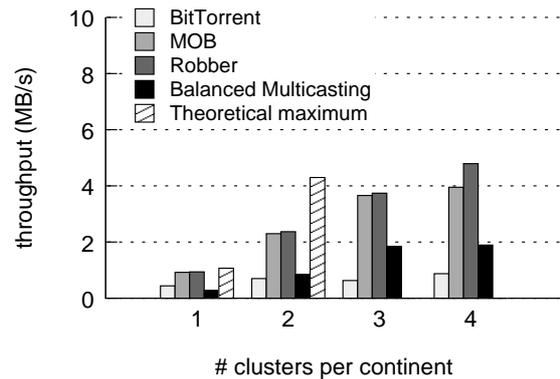
(b) Links between Europe and America turn out to be 100 KB/s

Fig. 10: Multicast throughput of four different multicast methods between clusters in four different continents. The European root node sends 100 MB to all others using four different methods.

America turn out to be 100 KB/s instead of 500 KB/s. This is not known by Balanced Multicasting, which still assumes 500 KB/s links. Fig. 10(b) shows the result of this imitated measurement error. Because there is now less bandwidth available, all multicast methods perform worse than before. Balanced Multicasting now overuses the links between Europe and America, causing its throughput to plummet. In contrast, Robber and MOB automatically adapt and still perform very well.

We also calculate the theoretical maximum throughput in each case, which is only feasible up to 8 clusters. With more clusters, the linear program becomes too large to fit into memory, which demonstrates the need for heuristic approaches. Furthermore, the optimal solutions use many multicast trees (e.g. 41 for 8 clusters) and assume zero overhead, which is not very realistic.

4.3 Local Bottleneck Test Cases

The second two test cases compare BitTorrent, MOB, Robber and Balanced Multicasting in several *'local bottleneck'* environments. The first test case uses homogeneous clusters, the second test case uses heterogeneous clusters.

4.3.1 Test Case 3: Homogeneous Clusters

In our third test case, we emulate nodes that use a 100 Mbit/s NIC for wide-area communication. Nodes
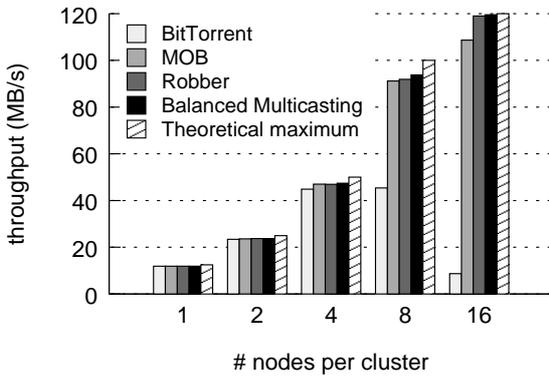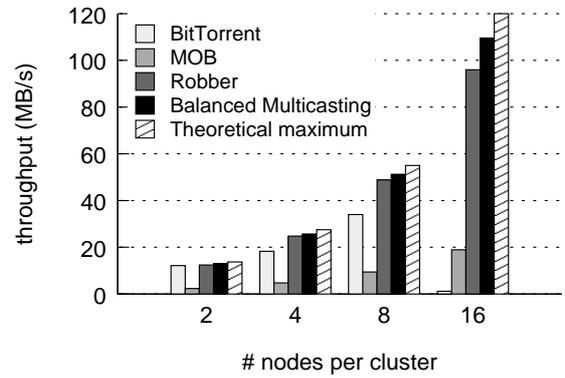
(a) Two clusters
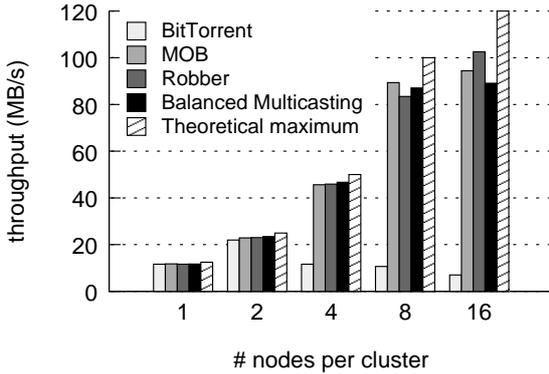
(b) Four clusters

Fig. 11: Multicast throughput between multiple homogeneous clusters. Each node has a 100 Mbit NIC. The root node sends 1 GB to all others using four different methods.

