HAL
open science

Scalable and Reliable Data Broadcast with Kascade

Stéphane Martin, Tomasz Buchert, Pierric Willemet, Olivier Richard, Emmanuel Jeanvoine, Lucas Nussbaum

To cite this version:

Stéphane Martin, Tomasz Buchert, Pierric Willemet, Olivier Richard, Emmanuel Jeanvoine, et al.. Scalable and Reliable Data Broadcast with Kascade. HPDIC - International Workshop on High Performance Data Intensive Computing, in conjunction with IEEE IPDPS 2014, May 2014, Phoenix, United States. hal-00957671v3

HAL Id: hal-00957671

https://inria.hal.science/hal-00957671v3

Submitted on 23 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Author manuscript, published in "HPDIC - International Workshop on High Performance Data Intensive Computing, in conjunction with IEEE IPDPS 2014 (2014)"

hal-00957671, version 2 - 16 Jun 2014

Scalable and Reliable Data Broadcast with Kascade

Stéphane Martin*, Tomasz Buchert*, Pierric Willemet*, Olivier Richard†, Emmanuel Jeanvoine*, Lucas Nussbaum*
* Inria, Villers-lès-Nancy, F-54600, France
Université de Lorraine, LORIA, F-54500, France
CNRS, LORIA - UMR 7503, F-54500, France
† Université de Grenoble, LIG, F-38000, France

***Abstract*—Many large scale scientific computations or Big Data analysis require the distribution of large amounts of data to each machine involved. That distribution of data often has a key role in the overall performance of the operation. In this paper, we present Kascade, a solution for the broadcast of data to a large set of compute nodes. We evaluate Kascade using a set of large scale experiments in a variety of experimental settings, and show that Kascade: (1) achieves very high scalability by organizing nodes in a pipeline; (2) can almost saturate a 1 Gbit/s network, even at large scale; (3) handles failures of nodes during the transfer gracefully thanks to a fault-tolerant design.**

***Keywords*-data broadcast; multicast; large scale; fault-tolerance**

I. Introduction

Over the recent years, many areas of scientific research and industry shifted to a data-driven model, which paves the way to many ground-breaking changes in today's society, science and engineering. However, this important paradigm evolution changes our vision of our computing infrastructure: due to the exponential growth of enormous amounts of information, the storage and the management of data emerged as the new bottleneck for many applications.

Several very different ways to organize data have been designed and used over the years. Traditional storage servers with large RAID array of disks continue to be used, but are often aggregated into storage clusters as part of a distributed file system such as Lustre [1], [2], or more recently GlusterFS [3] or Ceph [4]. Those file systems provide a POSIX interface, or at least an interface that is very similar to POSIX. Switching from POSIX-compliant semantics to other logical organizations of data enables higher performance, scalability and fault-tolerance, as demonstrated by NoSQL databases such as Apache Cassandra [5] and MongoDB [6], or the MapReduce [7] programming model.

A common feature of those new solutions is that they move away from centralized and expensive storage servers to more commodity hardware. Similarly to the move from large supercomputers to clusters of inexpensive workstations [8], recent solutions leverage cheaper and off-the-shelf hardware found in standard machines to achieve higher scalability, addressing the necessary fault-tolerance concerns.

In this paper, we address the case of broadcasting a large amount of data from one storage system to a large number of nodes. This time-critical operation is typically used as the first operation of distributed data analysis, in order to distribute the data being analyzed to each node involved so that it can then be accessed from local storage. A similar operation is also required in other contexts, such as the efficient broadcast of system images in Clouds or HPC clusters – a use case that was our original motivation as part of our work on Kadeploy [9].

The main advantage of this approach is that it limits the burden on the source storage server – as we will show later, broadcasting to hundreds of nodes can be as easy as sending the data to one node, if done properly. But this approach has two main intrinsic limitations. First, it is only applicable to cases where all nodes require the same data, or at least enough identical data to justify sending everything to all nodes, and letting each node filter the data to store only the pieces required locally. Second, the local storage capacity on each node needs to be large enough to contain all the required data. We will show later that, while those limitations prevent from using this approach in some cases, the remaining cases can benefit from great performance.

In this paper, we present Kascade, a solution for the broadcast of data to a possibly large set of compute nodes. Kascade organizes nodes in a pipeline to achieve high scalability, and includes some fault-tolerance mechanisms to handle the failure of nodes during the transfer.

The remainder of this paper is organized as follows. Section II outlines the challenges presented by data broadcasting (Section II-A), and describes the related works (Section II-B). Kascade is then presented in Section III, and is evaluated in Section IV. Finally, we conclude this paper in Section V.

II. Context

In this section, we describe the main challenges encountered when broadcasting data, and then present the existing solutions that aim at addressing this problem.

A. Challenges

Data broadcasting needs to overcome several challenges.

1) Local storage performance: First, the local storage on nodes (hard disk drives or SSD drives) is a performance bottleneck: a 7200 RPM SATA drive typically provides a raw write throughput of 100 MB/s, lower than the Gigabit Ethernet bandwidth, and the fastest SSD drives provide from 500 MB/s to 600 MB/s, still much lower than the 10 Gbit/s Ethernet bandwidth. Additionally, when files are written to a file system (and especially small files), performance is even lower. Therefore, one key requirement is that receivers start

hal-00957671, version 2 - 16 Jun 2014

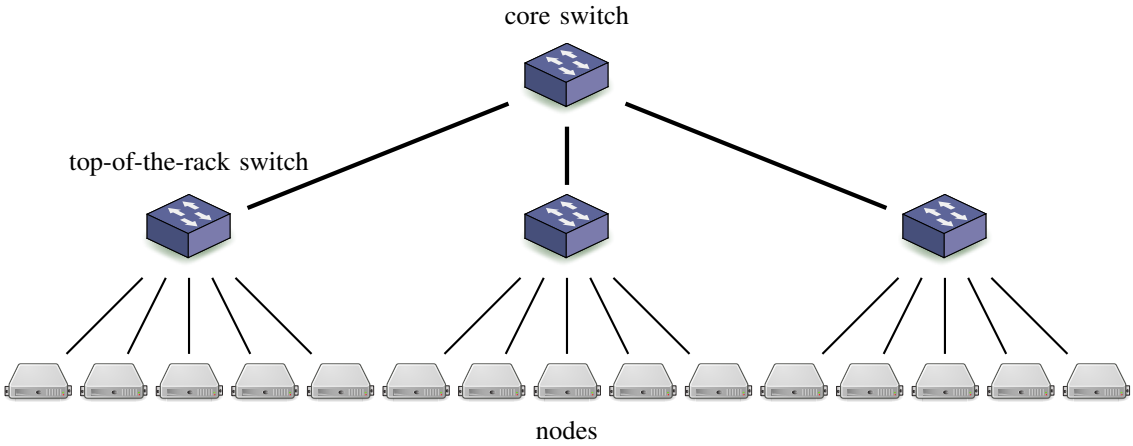

Fig. 1. Fat tree network: links between the core switch and the top-of-the-rack switches are *fatter* (higher available bandwidth) than the ones between the top-of-the-rack switches and the nodes.

writing data as soon as possible (that is, as soon as data starts arriving to nodes), rather than waiting for the full data to be received. Moreover, special care should be taken to write data in an optimal way – avoiding disk seeks, for example. Our tests in the context of our work on Kadeploy [9] showed that writing a whole file system dump with `e2image` was much faster than writing the equivalent file system content using `tar` thanks to the use of sequential writes avoiding disk seeks, instead of numerous file operations (open, close, permissions change, etc.).

