HAL
open science

BitDew: A Programmable Environment for Large-Scale Data Management and Distribution

Gilles Fedak, Haiwu He, Franck Cappello

To cite this version:

Gilles Fedak, Haiwu He, Franck Cappello. BitDew: A Programmable Environment for Large-Scale Data Management and Distribution. [Research Report] 2008, pp.24. inria-00216126v1

HAL Id: inria-00216126

https://inria.hal.science/inria-00216126v1

Submitted on 24 Jan 2008 (v1), last revised 22 Apr 2008 (v4)

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BitDew: A Programmable Environment for Large-Scale Data Management and Distribution

Gilles Fedak — Haiwu He — Franck Cappello

N° ????

January 2008

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

BitDew: A Programmable Environment for Large-Scale Data Management and Distribution

Gilles Fedak , Haiwu He , Franck Cappello

Thème SYM — Systèmes symboliques
Équipes-Projets Grand Large

Rapport de recherche n° ???? — January 2008 — 24 pages

Abstract:

Desktop Grids use the computing, network and storage resources from idle desktop PC's distributed over multiple-LAN's or the Internet to compute a large variety of resource-demanding distributed applications. While these applications need to access, compute, store and circulate large volumes of data, little attention has been paid to data management in such large-scale, dynamic, heterogeneous, volatile and highly distributed Grids. In most cases, data management relies on ad-hoc solutions, and providing general approach is still a challenging issue.

To address this problem, we propose the BitDew framework, a programmable environment for automatic and transparent data management on computational Desktop Grids. This paper describes the BitDew programming interface, its architecture, and the performance evaluation of its runtime components. BitDew relies on a specific set of meta-data to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction. The Bitdew runtime environment is a flexible distributed service architecture that integrates modular P2P components such as DHT's for a distributed data catalog and collaborative transport protocols for data distribution. Through several examples, we describe how application programmers and Bitdew users can exploit Bitdew's features. The performance evaluation demonstrates that the high level of abstraction and transparency is obtained with a reasonable overhead, while offering the benefit of scalability, performance and fault tolerance with little programming cost.

Key-words: Data Management, Desktop Grids

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

BitDew: un environnement programmable pour la gestion et la diffusion des données à large échelle

Résumé : Les Grilles de PCs utilisent les capacités de calcul, de communication et de stockage d'ordinateurs personnels distribués sur plusieurs LAN ou sur l'Internet. La gestion des données dans ces grilles de grande échelle, dynamiques, hétérogènes, volatiles et hautement distribuées est un défi qui doit être relevé pour étendre l'usage des Grilles de PCs.

Nous proposons le logiciel BitDew, un environnement programmable pour la gestion et la distribution des données sur les Grilles de PCs. Ce rapport de recherche présente l'interface de programmation de BitDew, son architecture ainsi que les évaluations de performance des composants de l'environnement d'exécution. Nous décrivons l'API qui, avec un haut niveau d'abstraction et de transparence, contrôle les opérations de gestion des données : cycle de vie, distribution, placement, réplication et la tolérance aux pannes. Notre environnement d'exécution repose sur une architecture distribuée flexible, qui intègre de façon modulaire des composants P2P tels qu'une DHT pour implémenter un catalogue distribué de données et BitTorrent pour la distribution des données. Dans ce rapport de recherche nous effectuons une évaluation de performance de ces composants, et nous évaluons la scalabilité et l'efficacité de l'environnement lors de l'exécution de l'application BLAST.

Mots-clés : Gestion des données, Grille de PC

1 Introduction

Enabling Data Grids is one of the fundamental efforts of the computational science community as emphasized by projects such as EGEE [14] and PPDG [32]. This effort is pushed by the new requirements of e-Science. That is, large communities of researchers collaborate to extract knowledge and information from huge amounts of scientific data. This has lead to the emergence of a new class of applications, called *data-intensive* applications which require secure and coordinated access to large datasets, wide-area transfers and broad distribution of TeraBytes of data while keeping track of multiple data replicas. The Data Grid aims at providing such an infrastructure and services to enable data-intensive applications.

Our project, BitDew[1], targets a specific class of Grids called Desktop Grids. Desktop Grids use computing, network and storage resources of idle desktop PCs distributed over multiple LANs or the Internet. Today, this type of computing platform forms one of the the largest distributed computing systems, and currently provides scientists with tens of TeraFLOPS from hundreds of thousands of hosts. Despite the attractiveness of this platform, little work has been done to support data-intensive applications in this context of massively distributed, volatile, heterogeneous, and network-limited resources. Most Desktop Grid systems, like BOINC [4], XtremWeb [15], Condor [28] and OurGrid [6] rely on a centralized architecture for indexing and distributing the data, and thus potentially face issues with scalability and fault tolerance.

