Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach

Bogdan Nicolae, Gabriel Antoniu, Luc Bougé

Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach

Bogdan Nicolae[1], Gabriel Antoniu[2], and Luc Bougé[3]

[1] University of Rennes 1, IRISA, Rennes, France
[2] INRIA, Centre Rennes - Bretagne Atlantique, IRISA, Rennes, France
[3] ENS Cachan/Brittany, IRISA, France

Abstract. Whereas traditional Desktop Grids rely on centralized servers for data management, some recent progress has been made to enable distributed, large *input* data, using to peer-to-peer (P2P) protocols and Content Distribution Networks (CDN). We make a step further and propose a generic, yet efficient data storage which enables the use of Desktop Grids for applications with high *output* data requirements, where the access grain and the access patterns may be random. Our solution builds on a blob management service enabling a large number of concurrent clients to efficiently read/write and append huge data that are fragmented and distributed at a large scale. Scalability under heavy concurrency is achieved thanks to an original metadata scheme using a distributed segment tree built on top of a Distributed Hash Table (DHT). The proposed approach has been implemented and its benefits have successfully been demonstrated within our *BlobSeer* prototype on the Grid'5000 testbed.

1 Introduction

During the recent years, Desktop Grids have been extensively investigated as an efficient way to build cheap, large-scale virtual supercomputers by gathering idle resources from a very large number of users. Rather then relying on clusters of workstations belonging to institutions and interconnected through dedicated, high-throughput wide-area interconnect (which is the typical physical infrastructure for Grid Computing), Desktop Grids rely on desktop computers from individual users, interconnected through Internet. Volunteer computing is a form of distributed computing in which Desktop Grids are used for projects of public interest: the infrastructure is built thanks to volunteers that accept to donate their idle resources (cycles, storage) in order to contribute to some public cause. These highly-distributed infrastructures can typically benefit to embarrassingly-parallel, computationally intensive applications, where the workload corresponding to each computation can easily be divided into a very large set of small independent jobs that can be scattered across the available computing nodes. In a typical (and widely used) setting, Desktop Grids rely on a master/worker scheme: a central server (playing the master role) is in charge of distributing the small jobs to many volunteer workers that have announced their availability. Once the computations are complete, the results are typically gathered on the central (master) server, which

validates them, then builds a global solution. BOINC [1], XtremWeb [2] or Entropia [3] are examples of such Desktop Grid systems.

The initial, widely-spread usage of Desktop Grids for parallel applications consisting in non-communicating tasks with small input/output parameters is a direct consequence of the physical infrastructure (volatile nodes, low bandwidth), unsuitable for communication-intensive parallel applications with high input or output requirements. However, the increasing popularity of volunteer computing projects has progressively lead to attempts to enlarge the set of application classes that might benefit of Desktop Grid infrastructures. If we consider distributed applications where tasks need very large input data, it is no longer feasible to rely on classic centralized server-based Desktop Grid architectures, where the input data was typically embedded in the job description and sent to workers: such a strategy could lead to significant bottlenecks as the central server gets overwhelmed by download requests. To cope with such data-intensive applications, alternative approaches have been proposed, with the goal of offloading the transfer of the input data from the central servers to the other nodes participating to the system, with potentially under-used bandwidth.

Two approaches follow this idea. One of them adopts a P2P strategy, where the input data gets spread across the distributed Desktop Grid (on the same physical resources that serve as workers) [4]. A central data server is used as an initial data source, from which data is first distributed at a large scale. The workers can then download their input data from each other when needed, using for instance a BitTorrent-like mechanism. An alternative approach [4] proposes to use Content Distribution Networks (CDN) to improve the available download bandwidth by redirecting the requests for input data from the central data server to some appropriate surrogate data server, based on a global scheduling strategy able to take into account criteria such as locality or load balancing. The CDN approach is more costly then the P2P approach (as it relies on a set of data servers), however it is potentially more reliable (as the surrogate data servers are supposed to be stable enough).