2) Efficient use of fat tree networks: Most clusters networks are hierarchical, fat tree networks [10] such as the one pictured in Figure 1. It is very common for such networks that core links (the ones interconnecting top-of-the-rack switches to the main networking equipment) suffer from under-capacity when all the nodes belonging to the same switch communicate with nodes on other switches. As an example, for the experiments described in Section IV, each top-of-the-rack switch is connected to between 30 and 35 nodes using a 1 Gbit Ethernet link, but is only connected via one 10 Gbit link to core switch. One could argue that this is bad infrastructure design. Actually, economical concerns often prevent from purchasing large centralized and non-blocking Ethernet switches. Data broadcast solutions need to be aware of this topology, and preserve core links by taking into account nodes *locality*.

3) Fault-tolerance: Failures are an important problem in the context of large-scale infrastructures. When dealing with data broadcast, the importance of this problem is even higher, as (1) this operation relies heavily on hard disk drives, which, as mechanical parts, are particularly prone to failures [11], [12]; (2) this operation is often executed early in the execution of a task, possibly after the nodes were turned off or idle, so it is more likely to encounter malfunctioning nodes than an operation executed at the end of tasks. Data broadcast solutions need to be able to properly detect and handle the failure of a node.

B. Related works

Broadcasting has been the subject of a lot of attention both from a theoretical and from a more practical point of view.

From a theoretical point of view, several algorithms based on binomial trees have been designed [13], [14], which have optimal or near-optimal performance on contention-free networks. However, this is not the case of Ethernet networks, even with a single switch, as the backplane switching capacity is often a bottleneck. Pipelined broadcasts algorithms are more interesting in such networks, but topology-unaware pipelined broadcast performs poorly, as shown in [15], which evaluates pipelined broadcast algorithms in MPI implementations.

High performance network fabrics also provide special operations that can be leveraged to achieve efficient broadcast. In [16] and [17], implementations of the `MPI_Bcast` operation on top of InfiniBand hardware multicast support are proposed.

IP multicast can also be used to achieve reliable broadcast. UDPCast [18], used by SystemImager and other disk cloning solutions, is the most popular implementation of this idea. However, the use of multicast has a number of drawbacks:

- Multicast support is usually disabled by default in network switches, requiring special configuration that might not be possible in hosted environments (e.g., cloud computing);
- A central allocation of the multicast groups is required to avoid conflicts between applications – or sub-optimal solutions such as randomized multicast addresses or per-application packet tagging could be used;
- The use of high-throughput UDP broadcast is unfair to other protocols;
- The one-way communication raises a number of challenges. UDPCast proposes two modes of operation: an unidirectional mode (without return channel), and a bidirectional mode (with return channel). The unidirectional mode relies on FEC (Forward Error Correction) packets to work-around congestion, but still requires a lot of tuning (sending throughput and amount of additional FEC

Simple file broadcast to three hosts:

```
kascade -N n2,n3,n4 -i myfile.tgz -o /home/login/myfile.tgz
```

Copy a tar archive and decompress on-the-fly:

```
kascade -N n2,n3,n4 -i myfile.tgz -O 'tar -xzC /opt/'
```

Cloning a disk partition using `dd`*, streaming Kascade's standard input:*

```
dd if=/dev/sda2 | gzip | kascade -N n2,n3,n4 -O 'gunzip | dd of=/dev/sda2'
```

Fig. 2. Example uses of Kascade.

packets to send). During our experiments, we were unable to get it to work reliably. Also, in that mode the sender is not able to know if the receivers have correctly received the data. In the bidirectional mode, receivers re-request lost packets at the end of the transfer, which can lead to "ACK-implosion" at large scale as pointed in [16].

P2P-based approaches are also appealing. In [19], a BitTorrent-based broadcast is compared to a MPI-based implementation. Authors conclude that BitTorrent performs better in heterogeneous networks (networks with bottleneck links). However, in their experiments, BitTorrent only achieves a maximum throughput of about 12 MB/s, which is very disappointing as the bottleneck link in the experiment was a 1 Gbit/s link. Our own experiments with BitTorrent [20] showed that its verbose protocol and its complex mechanisms (such as *tit-for-tat*) incur a strong performance penalty on high-performance networks.

Finally, Ka [21] (which is an ancestor of Kascade, as it was also developed in the context of the Kadeploy project), Dolly [22] and Dolly+ [23] are pipelined broadcast solutions that are similar to what we will present in this paper. However, (1) Dolly and Dolly+ were not evaluated at large scale (at most ten nodes); (2) we tried to evaluate Dolly+ by ourselves, but its compilation failed, and its authors were not able to provide a solution; (3) Dolly and Ka do not provide any fault-tolerance mechanism, and whether Dolly+ provides one is unclear: none is mentioned [23], but one is mentioned in a presentation[1] – however, the corresponding code cannot be found in the archive available on Dolly+'s website[2], so the presentation might describe planned future work at the time.

III. Pipelined broadcast with Kascade

In this section, we present Kascade, a pipelined and fault tolerant file or stream broadcast tool. Kascade leverages standard network technologies (TCP/IP), is written in Ruby, and provides a friendly command-line interface. Figure 2 presents some Kascade use cases.

In the following, we describe how Kascade organizes nodes in a topology-aware pipeline to achieve performance and scalability (III-A), and how it leverages efficient external tools to control destination nodes and initiate the transfer (III-B). Then, we describe Kascade's protocol (III-C) and its support to handle node failures.

A. Topology-aware pipeline

In order to transfer data from the origin node to all the destination nodes, Kascade builds a pipeline: each node receives data from the previous one, and forwards it to the next one.

As described in Section II-A2, the pipeline must be built according to the underlying network topology to avoid saturation of some network links (e.g., inter-switch links).