However, we believe that the basic blocks for building BitDew can be found in P2P systems. Researchers of DHT's (Distributed Hash Tables) [38, 30, 34] and collaborative data distribution [12, 20, 16], storage over volatile resources [1, 11, 40] and wide-area network storage [9, 27] offer various tools that could be of interest for Data Grids. To build Data Grids from and to utilize them effectively, one needs to bring together these components into a comprehensive framework. BitDew suits this purpose by providing an environment for data management and distribution in Desktop Grids.

BitDew is a subsystem which could be easily integrated into other Desktop Grid systems. It offers programmers (or an automated agent that works on behalf of the user) a simple API for creating, accessing, storing and moving data with ease, even on highly dynamic and volatile environments.

BitDew leverages the use of metadata, a technics widely used in Data Grid [23], but in more directive style. We define 5 different types of metadata : *i*) *replication* indicates how many occurrences of data should be available at the same time in the system, *ii*) *fault tolerance* controls the resilience of data in presence of machine crash, *iii*) *lifetime* is a duration, absolute or relative to the existence of other data, which indicates when a datum is obsolete, *iv*) *affinity* drives movement of data according to dependency rules, *v*) *transfer protocol* gives the runtime environment hints about the file transfer protocol appropriate to distribute the data. Programmers tag each data with these simple attributes, and simply let the BitDew runtime environment manage operations of data creation, deletion, movement, replication, as well as fault tolerance.

The BitDew runtime environment is a flexible environment implementing the APIs. It relies either on centralized or and distributed protocols for indexing, storage and transfers providing reliability, scalability and high performance. In this paper, we present the architecture of the prototype, and we describe in depth the various mechanisms used. We also provide detailed quantitative evaluation of the runtime environment on two environments : the GRID5000 experimental Grid platform, and DSL-Lab, an experimental platform over broadband ADSL.

Through a set of micro-benchmarks, we measure the costs and benefits, components by components, of the underlying infrastructures. We run communication benchmark in order to evaluate the overhead of the BitDew protocol when transferring files and we assess fault-tolerant

[1] BitDew can be found at `http://www.bitdew.net` under GPL license

capabilities. And finally we show how to program a master/worker application with BitDew and we evaluate its performance in a real world Grid deployment.

The rest of the paper is organized as follows. Section 2 presents the background of our researches. In Section 3, we present the API and the runtime environment of BitDew. Then in Section 4, we conduct performance evaluation of our prototype, and Section 5 presents a master/worker application. Finally we present related work in Section 6 and we conclude the paper in Section 7.

2 Background

In this section we overview Desktop Grids characteristics and data-intensive application requirements. Following this analysis, we give the required features of BitDew.

2.1 Desktop Grids Characteristics

Desktop Grids are composed of a large set of personal computers that belong both to institutions, for instance an enterprise or a university, and to individuals. In the former case, these home PCs are volunteered by participants who donate a part of their computing capacities to some public projects. However several key characteristics differentiate DG resources from traditional Grid resources : *i*) performance; mainstream PCs have no reliable storage and potentially poor communication links, *ii*) volatility; PCs can join and leave the network at any time and appear with several identities, *iii*) shared between their users and the desktop grid applications, *iv*) scattering across administrative domains with a wide variety of security mechanisms ranging from personal routers/firewalls to large-scale PKI infrastructures.

Because of these constraints, even the simplest data administration tasks, are difficult to achieve on a Desktop Grid. For instance, to deploy a new application on a cluster, it is sufficient to copy the binary file on a network file server shared by the cluster nodes. After a computation, cluster users usually clean the storage space on the cluster nodes simply by logging remotely to each of the compute nodes and by deleting recursively the temporary files or directories created by the application. By contrast, none of the existing Desktop Grids systems allows such tasks to be performed because : *i*) a shared file system would be troublesome to setup because of hosts connectivity and volatility and volunteers churn, and *ii*) remote access to participant's local file system is forbidden in order to protect volunteer's security and privacy.

2.2 Requirements to Enable Data-Intensive Application on Desktop Grids

Currently, Desktop Grids are mostly limited to embarrassingly parallel applications with few data dependencies. In order to broaden the use of Desktop Grids we examine several challenging applications and outline their needs in terms of data management. From this survey, we will deduce the features expected from BitDew.