In this paper, we make a step further and consider using Desktop Grids for distributed applications with high *output* data requirements. We assume that each such application consists of a set of distributed tasks that *produce and potentially modify large amounts of data* in parallel, under heavy concurrency conditions. Such characteristics are featured by 3D rendering applications, or massive data processing applications that produce data transformations. In such a context, an new approach to data management is necessary, in order to cope with both input and output data in a scalable fashion. Very recent efforts have partially addressed the above issues from the perspective of the *checkpoint* problem: the *stdchk* system [5] proposes to use the Desktop Grid infrastructure to store the (potentially large) checkpoints generated by Desktop Grid applications. This proposal is however specifically optimized for checkpointing, where large data units need to be written sequentially. It relies on a centralized metadata management scheme which becomes a potential bottleneck when data access concurrency is high. Related work has been carried out in the area of parallel and distributed file systems [6–8] and archiving systems [9]: in all these systems the metadata management is centralized. We propose a *generic*, yet *efficient* storage solution allowing Desktop Grids to be used by applications that generate large amounts of data (not only for checkpointing!), with

random access patterns, variable access grains, under potentially heavy concurrency. Our solution is based on BlobSeer, a blob management service we developed in order to address the issues mentioned above.

The main contribution of this paper is to demonstrate how the decentralized metadata scheme used in BlobSeer fits the needs of the considered application class, and to demonstrate our claims through extensive experimental evaluations. The paper is structured as follows. Section 2 gives an overview of our approach. Section 3 introduces the proposed architecture, with a focus on metadata management in Section 4. Extensive experimental evaluation is performed in Section 5. On-going and future work is discussed in Section 6.

2 Towards a decentralized architecture for data and metadata management in desktop grids

A sample scenario: 3D rendering High resolution 3D rendering is known to be costly. Luckily, as rendering is embarrassingly parallel, both industry [10] and the research community [11] have an active interest in building render farms based on the Desktop Grid paradigm. In this context, one of the limitations encountered regards data management: both rendering input and output data reach huge sizes and are accessed in a fine-grain manner (frame-by-frame) under high contention (task-per-frame). From a more general perspective, we aim at providing efficient data management support on Desktop Grids for workloads characterized by: *large input data*, *large output data*, *random access patterns*, *fine-grain data access for both reading and writing* and *high access concurrency*.

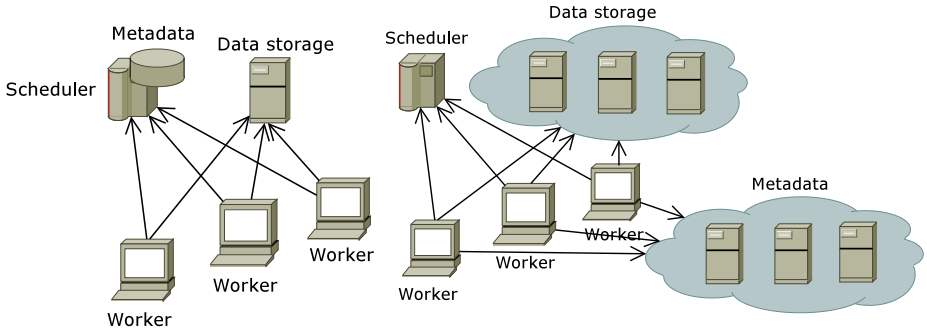
Requirements for data management Given the workload characterization given above, we can infer the following desirable properties:

Storage capacity and space efficiency. The data storage system should be able to store huge pieces of individual data blobs, for a large amount of data overall. Besides, as we assume the applications to be write-intensive, they may generate many versions of the data. Therefore, a space-efficient storage is highly desirable.

Read and write throughput under heavy concurrency. The data storage system should provide efficient and scalable support for input/output data accesses, as a large number of concurrent processes concurrently potentially read and write the data.

Efficient fine-grain, random access. As the grain and the access pattern may be random, the storage system should rely on a generic mechanism enabling efficient random fine-grain accesses within huge data blobs. We definitely favor this approach rather than relying on optimized sequential writes (as in [5]) to a large number of small files, for manageability reasons.