Kascade assumes that the logical ordering of nodes (the numbers in their host names) matches the underlying physical topology (that is, that nodes 1 to 30 are on the first switch, that nodes 31 to 60 are on the second switch, etc.). Based on that assumption, Kascade sorts the nodes according to their numbers to achieve maximum performance. If needed, it is also possible to specify a custom sort order (if the nodes numbers do not reflect the underlying topology).

As shown in Figure 3, each link is only used once in each direction, avoiding network congestion since network links support nowadays full-duplex communications.

A final connection is performed at the end of the transfer by the last node to the first node in order to send list of failed nodes (if any).

B. Efficient start up and control of destination nodes

One crucial operation to ensure scalability is the start up of Kascade on every node. For that operation, Kascade relies on external tools such as TakTuk [24] (by default) or ClusterShell [25]. A fallback mode using pure SSH is also available.

To initiate the transfer, Kascade first copies itself and the list of nodes to all the target nodes. This copy is performed using TakTuk or ClusterShell, as they are efficient-enough to copy small files – we will show in Section IV that they are not for larger files.

Then, Kascade starts itself on each receiving node. Taktuk provides two ways to connect to nodes: an adaptive tree, where nodes already reached are used to connect to additional

[1] http://corvus.kek.jp/~manabe/pcf/dolly/dolly.ppt
[2] http://corvus.kek.jp/~manabe/pcf/dolly/index.htm

hal-00957671, version 2 - 16 Jun 2014

hal-00957671, version 2 - 16 Jun 2014

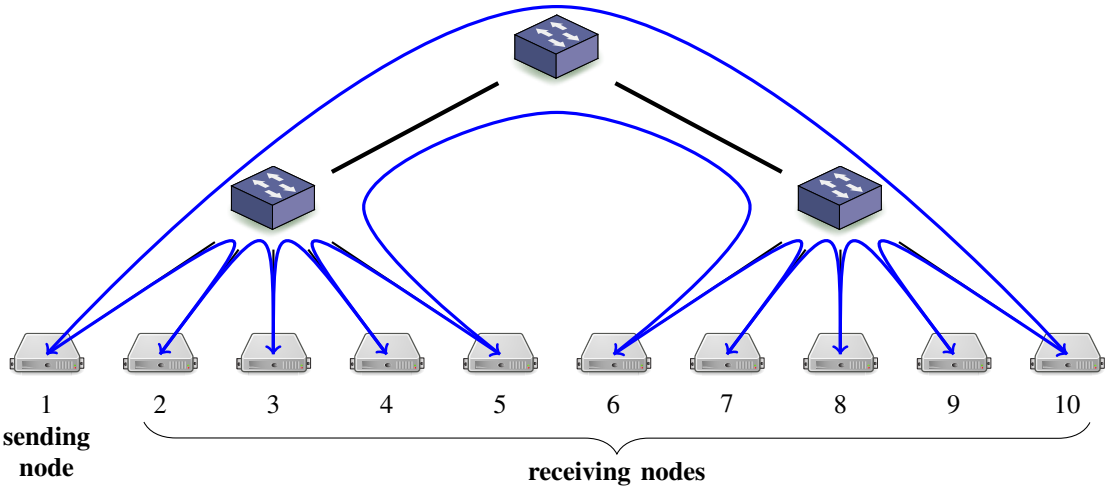

Fig. 3. Topology-aware pipeline built by Kascade: node 1 (sending node) connects to node 2, which connects to node 3, etc. Node 10 connects back to node 1, which is used to forward the final report.

GET(o)	Request stream data from offset o
PGET(o,t)	Request stream between offset o and offset t (in bytes)
FORGET(o)	Answer to a GET or PGET request when the asked part is not available anymore (case of recycled buffer). o indicates the minimal offset
DATA(s)	Answer to a GET or PGET request, followed by s bytes of data
END	Signal the end of stream
QUIT	Signal the anticipated end of stream (case of user interruption for example)
REPORT(s)	After END or QUIT, a report is sent. The report's length is s bytes
PASSED	The receiver sends to the previous node an acknowledgment to signal that the report has been sent to the first node

Fig. 4. Kascade protocol messages

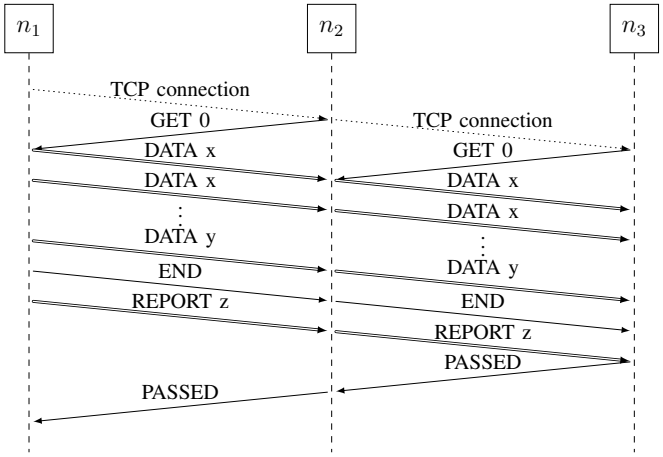

Fig. 5. Message sequence chart of communication between three nodes

nodes, and a windowed mode where the root node connects to each other node. The adaptive tree distribution is much more efficient [24], but is not able to handle failures of nodes in the middle of the tree. Therefore, Kascade uses the windowed mode by default.

ClusterShell provides a similar windowed mode, and a tree-based mode is planned to be included in the future [26].

C. Protocol

TCP/IP provides applications with reliable and ordered data transfer between two hosts. On top of TCP/IP, Kascade builds a pipeline overlay to perform efficient data transfer among a large number of nodes.

Here is a naive version of a fault-tolerant pipelined broadcast protocol. The idea is that a node n_i in the pipeline forwards data to its neighbor n_{i+1} until a failure is detected. In this case, n_i connects to n_{i+2} and recovers the transfer. This is pretty easy to understand but this does not work.

Actually, there are two issues:

1) the global size of the data is required from the beginning in order to know if the transfer is finished or if the previous node has just crashed. Being able to know at the beginning the amount of data is not an acceptable constraint since we want to support the read from the standard input stream because this is very useful in many situations;
2) there is no way to inform all nodes that the user has requested the end of the transfer. Indeed, in the naive protocol, each node has to wait for the entirety of data. When the connection is closed, they can not differentiate between a failure and a user interruption.

Thus, we propose to enhance this naive protocol by adding control instructions. Basically, this consists in defining several messages that correspond to the transitions in the protocol.

hal-00957671, version 2 - 16 Jun 2014

The complete list is presented in Figure 4.