Parameter-sweep applications composed of a large set of independent tasks sharing large data are the first class of applications which can benefit from BitDew. Large data movement across wide-area networks can be costly in terms of performance because bandwidth across the Internet is often limited, variable and unpredictable. Caching data on local workstation storage [21, 31, 40] with adequate scheduling strategies [35, 41] to minimize data transfers can improve overall application execution performance.

Moreover, the work in [22] showed that data-intensive applications in high energy physics tend to access data in groups of files called "filecules". For these types of applications, replication of

groups of files over a large set of resources is essential to achieve good performance. If data are replicated and cached on local storage of computing resources, one should provide transparent fault tolerance operation on data.

In a previous work [41], we have shown that using a collaborative data distribution protocol BitTorrent over FTP can improve execution time of parameter sweep applications. In contrast, we have also observed that the BitTorrent protocol suffers a higher overhead compared to FTP when transferring small files. Thus, one must be allowed to select the correct distribution protocol according to the size of the file and level of "sharability" of data among the task inputs.

The high-level of collaboration in e-Science communities induces the emergence of complex workflow applications [7]. For instance in the case of multi-stage simulations, data can be both results of computation and input parameters for other simulations. To build execution environment for applications with task and data dependencies, it requires a system that can move data from one node to another node according to dependency rules. A key requirement of the system is to efficiently publish, search and localize data. Distributed data structures such as the DHT proposed by the P2P community might fulfill this role by providing distributed index and query mechanisms.

Long-running applications are challenging due to the volatility of executing nodes. To achieve application execution, it requires local or remote checkpoints to avoid losing the intermediate computational state when a failure occurs. In the context of Desktop Grid, these application have to cope with replication and sabotage. An idea proposed in [25] is to compute a signature of checkpoint images and use signature comparison to eliminate diverging execution. Thus, indexing data with their checksum as is commonly done by DHT and P2P software permits basic sabotage tolerance even without retrieving the data.

2.3 BitDew Features

Previously, we profiled several classes of "data-bound" applications and we now give the expected features to efficiently manage data on Desktop Grids.

- Fault tolerance: the architecture should handle frequent faults of volatile nodes which can leave and join the network at any time.
- Scalability: the architecture should provide a decentralized approach when a centralized approach might induce a performance bottleneck.
- Replication: to achieve application performance, the system should allow data replication and distributed data cache.
- Efficiency: the architecture should provide adequate and user optional protocols for both high throughput data distribution and low latency data access.
- Reliability: interrupted transfers should be automatically resumed or canceled according to the programmer's preference.
- Simplicity: the programming model should offer a simple view of the system, unifying the data space as a whole.
- Transparency: faults and data location should be kept hidden from the programmer.

We have designed our system to address each of those design goals in mind. In this paper, we give evidence for the system's manageability, scalability, efficiency, and simplicity by performing a set of micro-benchmarks and by deploying a real scientific application.

Security issues are not specifically addressed in this paper, because existing solutions in literature could be applied to our prototype. In fact, a relevant analysis of security for Data Desktop Grid has been done by [29], where is also proposed a protocol to maintain data confidentiality given that data will be stored on untrusted nodes. Authors use methods known as Information Dispersal Algorithms (ISA) which allows one to split a file into pieces so that by carefully dispersing the pieces, there is no method for a single node to reconstruct the data. Another well known issue is protection against data tampering, which has been addressed in the literature under the generic name of "results certification" [36]. It is a set of methods (spot-checking, voting, credibility) to verify that results computed by volunteers are not erroneous (for example, because of intentional modifications by malicious participants). Result certification is also mandatory to protect the DG storage space. In public DG system, the information to upload task result could be exploited by malicious participants. We assume that a result checker such as the *assimilator* of BOINC exists to sort between correct and incorrect results. Furthermore, in a Volunteer Computing setup, BitDew should be run in a confined environment, such as a sandbox [24] to protect volunteer's privacy and security. For future work, we will show the system's ability to deal with security issues.

Also, in this paper we consider data as immutable. However one could leverage the built-in distributed data catalog to provide data consistency. For example, authors in [8] have proposed an *entry consistency* protocol for a P2P system with mutable data.

3 BitDew Architecture

In this section we detail the BitDew architecture: programing interface, runtime environment and implementation.

3.1 Overview

Figure 1 illustrates the layered BitDew software architecture upon which distributed application can be developed. The architecture follows strict design rules : each layer is composed of independent components; components of the same layer do not interact directly and a component of an upper layer only interacts with components of the immediate lower layer.