(a) Two clusters

(b) Four clusters

Fig. 12: Multicast throughput between multiple heterogeneous clusters. Half of the nodes in each clusters has a 10 Mbit NIC, the other half has a 100 Mbit NIC. The root node sends 512 MB to all others using four different methods.

in the same cluster communicate directly without any traffic shaping, which 'emulates' a separate high-speed interconnect. All wide-area links are set to 1 Gbit/s and have a round-trip time of 10 ms. The overall setup resembles the layout of our former Distributed ASCI Supercomputer 2 (DAS-2) [35].
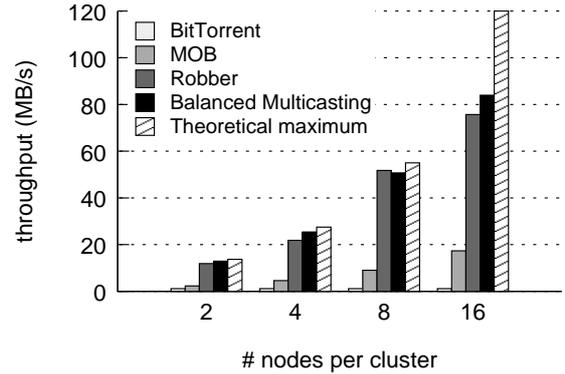
We perform two experiments: one with two, and one with four emulated clusters. In each experiment, we use BitTorrent, MOB, Robber, and Balanced Multicasting to send 1 GB from a root node to all others, using up to 16 nodes per cluster. We also calculate the theoretical maximum throughput for each case.

Fig. 11 shows that both MOB, Robber, and Balanced Multicasting split the data over multiple NICs in parallel, thereby overcoming the throughput of a single wide-area NIC of the nodes. BitTorrent mimics this behavior, but its throughput decreases quickly when the number of nodes increases. In the case of a small environment, the chance is fairly high that some BitTorrent nodes will have one or more peers in the same cluster. These nodes will then spontaneously behave like a collective, and achieve similar throughput. Yet with more nodes per cluster, the chance of this happening decreases quickly, and the amount of duplicate pieces transferred to the same cluster slows down BitTorrent's throughput significantly.

The ultimate bottleneck in this experiment is the

achievable bandwidth between clusters. The local capacity of the clusters and the wide-area bandwidth is set to 1 Gbit/s, which limits the maximum throughput to about 120 MB/s. Both MOB, Robber and Balanced Multicasting get very close to the theoretical maximum throughput, although with four larger clusters the impact of practical overhead starts to show.

4.3.2 Test Case 4: Heterogeneous Clusters

The fourth and last test case is similar to the test case with homogeneous clusters, except that half of the nodes now has a 10 Mbit/s NIC instead of a 100 Mbit/s NIC. We emulate two and four of these clusters, transfer 512 MB from one root node to all others using all four multicast methods and measure the overall throughput.

Fig. 12 shows that MOB does not benefit at all from the faster network cards. In MOB, the 'slow' nodes with a 10 Mbit NIC steal the same amount of data as the 'fast' nodes with a 100 Mbit NIC. The slow nodes therefore determine the overall throughput. With N nodes per cluster, each node steals $1/Nth$ of all data, which limits MOB's throughput to $N \cdot 10$ Mbit in these experiments.

BitTorrent does benefit from the fast nodes, but only with two clusters and for 2, 4, and 8 nodes per cluster. In those cases, its throughput is close to that of Robber and

Balanced Multicasting. In all other cases, its throughput drops to 10 Mbit/s, which can be explained by considering BitTorrent's communication graph (Section 3.3). Each edge in the communication graph is now either slow or fast, depending on whether a peer is located in the same cluster and the speed of a node's wide-area NIC. If there is a path from the root to a node n consisting of fast edges only, n will be able to receive pieces at a rate of 100 Mbit/s. If all nodes are connected to the root by such a 'fast path', the overall throughput will also be 100 Mbit/s. With multiple fast paths per node, the throughput can increase even further. However, if just one node is not connected to the root by a fast path, its throughput will be limited by the slow nodes. The chance of having a fast path decreases when the number of nodes and clusters increases, which explains BitTorrent's low throughput in these cases.