Instead of sending raw data as stated in the naive protocol, Kascade splits the stream into chunks. It begins to send a DATA message that contains the size s of the chunk and the chunk itself. Once s bytes are read, the node waits another message that can be either another DATA message (next chunk reception), an END message, or a QUIT message if the user has interrupted the transfer. Splitting the sent data into chunks is also useful to enable the transfer of data without knowing its size beforehand, in the case of streaming the output of another process.

After the end of the transfer the report is sent in a REPORT message. The report contains all detected node failures occurred on the previous nodes (considering the pipeline topology). Once the report has been successfully forwarded through the pipeline to the first node (i.e. the sender), the node informs the previous node that the report is sent with PASSED message. After sending a such message, a node can quit.

A message sequence chart of the transfer between three nodes without error is shown in Figure 5.

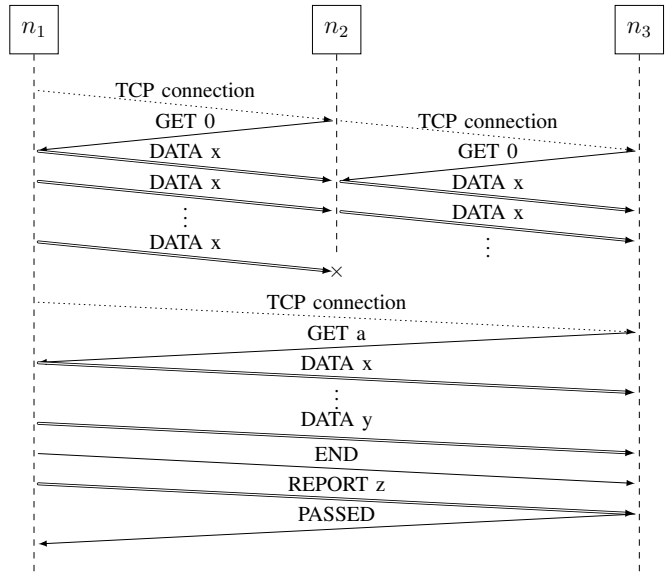

Fig. 6. Message sequence chart of communication between three nodes when error occurs

D. Fault tolerance

1) Fault detection: Kascade uses two mechanisms to detect failures. The first one is the *syscall* error catching on `read()` or `write()` functions. The second one is the use of timers on critical operations. Indeed, a timeout is triggered when the next node stops to read the stream. Such situation can happened in several cases: the next node has crashed, a subsequent node has crashed causing the next node to hang until that crash is detected, the network is congested, etc. To differentiate these cases, Kascade connects to its first alive neighbor and sends a *ping* message. If the neighbor answers quickly, the sender assumes the receiver is still alive, and the sender waits again that the write finishes. If the node does not respond, it is considered as dead and the sender connects to its next neighbor.

2) Recovery: Upon failure of n_i, the sender looks for the next available node n_j (in case of multiple adjacent failures n_j is not n_{i+1}). It is possible that the data sent already by n_i has not yet been sent from n_{j-1} to n_j. In this case, n_i needs to resend these missing parts to n_j. To this end, a Kascade node keeps some data chunks in memory, so it is able to re-send them if needed.

When n_{i-1} connects to the host n_j, n_j answers with GET(offset) and n_{i-1} transmits the required data from this offset. Because n_{i-1} is probably in advance with respect to the amount of data already sent, it needs to pick data in memory.

If too much data is lost, for example in the case of too many simultaneous node failures, and if the first node reads a file, n_j sends a PGET message to the first node in order to get the missing data. If the first node reads from a stream, its buffer probably does not contain relevant data anymore. Thus the transfer fails on n_j and on all following nodes in the pipeline. In this case the first node sends a FORGET message, n_j and all nodes after send a QUIT message to their next neighbor and quit. Then n_{i-1} continues to receive remaining data.

Sending a GET message with an offset on every connection is important because if it is sent only after the first connection and if there is a node crash between n_i and n_{j-1} that crashes before having performed its first connection, that leads to a deadlock. Indeed, n_{i-1} waits for a GET message whereas n_j waits for a first DATA message.

After such failure handling, the protocol continues such as described in previous section without nodes n_i to n_{j-1}. The Figure 6 shows this mechanism with one node failure.

IV. Experimental validation

In the following section we present an extensive evaluation of our approach to data distribution and its comparison with other existing methods. In particular, we seek to answer the following questions:

- How do the various solutions perform and scale up to large number of nodes? (IV-A)
- How does Kascade perform on high-performance networks (10 Gbit/s Ethernet, IP over InfiniBand)? (IV-B)
- What is the impact of network topology and communication structure on performance? (IV-C)
- What it the impact of I/O performance on the overall performance? (IV-D)
- How does Kascade perform on large-scale (Internet-like) setups? (IV-E)
- How does Kascade perform on smaller files? (IV-F)
- How well does Kascade's fault tolerance mechanism perform? (IV-G)

To answer these questions, we evaluated different methods for data distribution in computer networks. They are described in the following paragraphs.

hal-00957671, version 2 - 16 Jun 2014

Kascade is our approach to data distribution and is described in Section III. The version used in our experiments is 0.1.5.

TakTuk is a tool for large-scale remote execution and file distribution using an efficient tree-like topology constructed over the set of nodes (the arity of the tree is configurable). We evaluate two different TakTuk overlays: a tree of arity 1 (i.e., a tree that degrades into a chain, shown as *TakTuk/chain*) and one that is a tree of arity 2 (*TakTuk/tree*). The version used is 3.7.4.

UDPCast is a file transfer tool that can send data simultaneously to many destinations on a LAN using IP multicast. It features various methods and enables a fine-grained tuning of parameters. In our evaluation, we use a default mode that uses a feedback to coordinate data transmission. UDPCast features another mode that allows a fully uncoordinated transfer, but we found it difficult to properly tune and very unreliable. The version used is 20120424.

MPI Broadcast is a home-made implementation of data distribution method that uses MPI runtime and MPI primitives (mostly `MPI_Bcast`) for distribution. The algorithm uses a 1 MB size buffer to send consecutive fragments of a file to participating nodes. The advantage of using MPI is that the implementation is portable to different network technologies, as long as MPI runtime supports them. In fact, in our experiments two ways of execution are considered: with Ethernet (*MPI/Eth*) and with InfiniBand (*MPI/IB*). The implementation of MPI used to conduct the experiments is Open MPI (version 1.4.5).

The operating system is Debian/Linux 7 with kernel version 3.2.0. The experiments were run on different clusters of the Grid'5000 infrastructure [27].

All figures show average, normalized bandwidth as a function of client number (the node initiating the transfer is not included in that number), excepted the last one (Section IV-G) dedicated to the evaluation of fault-tolerance which shows the average normalized bandwidth under several failure conditions. The bandwidth is computed as a size of a file being transmitted divided by the time required to finish transmission. The results are presented with their respective 95 % confidence intervals according to the Student's t-distribution.