Versioning support. Providing versioning support is also a desirable feature, favoring concurrency, as some version of a data blob may be read while a new version is concurrently created. This fits the needs the needs of checkpointing applications [5], but is also suitable for efficient pipelining through a sequence of transformations on huge amounts of data.



(a) A classic Desktop Grid architecture (b) A decentralized Desktop Grid architecture

Fig. 1. Metadata and data management in Desktop Grids: centralized versus decentralized

Global architecture overview In a classic Desktop Grid architecture (Figure 1(a)), the workers directly interact with the central scheduler, in order to request a job. Each job is defined by its input/output data, each of which can be downloaded/uploaded from a (typically centralized) location provided by the scheduler. This scheme has several major disadvantages. First, data is stored in a centralized way: this may lead to a significant potential bottleneck for data intensive applications. Second, the whole *metadata* is managed by the central scheduler, which is put under high I/O pressure by all clients that concurrently read the metadata and may potentially become an I/O bottleneck.

Whereas several recent approaches have proposed to decentralize data management in Desktop Grids through P2P or CDN-based strategies [4], to the best of our knowledge, none has explored the idea of decentralizing metadata. We specifically take this approach and propose an architecture based on the BlobSeer prototype, which implements an efficient lock-free, distributed metadata management scheme, in order to provide support for performant concurrent write accesses to data. The Desktop Grid architecture is modified as illustrated on Figure 1(b). The scheduler keeps only a minimal input/output metadata information: the bulk of metadata is delegated to a set of distributed metadata providers. An overview of our approach is given in Section 3, with a focus on metadata management in Section 4. Note that a full description of the algorithms used is available in [12]: in this paper we briefly remind them, then we focus on the impact of distributing data and metadata.

3 The core of our approach: the BlobSeer service

BlobSeer at a glance We have experimentally studied the approach outlined above using our *BlobSeer* prototype: a blob (Binary Large OBject) management service [13, 14]. To cope with very large data blobs, BlobSeer uses striping: each blob is made up of blocks of a fixed size *psize*, referred to as *pages*. These pages are distributed among storage space providers. *Metadata* facilitates access to a range *(offset, size)* for any existing version of a blob snapshot, by associating such a range with the physical nodes where the corresponding pages are located.

BlobSeer's client applications may update blobs by writing a specific range within the blob (WRITE). Rather than updating the current pages, each such operation generates a *new* set of pages corresponding to the offset and size requested to be updated.

Clients may also APPEND new data to existing blobs, which also results in the creation of new pages. In both cases, metadata is then generated and "weaved" together with the old metadata in such way as to create the illusion of a new incremental snapshot that actually shares the unmodified pages of the blob with the older versions. Thus, two successive snapshots v and $v + 1$ physically share the pages that fall outside of the range of the update that generated snapshot $v + 1$. Metadata is also partially shared across successive versions, as further explained below. Clients may access a specific version of a given blob through the READ primitive.

Metadata is organized as a segment-tree like structure (see Section 4) and is scattered across the system using a Distributed Hash Table (DHT). Distributing data and metadata is the key choice in our design: it enables high performance through parallel, direct access I/O paths, as demonstrated in Section 5.

BlobSeer's architecture. Our storage infrastructure consists of a set of distributed processes.

Clients may CREATE blobs, then READ, WRITE and APPEND data to them. There may be multiple concurrent clients, and their number may dynamically vary in time.

Data providers physically store the pages generated by WRITE and APPEND. New data providers may dynamically join and leave the system. In a Desktop Grid setting, the same physical nodes which act as workers may also serve as data providers.

The provider manager keeps information about the available storage space and schedules the placement of newly generated pages according to a load balancing strategy.

Metadata providers physically store the metadata allowing clients to find the pages corresponding to the blob snapshot version. We use a distributed metadata management scheme to enhance data throughput through parallel I/O: this aspect is addressed in detail in Section 4. In a Desktop Grid setting, as the data providers, metadata providers can be physically mapped to the physical nodes acting as workers.