The clear winners in these experiments are Balanced Multicasting and Robber. Both are able to harvest the performance of fast nodes, and reach a higher throughput given more nodes per cluster. The difference with the theoretical maximum is marginal, and increases for larger and more clusters due to practical overhead. Balanced Multicasting is a little faster than Robber since the former does not have to transfer any additional metadata. However, Robber does not need any additional monitoring data to reach this throughput, while such information is crucial for Balanced Multicasting.

4.4 Protocol Overhead

We analyze both communication and computational overhead of MOB and Robber using test cases 3 and 4, which cover environments of various size and heterogeneity. Doing so, we trigger Robber's additional load balancing mechanism.

We express the *communication* overhead as a percentage of the total amount of multicast data D. During each multicast operation, we record B: the total amount of bytes sent and received by all N nodes. The communication overhead is then calculated as $\left(\frac{B}{2(N-1)} - D\right)/D \cdot 100$.

Fig. 13 shows the average communication overhead of MOB and Robber in test cases 3 and 4. The overhead grows sub-linearly with the number of nodes and clusters, and is always less than 1%. Robber sends slightly more meta-data over the WAN because of the updated *'desire'* messages after a successful steal. With larger clusters, MOB generates more overhead by sending more local *'have'* messages. To measure the overhead for small amounts of data, we repeated the experiments with 10 MB and 5 MB of multicast data in test case 3 and 4, respectively. The communication overhead then becomes even less because of the smaller number of pieces and hence the smaller *'bitfield'* and *'desire'* messages.

Next, we quantify the *computational* overhead of MOB and Robber by measuring their CPU usage per core. After each multicast operation, we retrieve the total system and user time of all Java threads via JMX and
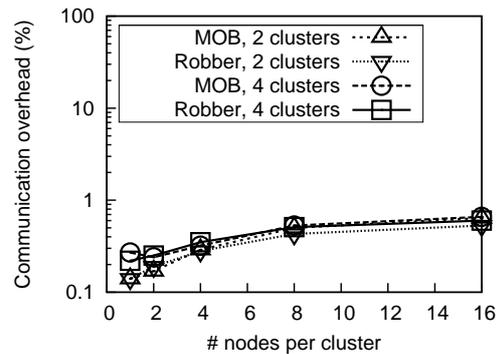
(a) Test case 3
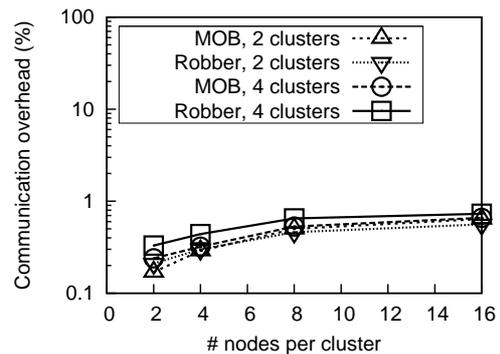
(b) Test case 4

Fig. 13: Communication overhead of MOB and Robber in test cases 3 and 4.

divide that by the total number of nodes and the number of cores per node (i.e. four in our DAS-3 testbed).

Fig. 14 shows the overall completion time of each experiment in test cases 3 and 4. The solid black area in the bottom of each bar denotes the CPU usage time per core. We can see that the CPU usage of MOB and Robber is comparably low: at most 2 seconds. It is also a small fraction of the overall completion time, showing that the throughput in these experiments is network-bound.