Our experiments were conducted using XPFLOW experiment workflow engine [28], [29] that allows to specify experiments in terms of business workflows, featuring scalability and robustness of execution. The raw results, XPFLOW scripts and XPFLOW distribution are available at http://www.loria.fr/~buchert/kascade2014.tar.xz.

A. Raw performance and scalability

To measure the raw performance of the methods, the experiment with a varying number of nodes was performed. A single run consisted in distributing a single 2 GB file to all nodes. The source file was stored on a RAM-backed file system and was sent to `/dev/null` on receiving nodes. Therefore the impact of storage speed is not taken into account, making the network bandwidth and latency the most important factors.

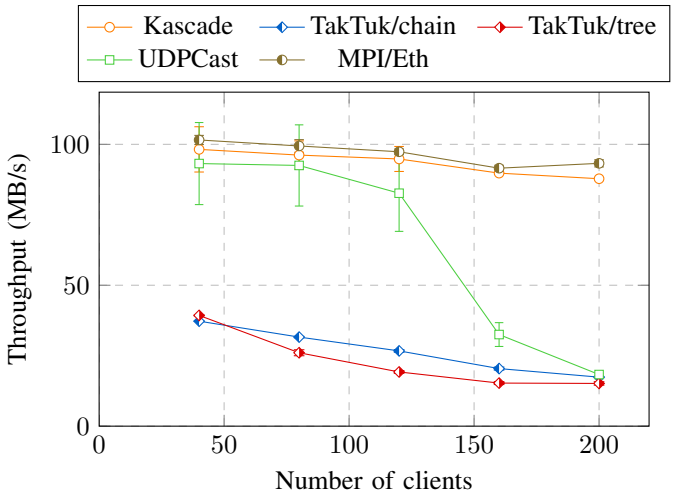

Fig. 7. Performance and scalability using up to 200 nodes on 1 Gbit/s Ethernet. Only Kascade and MPI Broadcast fully saturate the available bandwidth and scale very well with number of nodes.

The network is Ethernet with 1 Gbit/s links. The nodes are ordered in an optimal way (as was described in Section III-A). The results are presented in Figure 7.

Only Kascade and MPI Broadcast were able to nearly saturate the network links. Moreover, they scale very well with an increasing number of nodes, showing a negligible loss of performance. UDPCast shows a similar behavior for small cases, but its performance degrades rapidly when number of nodes is bigger than 100. It is due to the costly synchronization between the sender and its clients. Both variations of TakTuk perform equally bad showing very low bandwidth utilization (more than one third of the theoretical value).

To sum up, Kascade achieves almost a maximum utilization of a 1 Gbit/s network and scales very well with number of participating nodes.

B. Performance in high-performance networks

The second experiment explores the performance on a cluster of 14 nodes interconnected with 10 Gbit/s Ethernet network. The transmitted file has 5 GB. The results are presented in Figure 8.

In this scenario, none of the evaluated methods is able to saturate the link. Among them, the best one is MPI Broadcast that peaked at approximately 5 Gbit/s (i.e., 50 % of the available bandwidth), but usually stays around 3 Gbit/s. It is followed by UDPCast that is able to reach more than 3 Gbit/s, but usually rests slightly above 2 Gbit/s. Kascade shows more stable behavior with transfer bandwidth slightly above 2 Gbit/s. In contrast, TakTuk-based methods show particularly low performance.

By monitoring CPU usage, it was noticed that it is saturated during transmission with MPI Broadcast and Kascade. The bottleneck is the memory that cannot provide 10 Gbit/s throughput, at least with the current implementations. The problem can be arguably mitigated by avoiding extraneous memory accesses and using multiple threads.

hal-00957671, version 2 - 16 Jun 2014

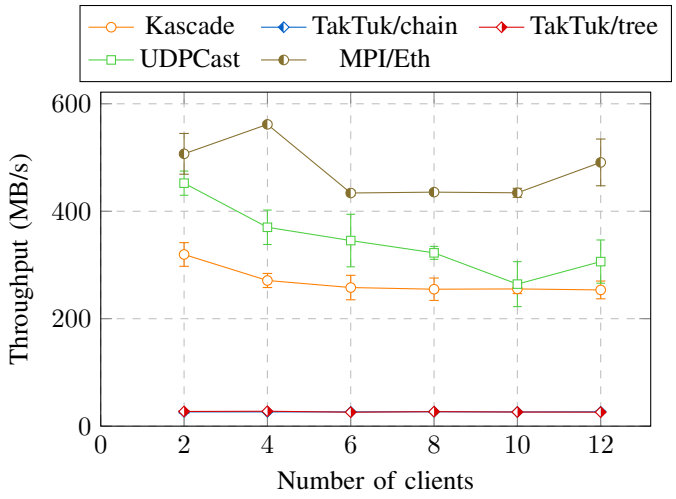

Fig. 8. The performance of the methods using 10 Gbit/s Ethernet network. No method is able to saturate the available bandwidth.

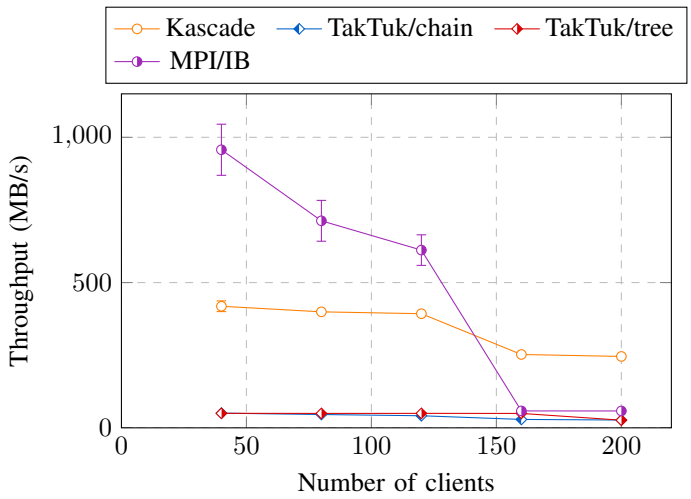

Fig. 9. The performance of the methods using IP over InfiniBand (20 Gbit/s), however *MPI/IB* is using InfiniBand directly. No method is able to saturate the bandwidth, but among them only Kascade shows scalable behavior.

The second experiment measures the performance using IP-over-InfiniBand interconnect (the transmitted file has 5 GB in this case). They are presented in Figure 9.