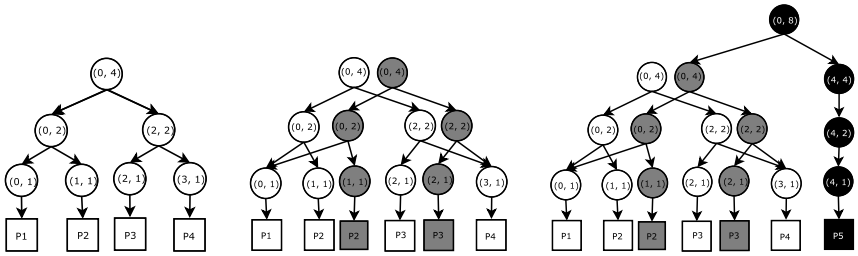
The version manager is the key actor of the system. It registers update requests (APPEND and WRITE), assigns snapshot version numbers, end eventually publishes new blob versions, while guaranteeing total ordering and atomicity. In a Desktop Grid setting, this entity can be collocated with the centralized scheduler.

Reading data To read data, clients need to provide a blob id, a specific version of that blob, and a range, specified by an offset and a size. The client first contacts the version manager. If the version has been published, the client then queries the metadata providers for the metadata indicating on which providers are stored the pages corresponding to the required blob range. Finally, the client fetches the pages in parallel from the data providers. If the range is not page-aligned, the client may request only the required part of the page from the page provider.

Writing and appending data To write data, the client first determines the number of pages that cover the range to be written. It then contacts the provider manager and requests a list of page providers able to store the pages. For each page in parallel, the client generates a globally unique page id, contacts the corresponding page provider and stores the contents of the page on it. After successful completion of this stage, the client

contacts the version manager to registers its update. The version manager assigns to this update a new snapshot version v and communicates it to the client, which then generates new metadata and "weaves" it together with the old metadata such that the new snapshot v appears as a standalone entity (details are provided in Section 4). Finally, the client notifies the version manager of success, and returns successfully to the user. At this point, the version manager is responsible for eventually publishing the version v of the blob. The APPEND operation is almost identical to the WRITE: the implicit offset is the size of the previously published snapshot version.

4 Zoom on metadata management



(a) The metadata after a write of four pages

(b) The metadata after overwriting two pages

(c) The metadata after an append of one page

Fig. 2. Metadata representation

Metadata stores information about the pages which make up a given blob, for each generated snapshot version. To efficiently build a full view of the new snapshot of the blob each time an update occurs, BlobSeer creates new metadata, rather than updating old metadata. As we will explain below, this decision significantly helps us provide support for heavy concurrency, as it favors *independent concurrent accesses to metadata without synchronization.*

The distributed metadata tree We organize metadata as a *distributed segment tree* [15]: one such tree is associated to each snapshot version of a given blob id. A segment tree is a binary tree in which each node is associated to a range of the blob, delimited by *offset* and *size*. We say that the node *covers* the range (*offset*, *size*). For each node that is not a leaf, the left child covers the first half of the range, and the right child covers the second half. Each leaf covers a single page. We assume the page size $psize$ is a power of two. Figure 2(a) depicts the structure of the metadata for a blob consisting of four pages. We assume the page size is 1. The root of the tree covers the range $(0, 4)$, while each leaf covers exactly one page in this range.

To favor efficient concurrent access to metadata, tree nodes are stored on the metadata providers in a distributed way, using a simple DHT (Distributed Hash Table). Each tree node is identified uniquely by its version and range specified by the offset and size it covers.

Sharing metadata across snapshot versions Such a metadata tree is created when the first pages of the blob are written, for the range covered by those pages. To avoid the overhead (in time and space!) of rebuilding such a tree for the subsequent updates, we create new tree nodes only for the ranges that *do* intersect with the range of the update. These new tree nodes are "weaved" with existing tree nodes generated by past updates (for ranges that do not intersect with the range of the update), in order to build a a new consistent view of the blob, corresponding to a new snapshot version. Figure 2(b), shows how metadata evolves when pages 2 and 3 of the 4-page blob represented on Figure2(a) are modified for the first time.