5 CONCLUSIONS

The completion time of large-data multicast transfers depends primarily on the bandwidth an application can achieve across the interconnection network. Traditionally, multicasting is implemented using a sender-initiated approach in which the application nodes are arranged in spanning trees over which the data are sent. However, the bandwidth between sites in a grid environment can be significantly different and also dynamically changing. To remain efficient, existing methods compute optimal multicast trees based on network monitoring information. This approach, however, is problematic because it assumes network monitoring systems to be deployed ubiquitously. It also assumes monitored data to be both accurate and stable during a multicast operation, which might not be the case in shared networks with variable background traffic. Our Balanced Multicasting approach indeed suffers from all these problems.
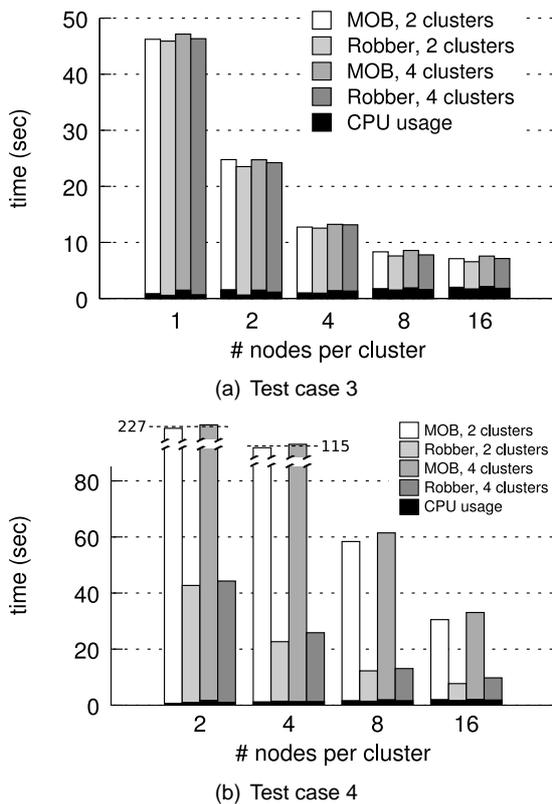
(a) Test case 3

(b) Test case 4

Fig. 14: Overall completion time and actual CPU usage per core of MOB and Robber in test cases 3 and 4.

TABLE 3: Suitability of Balanced Multicasting (BM), BitTorrent, MOB, and Robber in various cases.

	BM	BitTorrent	MOB	Robber
Individual hosts	+	+	+	+
Clusters of the same size	+	-	+	+
Clusters of different sizes	+	-	+	+
Homogeneous clusters	+	-	+	+
Heterogeneous clusters	+	-	-	+
Large number of clusters	o	-	o	+
Heterogeneous WAN bandwidth	+	+	+	+
Fluctuating WAN bandwidth	-	+	+	+
Ease of deployment	-	+	+	+
Legend: good (+) mediocre (o) bad (-)				

We have implemented Robber (as well as the BitTorrent protocol, MOB, and Balanced Multicasting) within our Java-based Ibis system. We have experimentally evaluated the four approaches by emulating various WAN scenarios. We compared BitTorrent, MOB and Robber to Balanced Multicasting (as a practical, (close-to) optimal solution using spanning trees that can be found with complete network performance information) and the theoretical maximum throughput. Dynamically changing network performance, however, can not be taken into account by Balanced Multicasting.

Our experimental evaluation shows that Robber consistently achieves high throughput in a variety of bandwidth bottleneck scenarios. Both Robber and MOB outperform the original BitTorrent protocol by wide margins due to the much better utilization of the wide-area network. In the case of stable wide-area bandwidth, the multicast throughput of Robber and MOB is similar to that of Balanced Multicasting. When the wide-area bandwidth changes in the middle of a run, Balanced Multicasting drastically suffers from dynamically changing bandwidth as its pre-computed spanning trees no longer match the actual network conditions. Robber and MOB automatically adapt to the new situation and perform much better. In large environments or heterogeneous clusters, MOB's throughput is severely limited by slow nodes. In theses cases, Robber automatically tunes the amount of work of a node to its relative performance in a collective, which results in much higher throughput. This holds for the bandwidth bottleneck being either in the WAN (the global bottleneck case) or in the LAN (the local bottleneck case). The communication and computational overhead of the Robber protocol is very low, and comparable with MOB.

To summarize, Robber significantly improves achievable multicast bandwidth without needing any network performance monitoring data. Table 5 lists the variety of network conditions and imbalanced clusters Robber can cope with, making it an ideal strategy for multicasting in multi-cluster grid environments.

Receiver-initiated multicast approaches avoid static routing of data over trees and distribute data randomly over a peer-to-peer mesh. Instead of planning multicast trees based on potentially outdated monitoring data, they implicitly use the network links between all clusters, while automatically adapting to the currently achievable bandwidth ratios. Previously, we developed MOB, a receiver-initiated multicast approach based on BitTorrent. MOB nodes in the same cluster team up in a collective that tries to steal data from other clusters as efficiently as possible. Each node steals an equal share of all data remotely, and distributes stolen data locally. This strategy reduces the amount of wide-area communication significantly compared to BitTorrent. However, MOB's static load balancing scheme becomes inefficient in larger or more heterogeneous grid environments.