MPI Broadcast (note that it is used over InfiniBand provided by 2 switches) is very efficient for a small number of nodes, but does not scale very well and with 160 nodes shows a very low performance similar to TakTuk. The reason for that is that with 120 nodes and less, only a single switch is used, but for 160 nodes and more 2 switches must be used, causing the inter-switch link to be saturated. Kascade, although having more modest performance for small number of nodes, is fairly scalable and shows a behavior similar to the experiments with 10 Gbit/s Ethernet network.

To conclude, Kascade does not saturate available bandwidth in two exemplary high-speed networks, but shows much better scalability than other methods. Moreover, inability to saturate the network is likely to disappear with more tuned implementation.

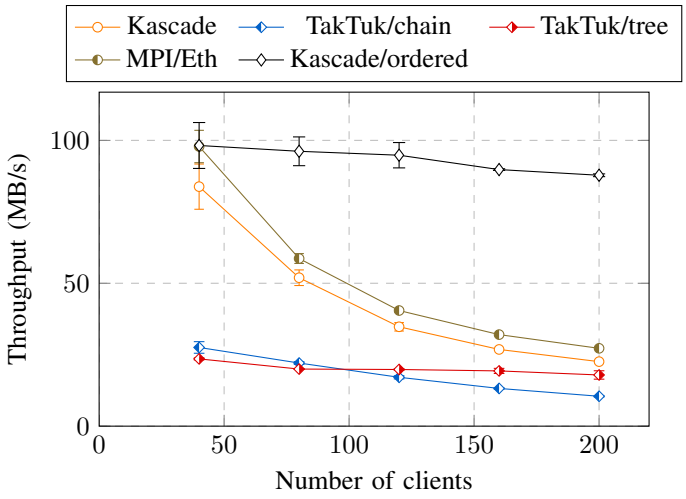

Fig. 10. The performance of the methods with random ordering of nodes. Kascade relies on a proper ordering of nodes and just like other methods shows low performance in this artificial scenario.

C. Impact of topology and structure of communication on performance

As discussed in Section III-A, some methods rely on a proper ordering of nodes so that they can profit from the underlying topology. The only difference from the experiment in Section IV-A is that the order of nodes is randomized (however the nodes remain in a single L2 network). We also include a plot of Kascade results obtained with optimal ordering of nodes (*Kascade/ordered*) as was presented in Figure IV-A. The results are presented in Figure 10.

Unsurprisingly, the performance of Kascade deteriorates when the ordering is randomized. This phenomenon, observed also for MPI Broadcast distribution, is due to the fact that the Kascade transmission chain passes switches multiple times and saturates them. This does not pose a problem in our case, since the topology of the network is generally known in practice, especially in systems such as clusters, grids and high-speed networks that are the main target of our work.

D. Impact of hard disk I/O operations on performance

In all previous experiments the hard disk was neither read from nor written to. In practice, however, data is written to permanent storage. Due to disparate bandwidths of computer memory, network and storage, this may have a large effect on the performance of data distribution.

In this experiment, a 2 GB file is distributed just like it was described in Section IV-A. This time, however, the data is written to a hard disk instead of being discarded. The fact of data reaching the disk is not concerned - its presence in the file system cache is enough.

The nodes involved in the experiment belong to the same cluster and are equipped with identical Hitachi Deskstar 7K1000.C (HDS721032CLA362) 320 GB hard disks with 16 MB buffer size and SATA II interface. A simple test

hal-00957671, version 2 - 16 Jun 2014

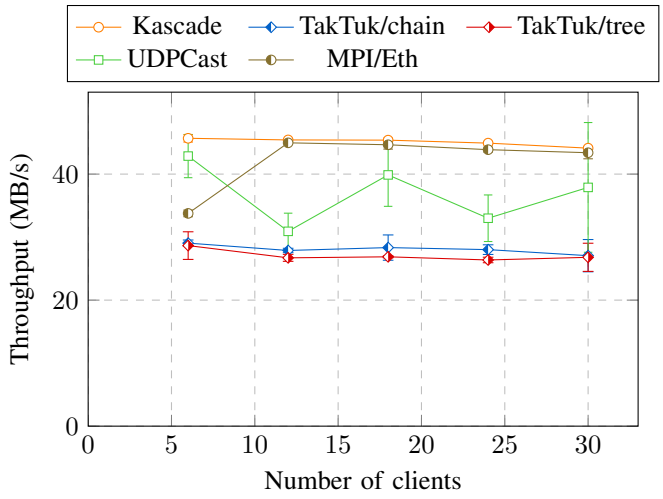

Fig. 11. The performance of the methods using 1 Gbit/s Ethernet network if the clients store the file on a hard disk. Kascade shows the best performance among the methods.

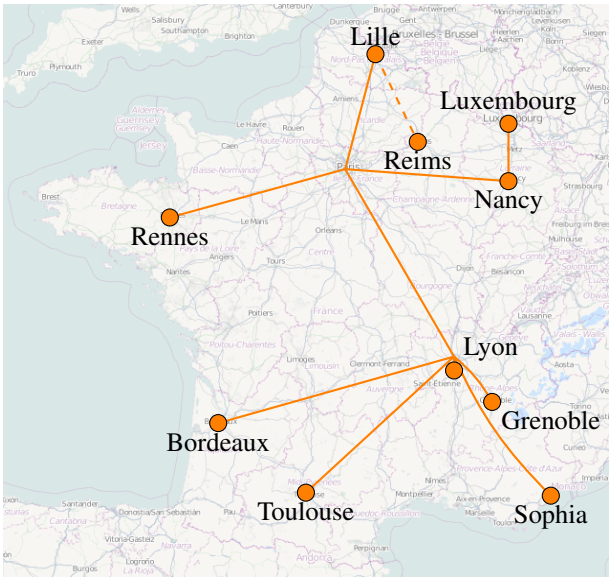

Fig. 12. The illustration of the experiment in Section IV-E. Circles represent Grid'5000 sites and lines are links between them. The order of sites used to get results presented in Figure 13 is: Lille, Grenoble, Luxembourg, Lyon, Rennes and Sophia. In particular, it means that the link between Paris and Lyon is used 5 times, but still not enough to saturate it.

using `dd` revealed that the maximal writing speed is about 83.5 MB/s. The results are presented in Figure 11.

The results show a much lower performance when the file is written to a disk. Among the methods, Kascade has the best performance by being able to write around 45 MB of data per second.