Expanding the metadata tree APPEND operations make the blob "grow": consequently, the metadata tree gets expanded, as illustrated on Figure 2(c), where new metadata tree nodes are generated, to take into account the creation of a fifth page by an APPEND operation.

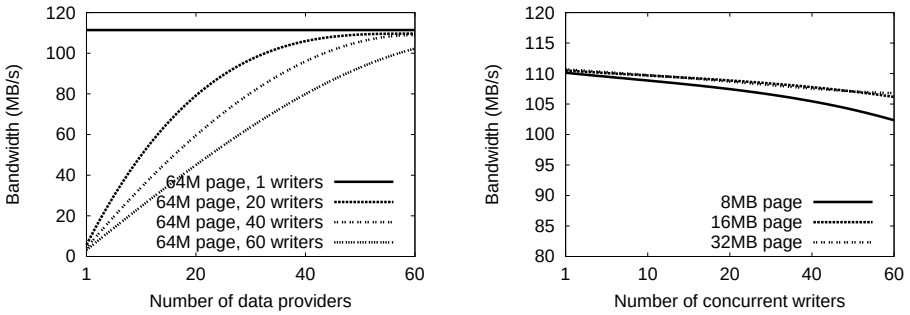
Reading metadata Metadata is accessed during a READ operation in order to find out what pages fully cover the requested range R. This involves traversing down the segment tree, starting from the root that corresponds to the requested snapshot version. To start the traversal, the client gets the root from the version manager, which is responsible to store the mapping between all snapshot versions and their corresponding roots. A node N that covers segment R_N is explored if the intersection of R_N with R is not empty. Since tree nodes are stored in a DHT, exploring a node involves fetching it from the DHT. All explored leaves reached this way provide information allowing to fetch the contents of the pages from the data providers.

Writing metadata For each update (WRITE or APPEND) producing a snapshot version *v*, it is necessary to build a new metadata tree (possibly sharing nodes with the trees corresponding to previous snapshot versions). This new tree is the smallest (possibly incomplete) binary tree such that its leaves are exactly the leaves covering the pages of range that is written. The tree is built bottom-up, from the leavers towards the root. Note that inner nodes may have children which do *not* intersect the range of the update to be processed. For any given snapshot version v, these nodes form the *set of border nodes* B_v. When building the metadata tree, the versions of these nodes are required. The version manager is responsible to compute the versions of the border nodes such as to assure proper update ordering and consistency.

5 Experimental evaluation

To illustrate the advantages of our proposed approach we have performed a set of experiments that study the impact of both data and metadata distribution on the achieved performance levels under heavy write concurrency. All experimentation has been performed using our BlobSeer prototype.

Our implementation of the metadata provider relies on a custom DHT (Distributed Hash Table) based on a simple static distribution scheme. Communication between nodes is implemented on top of an asynchronous RPC library we developed on top of the Boost C++ ASIO library [16]. The experiments have been performed on the



(a) Impact of data distribution on the average output bandwidth of workers

(b) Impact of job output size on the average output bandwidth of workers

Fig. 3. Data distribution benefits under high write concurrency

Grid'5000 [17] testbed, a reconfigurable, controllable and monitorable Grid platform gathering 9 sites in France. For each experiment, we used nodes located within a single site (Rennes or Orsay). The nodes are outfitted with x86_64 CPUs and 4 GB of RAM. Intracluster bandwidth is 1 Gbit/s (measured: 117.5MB/s for TCP sockets with MTU = 1500 B), latency is 0.1 ms.

5.1 Benefits of data decentralization

To evaluate the impact of data decentralization on the workers' write performance, we consider a set of concurrent workers that write the output data in parallel and we measure the average write bandwidth.

Impact of the number of data providers In this setting, we deploy a version manager, a provider manager and a variable number of data providers. We also deploy a fixed number of workers and synchronize them to start writing output data simultaneously. Each process is deployed on a dedicated node within the same cluster. As this experiment aims at evaluating the impact of *data* decentralization, we fix the page size at 64 MB, large enough as to generate only a minimal metadata management overhead. We assume each worker writes a single page. A single metadata provider is thus sufficient for this experiment.