In this paper, we have presented *Robber*, a receiver-initiated multicast approach that combines collective data stealing with load balancing. In Robber, the amount of data a node steals remotely is treated as 'work'. Initially, the work is divided equally among all nodes in a collective, but once a node become idle it tries to steal work from local peers. This way, fast nodes will steal more data remotely than slow nodes.

We have proven that the Robber multicast algorithm eventually delivers all data to all nodes. With multiple clusters, there has to be at least one root cluster where all nodes together have all data to guarantee correctness.

REFERENCES

[1] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, and B. Wallenfelt, "Massive Parallel BLAST for the Blue Gene/L," in *High Availability and Performance Computing Workshop 2005 (HAPCW2005)*, Sante Fe, NM, USA, Oct 11 2005.

[2] F. Seinstra, J. Geusebroek, D. Koelma, C. Snoek, M. Worring, and A. Smeulders, "High-Performance Distributed Image and Video Content Analysis with Parallel-Horus," *IEEE Multimedia*, vol. 14, no. 4, pp. 64–75, Oct-Dec 2007.

[3] O. Beaumont, L. Marchal, and Y. Robert, "Broadcast Trees for Heterogeneous Platforms," in *19th Int. Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, Apr 3-8 2005.

[4] R. Izmailov, S. Ganguly, and N. Tu, "Fast Parallel File Replication in Data Grid," in *Future of Grid Data Environments workshop (GGF-10)*, Berlin, Germany, Mar 9 2004.

[5] 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," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 131–140, May 4-6 1999.

[6] B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, and M. Swany, "A Hierarchy of Network Performance Characteristics for Grid Applications and Services," Proposed Recommendation GFD-R-P.023, Global Grid Forum, 2004.

[7] M. den Burger, T. Kielmann, and H. E. Bal, "Balanced Multicasting: High-throughput Communication for Grid Applications," in *Supercomputing 2005 (SC05)*, Seattle, WA, USA, Nov 12-18 2005.

[8] B. Cohen, "Incentives Build Robustness in BitTorrent," in *1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, Jun 5-6 2003.

[9] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr, "Chainsaw: Eliminating Trees from Overlay Multicast," in *4th Int. Workshop on Peer-to-Peer Systems (IPTPS 2005)*, Ithaca, NY, USA, Feb 24-25 2005.

[10] M. den Burger and T. Kielmann, "MOB: Zero-configuration High-throughput Multicasting for Grid Applications," in *16th IEEE Int. Symposium on High Performance Distributed Computing 2007 (HPDC'07)*, Monterey, CA, USA, Jun 27-29 2007.

[11] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal, "Ibis: A Flexible and Efficient Java-based Grid Programming Environment," *Concurrency & Computation: Practice & Experience*, vol. 17, no. 7-8, pp. 1079–1107, Jun-Jul 2005.

[12] (2006) The Distributed ASCI Supercomputer 3. [Online]. Available: http://www.cs.vu.nl/das3/

[13] R. Cohen and G. Kaempfer, "A Unicast-based Approach for Streaming Multicast," in *20th Annual Joint Conf. of the IEEE Computer and Communications Societies (IEEE INFOCOM 2001)*, Anchorage, AK, USA, Apr 22-26 2001, pp. 440–448.

[14] M. Kim, S. Lam, and D. Lee, "Optimal Distribution Tree for Internet Streaming Media," in *23rd Int. Conf. on Distributed Computing Systems (ICDCS '03)*, Providence, RI, USA, May 19-22 2003.

[15] Y. Cui, Y. Xue, and K. Nahrstedt, "Max-min Overlay Multicast: Rate Allocation and Tree Construction," in *12th IEEE Int. Workshop on Quality of Service (IwQoS '04)*, Montreal, Canada, Jun 7-9 2004.

[16] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments," in *19th ACM Symposium on Operating System Principles (SOSP-19)*, Bolton Landing, NY, USA, Oct 19-22 2003.

[17] R. Wolski, "Experiences with Predicting Resource Performance On-line in Computational Grid Settings," *ACM SIGMETRICS Perf. Evaluation Review*, vol. 30, no. 4, pp. 41–49, Mar 2003.

[18] T. Gross, B. Lowekamp, R. Karrer, N. Miller, and P. Steenkiste, "Design, Implementation and Evaluation of the Remos Network," *Journal of Grid Computing*, vol. 1, no. 1, pp. 75–93, May 2003.