E. Performance in Internet-like, heterogeneous networks

This experiment measures the performance while using a routed, heterogeneous, long-distance network. To this end, we chose 5 geographically distant sites in Grid'5000 and

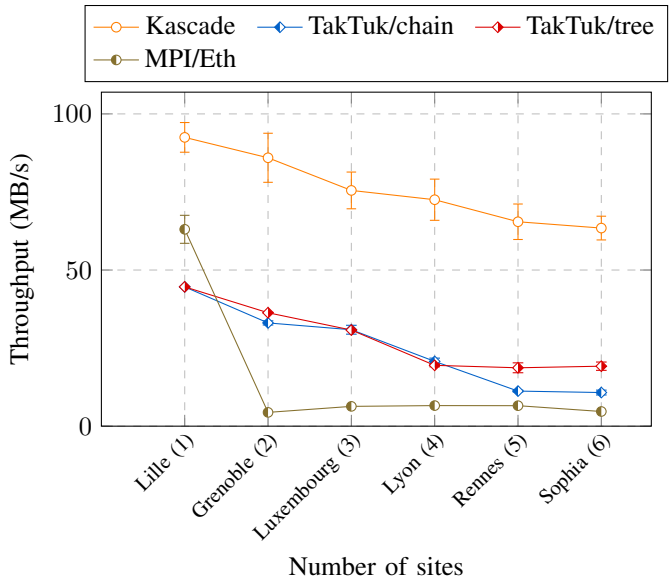

Fig. 13. The performance of the methods with multiple sites and routed IP traffic. Kascade offers the best performance among them. See Figure 12 for geographical distribution of sites.

reserved one node on each of them. Moreover, we reserved 2 more nodes on another site so that the first point in each plot represents intra-site distribution (see Figure 12). Each consecutive point, therefore, represents an inter-site link with higher latency and routed traffic. The network between sites features 10 Gbit/s bandwidth, but the latency is substantially higher (an *ICMP ping* takes about 16 ms between sites, but less than 0.2 ms within one site). As the transmission time is likely to increase, we use a 1 GB file instead. MPI Broadcast, however, showed very low performance and therefore was evaluated with a file of only 100 MB in size. The methods that cannot work with routed traffic are excluded from this experiment. The results are presented in Figure 13.

All methods tend to lose performance when using high-latency and heterogeneous links. Nevertheless, Kascade offers the best overall performance. On the other hand, MPI Broadcast suffers from network and node heterogeneity to the point that it is outperformed by TakTuk.

F. Overhead when transferring small files

To measure the overhead of protocols used by each method when transferring small files, we have run a transmission of a small file (50 MB) in a scenario similar to the one presented in Section IV-A. The obtained results are presented in Figure 14.

The results show that transmission of relatively small files gives a completely different picture than the one presented in Figure 7. The setup time takes relatively more time and methods that have efficient start-up (i.e., MPI and UDPCast) are clearly better. The results for Kascade or not surprising as it actually uses TakTuk to start itself on all involved nodes.

hal-00957671, version 2 - 16 Jun 2014

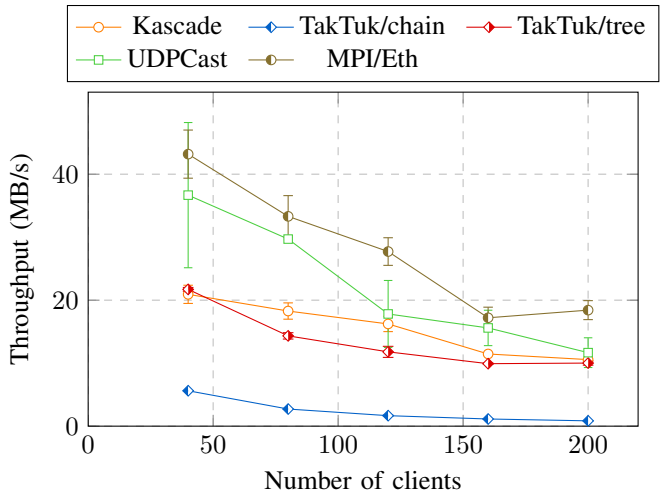

Fig. 14. Time required to distribute a small file (50 MB). All methods show similar characteristics, with MPI Broadcast outperforming the rest.

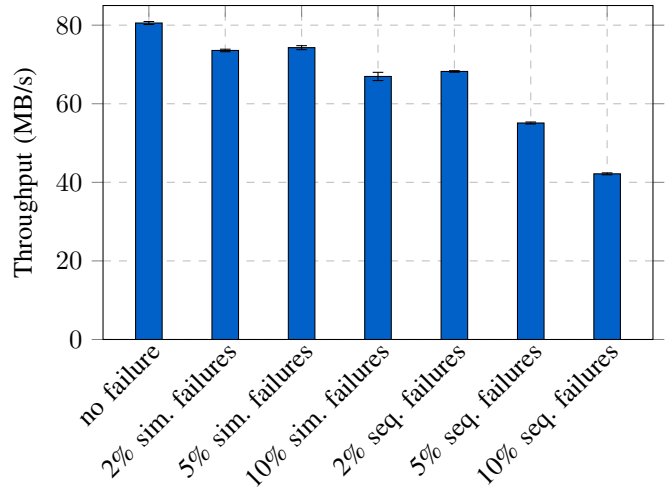

Fig. 15. Performance of Kascade in presence of failures. Three situations have been evaluated inside Distem: no failure, simultaneous failures (2%, 5%, and 10% of the nodes), sequential failures (2%, 5%, and 10% of the nodes).

G. Fault tolerance

To evaluate the fault-tolerance capabilities of Kascade described in Section III-D, we use the Distem [30] emulator. Distem is able to emulate a virtual platform over a regular cluster and is able to inject failures in the platform according to a specification provided by the user. The experimental setup is as follows:

- 20 physical nodes of a single cluster;
- each physical node has a 1 Gbit/s Ethernet network interface;
- a Distem virtual platform composed of 100 virtual nodes is launched over the physical nodes (5 virtual nodes are launched on each physical node).

In the experiment, the virtual nodes are called n_1 to n_{100}. Let us define $n_{[i-j]}$ as the virtual nodes from n_i to n_j (inclusively), and let $n_{[i,j,k,\ldots]}$ stand for the set $\{n_i, n_j, n_k, \ldots\}$.

Furthermore we define $\{\mathbf{t}, n_i\}$ an event where a failure is applied on n_i at the date **t** (e.g., t seconds after the beginning of the experiment).

The goal of the experiment is to transfer a 5 GB file from n_1 to $n_{[2-100]}$ with Kascade and to measure the average throughput.