Each worker generates and writes its output data 50 times. We compute the average bandwidth achieved by all workers, for all their iterations. The experiment is repeated by varying the total number of workers from 1 to 60.

Results are shown on Figure 3(a): for one single worker, using more than one data provider does not make any difference since a single data provider is contacted at the same time. However, when multiple workers concurrently write their output data, the benefits of data distribution become visible. Increasing the number of data providers leads to a dramatic increase in bandwidth performance: from a couple of MB/s to over 100 MB/s when using 60 data providers. Bandwidth performance flattens rapidly when the number of data providers is at least the number of workers. This is explained by the fact that using at least as many data providers as workers enables the provider manager to direct each concurrent write request to a distinct data provider. Under such conditions, the bandwidth measured for a single worker under no concurrency (115 MB/s)

is just by 12% higher than the average bandwidth reached when 60 workers write the output data concurrently (102 MB/s).

Impact of the page size We then evaluate the impact of the data output size on the achieved write bandwidth. As in the previous setting, we deploy a version manager, a provider manager and a metadata provider. This time we fix the number of providers to 60. We deploy a variable number of workers and synchronize them to start writing output data simultaneously. Each worker iteratively generates its job output and writes it as a single page to BlobSeer (50 iterations). The achieved bandwidth is averaged for all workers. We repeat the experiment for different sizes of the job output: 32 MB, 16 MB, 8 MB. As can be observed in Figure 3(b), a high bandwidth is sustained as long as the page is large enough. The average client bandwidth drops from 110 MB/s (for 32 MB pages) to 102 MB/s (for 8 MB pages).

5.2 Benefits of *metadata* decentralization

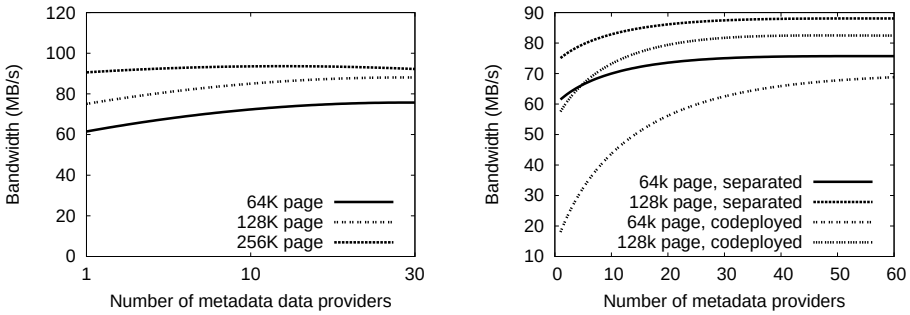
In the previous set of experiments we have intentionally minimized the impact of metadata management, in order to emphasize the impact of data distribution. As a next step, we study how metadata decentralization impacts performance. We consider a setting with a large number of workers, each of which concurrently generates many pages, so that metadata management becomes a concern.

Impact of the number of metadata providers To evaluate the impact of metadata distribution as accurately as possible, we first deploy as many data providers as workers, to avoid potential bottlenecks on the providers. Each process is deployed on a separate physical node. We deploy a version manager, a provider manager and a fixed number of 60 providers. We then launch 60 workers and synchronize them to start writing output data simultaneously. Each worker iteratively generates a fixed-sized 64 MB output and writes it to BlobSeer (50 iterations). The achieved bandwidth is averaged for all workers. We vary the number of metadata providers from 1 to 30. We repeat the whole experiment for various page sizes (64 KB, 128 KB and 256 KB).

Results on Figure 4(a) show that increasing the number of metadata providers results in an improved average bandwidth under heavy concurrency. The improvement is more significant when reducing the page size: since the amount of the associated metadata doubles when the page size halves, the I/O pressure on the metadata providers doubles too. We can thus observe that the use of a centralized metadata provider leads to a clear bottleneck (62 MB/s only), whereas using 30 metadata providers improves the write bandwidth by over 20% (75 MB/s).