[19] J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep, and M. den Burger, "Middleware Adaptation with the Delphoi Service," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 13, pp. 1659–1679, Nov 2006.

[20] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance," in *14th Int. Parallel and Distributed Processing Symposium (IPDPS '00)*, Cancun, Mexico, May 1-5 2000, pp. 377–384.

[21] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, and R. Hofman, "Network Performance-aware Collective Communication for Clustered Wide Area Systems," *Parallel Computing*, vol. 27, no. 11, pp. 1431–1456, 2001.

[22] K. Verstoep, K. Langendoen, and H. Bal, "Efficient Reliable Multicast on Myrinet," in *Int. Conf. on Parallel Processing (ICPP 1996)*, vol. 3, Bloomingdale, IL, USA, Aug 12-16 1996, pp. 156–165.

[23] Y. Cui, B. Li, and K. Nahrstedt, "On Achieving Optimized Capacity Utilization in Application Overlay Networks with Multiple Competing Sessions," in *16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*. Barcelona, Spain: ACM Press, Jun 27-30 2004, pp. 160–169.

[24] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh," in *19th ACM Symposium on Operating System Principles (SOSP-19)*, Bolton Landing, NY, USA, Oct 19-22 2003.

[25] R. Bindal, P. Cao, W. Chan, J. Medval, G. Suwala, T. Bates, and A. Zhang, "Improving Traffic Locality in BitTorrent via Biased Neighbor Selection," in *26th Int. Conf. on Distributed Computing Systems (ICDCS 2006)*, Lisboa, Portugal, Jul 4-7 2006.

[26] J. Pouwelse, P. Garbacki, J. W. A. Bakker, J. Yang, A. Iosup, D. Epema, M.Reinders, M. van Steen, and H. Sips, "Tribler: A Social-based Peer-to-Peer System," in *5th Int. Workshop on Peer-to-Peer Systems (IPTPS'06)*, Santa Barbara, CA, USA, Feb 27-28 2006.

[27] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *IEEE/INFOCOM'05*, Miami, FL, USA, Mar 13-17 2005.

[28] T. Fenner and A. Frieze, "On the Connectivity of Random m-orientable Graphs and Digraphs," *Combinatorica*, vol. 2, no. 4, pp. 347–359, 1982.

[29] W. Si and M. Li, "On the Connectedness of Peer-to-Peer Overlay Networks," in *11th Int. Conf. on Parallel and Distributed Systems (ICPADS'05)*, vol. 1, Fukuoka, Japan, Jul 20-22 2005, pp. 474–480.

[30] K. Chung, *Markov Chains with Stationary Transition Probabilities*. Springer-Verlag, 1967, ch. 3.

[31] J. Maassen and H. E. Bal, "Solving the Connectivity Problems in Grid Computing," in *16th IEEE Int. Symposium on High-Performance Distributed Computing (HPDC'07)*, Monterey, CA, USA, Jun 27-29 2007.

[32] Linux Advanced Routing and Traffic Control. [Online]. Available: http://lartc.org/

[33] M. Devera. HTB Linux Queuing Discipline Manual - User Guide. [Online]. Available: http://luxik.cdi.cz/~devik/qos/htb/

[34] S. Hemminger, "Network Emulation with NetEm," in *Linux Conf Au*, Canberra, Australia, Apr 18-23 2005.

[35] (2002) The Distributed ASCI Supercomputer 2. [Online]. Available: http://www.cs.vu.nl/das2/

Mathijs den Burger received his M.Sc. and Ph.D. degrees in Computer Science from Vrije Universiteit Amsterdam, The Netherlands, in 2002 and 2009, respectively. From 2006 to 2007, he worked as a software engineer in an email signing and encryption company. He is currently a postdoctoral researcher at Vrije Universiteit Amsterdam. His research interests include the design and performance evaluation of high-performance, distributed systems.

Thilo Kielmann studied Computer Science at Darmstadt University of Technology, Germany. He received his Ph.D. in Computer Engineering in 1997, and his habilitation in Computer Science in 2001, both from Siegen University, Germany. Since 1998, he is working at Vrije Universiteit Amsterdam, The Netherlands, where he is currently Associate Professor at the Computer Science Department. His research interests are in the area of high-performance, distributed computing, namely programming environments and runtime systems for grid and cloud applications. He is a steering group member of the Open Grid Forum and of Gridforum Nederland.