Three kinds of scenarios are evaluated:

1) no failure;
2) failures are injected to kill 2 %, 5 %, and 10 % of the virtual nodes simultaneously, 10 seconds after the beginning of the transfer. In the three cases, the following failures have been injected:
   - 2%: $\{\mathbf{10}, n_{[29,69]}\}$
   - 5%: $\{\mathbf{10}, n_{[9,29,49,69,89]}\}$
   - 10%: $\{\mathbf{10}, n_{[9,19,29,39,49,59,69,79,89,99]}\}$
3) failures are injected to kill 2 %, 5 %, and 10 % of the virtual nodes in sequence – the failures start 10 seconds after the beginning of the transfer. In the three cases, the following failures have been injected:
   - 2%: $\{\mathbf{10}, n_{29}\}, \{\mathbf{20}, n_{69}\}$
   - 5%: $\{\mathbf{10}, n_9\}, \{\mathbf{14}, n_{29}\}, \{\mathbf{18}, n_{49}\}, \{\mathbf{22}, n_{69}\}, \{\mathbf{26}, n_{89}\}$
   - 10%: $\{\mathbf{10}, n_9\}, \{\mathbf{12}, n_{19}\}, \{\mathbf{14}, n_{29}\}, \{\mathbf{16}, n_{39}\}, \{\mathbf{18}, n_{49}\}, \{\mathbf{20}, n_{59}\}, \{\mathbf{22}, n_{69}\}, \{\mathbf{24}, n_{79}\}, \{\mathbf{26}, n_{89}\}, \{\mathbf{28}, n_{99}\}$

Every experiment is repeated 50 times, and results are averaged.

Figure 15 shows the performance of Kascade when failures are injected. First, in all the cases, the file was transferred correctly. One can notice than the performance without failure is quite far from the maximum network link capacity, 80 MB/s instead of 128 MB/s. This is expected since the node folding and the virtualization technique for running virtual nodes inside Distem induce an overhead. Thus, this value just acts as a reference value. Then we can see that simultaneous failures have a lower cost than sequential failures. This is normal since failure detection can be pipelined in case of simultaneous failures. Indeed, failure detection uses timeouts and every time a timeout is reached, one second is lost. As a consequence, increasing the number of sequential failures has a bigger impact on performance than simultaneous failures.

The goal of this experiment was to ensure that Kascade works despite failures and to get an idea of its the performance. Kascade has a high tuning potential and could be tuned according to the network used in order to reduce timeouts and achieve better performance even in case of sequential failures.

V. Conclusions and Future Work

In this paper, we presented Kascade, a solution for the large scale broadcast of data. Kascade achieves high performance and scales very well – saturating a 1 Gbit/s network even at large scale. Also, a fault-tolerant mechanism enables Kascade to handle node failures during the transfer.

Kascade provides acceptable performance on high performance networks (10 Gbit/s Ethernet, 20 Gbit/s InfiniBand), but it could be further improved, as shown in our extensive performance evaluation. It also performs well in Internet-like

hal-00957671, version 2 - 16 Jun 2014

settings. Among the evaluated solutions, it is the only one that performs adequately in all situations.

Kascade does not currently defend very well against one specific scenario: the case where the network or disk performance of one specific node is slowing down the whole process. Kascade could be further improved to detect malfunctioning nodes (by measuring their performance during the transfer) and exclude them from the transfer if their performance is lower than a specific threshold.

Kascade is a standalone tool, released under the CeCILL v2 Free Software license. It can be found in the `addons/kascade` directory of the Kadeploy3 source (http://kadeploy3.gforge.inria.fr/).

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 2003.

[2] P. J. Braam *et al.*, "The lustre storage architecture." [Online]. Available: http://wiki.lustre.org

[3] "Glusterfs." [Online]. Available: http://www.gluster.org/

[4] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[5] "Apache cassandra." [Online]. Available: http://cassandra.apache.org/

[6] "mongodb." [Online]. Available: http://www.mongodb.org/

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings, International Conference on Parallel Processing*, vol. 95, 1995.

[9] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and Scalable Operating System Provisioning," *USENIX ;login:*, vol. 38, no. 1, pp. 38–44, Feb. 2013.

[10] C. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, 1985.

[11] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?" in *FAST*, vol. 7, 2007, pp. 1–16.

[12] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." in *FAST*, vol. 7, 2007, pp. 17–23.

[13] S. L. Johnsson and C.-T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1249–1268, 1989.

[14] R. Kesavan and D. K. Panda, "Optimal multicast with packetization and network interface support," in *Proceedings of the 1997 International Conference on Parallel Processing*, 1997, pp. 370–377.

[15] P. Patarasuk, X. Yuan, and A. Faraj, "Techniques for pipelined broadcast on ethernet switched clusters," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, pp. 809–824, 2008.

[16] J. Liu, A. R. Mamidala, and D. K. Panda, "Fast and scalable MPI-level broadcast using infiniband's hardware multicast support," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 10.

[17] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast," in *International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[18] "Udpcast." [Online]. Available: http://www.udpcast.linux.lu/

[19] K. Dichev and A. Lastovetsky, "Mpi vs. bittorrent: switching between large-message broadcast algorithms in the presence of bottleneck links," in *Euro-Par 2012: Parallel Processing Workshops*, 2013, pp. 185–195.

[20] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.

[21] P. Augerat, W. Billot, S. Derr, and C. Martin, "A scalable file distribution and operating system installation toolkit for clusters," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[22] F. Rauch, C. Kurmann, and T. M. Stricker, "Optimizing the distribution of large data sets in theory and practice," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 165–181, 2002.

[23] A. Manabe, "Disk cloning program 'dolly+'for system management of pc linux cluster," *Computing in High Energy Physics and Nuclear Physics*, 2001.

[24] B. Claudel, G. Huard, and O. Richard, "Taktuk, adaptive deployment of remote executions," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009, pp. 91–100.

[25] S. Thiell, A. Degrémont, H. Doreau, and A. Cedeyn, "Clustershell, a scalable execution framework for parallel tasks," in *Linux Symposium*, 2012, p. 77.

[26] S. Thiell, A. Degrémont, and H. Doreau, "ClusterShell, python library and tools for scalable cluster administration," in *PyHPC - Python in HPC Workshop*, 2013.

[27] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform," in *6th IEEE/ACM International Workshop on Grid Computing (Grid)*, Nov. 2005, pp. 99–106. [Online]. Available: http://hal.inria.fr/inria-00000284/en/

[28] T. Buchert and L. Nussbaum, "Leveraging business workflows in distributed systems research for the orchestration of reproducible and scalable experiments," in *9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication - MajecSTIC 2012 (2012)*, Lille, France, Aug. 2012. [Online]. Available: http://hal.inria.fr/hal-00724313

[29] T. Buchert, L. Nussbaum, and J. Gustedt, "A workflow-inspired, modular and robust approach to experiments in distributed systems," INRIA, Research Report RR-8404, Nov. 2013.

[30] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, "Design and Evaluation of a Virtual Experimental Environment for Distributed Systems," in *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Belfast, Royaume-Uni, Feb. 2013, pp. 172 – 179. [Online]. Available: http://hal.inria.fr/hal-00724308