Impact of the co-deployment of data and metadata providers In order to increase the scope of our evaluation without increasing the number of physical network nodes, we perform an additional experiment. We keep exactly the same setting as previously, but we co-deploy a data provider and a metadata provider on each physical node (instead of deploying them on separate nodes). We expect to measure a consistent performance drop and establish a correlation with the results of the previous experiment. This can enable us to predict the performance behavior of our system for larger physical configurations.

Results are shown in Figure 4(b), for two page sizes: 128 KB and 64 KB. For reference, the results obtained in the previous experiment for the same job outputs are



(a) Metadata and data providers deployed on distinct physical nodes

(b) Metadata providers and data providers: deployed on distinct physical nodes vs co-deployed in pairs

Fig. 4. Metadata striping benefits under high write concurrency

plotted on the same figure. We observe a 11% decrease for a 64 KB page size and 7% decrease for a 128 KB page size when using 60 metadata providers. Notice the strong impact of the co-deployment when using a single metadata provider. In this case the I/O pressure on the metadata provider adds to the already high I/O pressure on the co-deployed provider, bringing the bandwidth drop to more than 66%.

The torture test: putting the system under heavy pressure Finally, we run an additional set of experiments that evaluate the impact of metadata distribution on the total aggregated bandwidth under heavy write concurrency, when pushing the pressure on the whole system even further by significantly increasing the number of workers that write job outputs concurrently.

In this setting, we use a larger configuration: we deploy a version manager, a provider manager, 90 data providers and 90 metadata providers. The version manager and the provider manager are deployed on separate physical nodes. The data providers and metadata providers are co-deployed as in the previous experiment (one provider and one metadata provider per physical node). Finally, a variable number of additional, separate physical nodes (from 1 to 90) host 4 workers each, thus adding up a maximum of 360 workers in the largest configuration. All workers are synchronized to start writing at the same time. Each worker iteratively writes an 8 MB output consisting of 128 pages of 64 KB size. We measure the average write bandwidth per worker over 50 iterations, for all workers and then compute the overall aggregated bandwidth for the whole set of distributed workers.

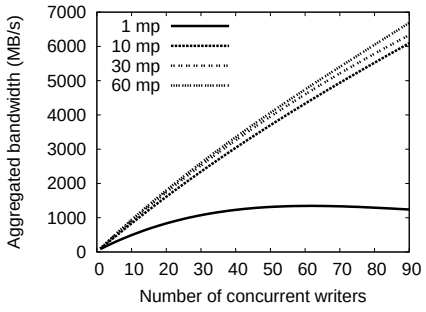


Fig. 5. Impact of the decentralized metadata management on the aggregated bandwidth.

The results are synthesized on Figure 5. First, we can clearly see that using a centralized metadata provider severely limits the overall aggregated write bandwidth under heavy concurrency, thus demonstrating the benefits of our decentralized metadata management scheme. We could thus measure up to a 6.7 GB/s aggregated bandwidth in a 60-metadata provider configuration, with 360 concurrent writers. Based on the correlation established in the previous experiment, we could estimate that a 7.4 GB/s bandwidth could thus be reached in a scenario where the metadata providers and data providers would be deployed on separate physical nodes. We can further notice the aggregated bandwidth always increases when adding metadata providers, however the improvement is not uniform. Beyond 10 metadata providers (in this experiment), the corresponding performance gain becomes less significant.

6 Conclusion

This paper addresses the problem of data management in Desktop Grids. While classic approaches rely on centralized mechanisms for data management, some recent proposals aim at making Desktop Grids suitable for applications which need to access large amounts of input data. These approaches rely on data distribution using P2P overlays or Content Distribution Networks. We make a step further and propose an approach enabling Desktop Grids to also cope with *write-intensive distributed applications, under potentially heavy concurrency*. Our solution relies on BlobSeer, a blob management service which specifically addresses the issues mentioned above. It implements an efficient storage solution by gathering a *large aggregated storage capacity* from the physical nodes participating to the Desktop Grid.

By combining data fragmentation and striping with an efficient *distributed* metadata management scheme, BlobSeer allows applications to efficiently access data within huge data blobs. The algorithms used by BlobSeer enable a *high write throughput under heavy concurrency*: for any blob update, the new data may asynchronously be sent and stored in parallel on data providers, with no synchronization. Metadata is then also built and stored in parallel, with minimal synchronization. Moreover, BlobSeer provides efficient versioning support thanks to its lock-free design, allowing multiple concurrent writers to efficiently proceed in parallel. Storage is handled in a *space-efficient* way by sharing data and metadata across successive versions. Finally, note that in BlobSeer, accessing data sequentially or randomly has the same cost.

The main contribution of this paper is to explain how the BlobSeer approach fits the needs write-intensive applications and to support our claims through extensive experimental evaluations, which clearly demonstrate the efficiency of our decentralized approach. As a next step, we are currently experimenting efficient ways of using replication and dynamic group management algorithms, in order to address volatility and fault tolerance issues, equally important in Desktop Grids.

7 Acknowledgments

This work was supported by the French National Research Agency (LEGO project, ANR-05-CIGC-11). Experiments were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Washington, DC, USA, IEEE Computer Society (2004) 4–10
2. Fedak, G., Germain, C., Neri, V.: Xtremweb: A generic global computing system. In: CCGRID '01: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, Press (2001) 582–587
3. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: Architecture and performance of an enterprise desktop grid system. Journal of Parallel and Distributed Computing **63** (2003) 597–610
4. Costa, F., Silva, L., Fedak, G., Kelley, I.: Optimizing data distribution in desktop grid platforms. Parallel Processing Letters (PPL) **18** (2008) 391 – 410
5. Al-Kiswany, S., Ripeanu, M., Vazhkudai, S.S., Gharaibeh, A.: stdchk: A checkpoint storage system for desktop grid computing. In: ICDCS '08: Proceedings of the the 28th International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2008) 613–624
6. : Lustre file system: High-performance storage architecture and scalable cluster file system. White Paper, http://wiki.lustre.org/index.php/Lustre_Publications (2007)
7. Carns, P.H., III, W.B.L., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for linux clusters. In: ALS '00: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USA, USENIX Association (2000) 317–327
8. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. SIGOPS Oper. Syst. Rev. **37**(5) (2003) 29–43
9. You, L.L., Pollack, K.T., Long, D.D.E.: Deep store: An archival storage system architecture. In: ICDE '05: Proceedings of the 21st International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2005) 804–8015
10. : Respower render farm. http://www.respower.com (2003)
11. Patoli, Z., Gkion, M., Al-Barakati, A., Zhang, W., Newbury, P.F., White, M.: How to build an open source render farm based on desktop grid computing. In Hussain, D.M.A., Rajput, A.Q.K., Chowdhry, B.S., Gee, Q., eds.: IMTIC. Volume 20 of Communications in Computer and Information Science., Springer (2008) 268–278
12. Nicolae, B., Antoniu, G., Bougé, L.: How to enable efficient versioning for large object storage under heavy access concurrency. In: EDBT '09: 2nd International Workshop on Data Management in P2P Systems (DaMaP '09), St Petersburg, Russia (2009)
13. Nicolae, B., Antoniu, G., Bougé, L.: Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection. In: CLUSTER '08: Proceedings of the 2008 IEEE International Conference on Cluster Computing, Tsukuba, Japan (2008) 310–315
14. Nicolae, B., Antoniu, G., Bougé, L.: Distributed management of massive data: An efficient fine-grain data access scheme. In: VECPAR '08: Proceedings of the 8th International Meeting on High Performance Computing for Computational Science, Toulouse, France (2008) 532–543
15. Zheng, C., Shen, G., Li, S., Shenker, S.: Distributed segment tree: Support of range query and cover query over dht. In: IPTPS '06: The Fifth International Workshop on Peer-to-Peer Systems, Santa Barbara, USA (2006)
16. : Boost c++ libraries. http://www.boost.org (2008)
17. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. **20**(4) (2006) 481–494