The uppermost level, the APIs level, offers the programmer a simplified view of the system, allowing him to create data and manage their repartition and distribution over the network of nodes. The programming model is similar to the Tuple Space model pioneered by Gelernter [17] in the Linda programming system; it aggregates the storage resources and virtualizes it as a unique space where data are stored. The BitDew APIs provide functions to create a slot in this space and to put and get files between the local storage and the data space. Additional metadata, called data attributes, are managed by the ActiveData API and help to control the behavior of the data in the system, named *replication*, *fault tolerance*, *placement*, *lifetime* and *distribution*. It also provides programmers event-driven programming facilities to react to the main data life-cycle events: creation, copy and deletion. Finally the TransferManager API offers a non-blocking interface to concurrent file transfers, allowing users to probe for transfer, to wait for transfer completion, to create barriers and to tune the level of transfers concurrency.

The intermediate level is the service layer which implements the API : data storage and transfers, replicas and volatility management. The architecture follows a classical approach commonly found in Desktop Grids: it divides the world in two sets of nodes : stable nodes and volatile nodes. Stable nodes run various independent services which compose the runtime environment: Data Repository (DR), Data Catalog (DC), Data Transfer (DT) and Data Scheduler (DS). We call these nodes the *service hosts*. The fault model we consider for service node is the transient

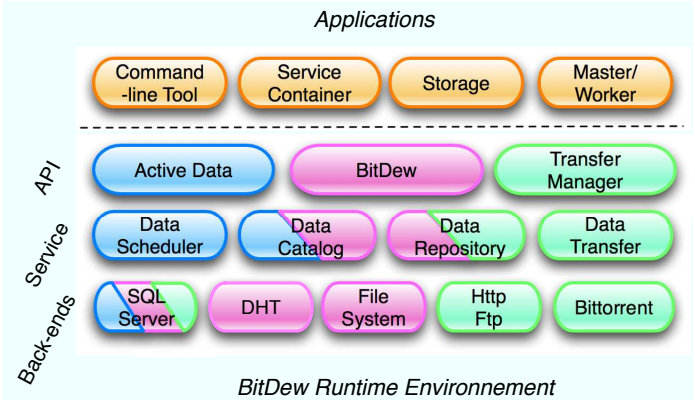


Figure 1: The BitDew software architecture. The upper part of the figure shows distributed applications designed using BitDew. Lower parts are the three layers composing the BitDew run-time environment : the API layer, the service layer and the back-ends layer. Colors illustrate how various components of different layers combine together. For instance, the TransferManager API uses two services : Data Repository and Data Transfer, which in turn use three back-ends : SQL Server, Http/FTP protocol and BitTorrent protocol.

fault where a host is assumed to be restarted by administrators after a failure. Volatile nodes can either ask for storage resources (we call them *client hosts*) or offer their local storage (they are called *reservoir hosts*). Classically in DG, we use a *pull model*, where volatile nodes periodically contact service nodes to obtain data and synchronize their local data cache. Failures of volatile nodes is detected by the mean of timeout on periodical heartbeats. Usually, programmers will not use directly the various D* services; instead they will use the API which in turn hides the complexity of internal protocols.

The lowermost level is composed of a suite of back ends. The Bitdew runtime environment delegates a large number of operations to third party components : 1) Meta-data information are serialized using a traditional SQL database, 2) data transfers are realized out-of-band by specialized file transfer protocols and 3) publish and look-up of data replica are enabled by the means of DHT protocols. One feature of the system is that all of these components can be replaced and plugged-in by the users, allowing them to select the most suitable subsystem according to their own criteria like performance, reliability and scalability.

3.2 Data Attributes

The key feature of BitDew is to leverage on metadata, called here Data Attributes. Though, metadata are not only used to index, categorize, and search data, as in other Data Grids System, but also to control dynamically repartition and distribution of data onto the storage nodes. Thus, complexity of Desktop Grids systems is hidden to the programmers who is freed from managing data location, host failure and explicit host to host data movement.

Instead, the runtime environment interprets data attributes and schedule data to host in order to satisfy the constraints expressed by the attributes. The following is the list of attributes a user can set :

`replica`: gives the number of instances of a datum that should exist at the same time in the system. The runtime environment will schedule new data transfers to hosts if the number of owners is less than the number of replica. As nodes are volatile there might be more replicas

in the system than what is specified by this attribute because the runtime environment will not issue orders for data deletion.

`fault tolerance`: indicates what the runtime environment should do if a reservoir host holding a data replica fails. If the data is resilient to host crash (the *fault tolerance* attribute is set), the data will be scheduled to another node so that the number of available replicas is kept at least equal to the value of the *replica* attribute over time. If the data are not marked as fault tolerance, the replica will be unavailable as long as the host is down.

`lifetime`: defines data lifetime, that is precise time after which a datum can be safely deleted by the storage host. The lifetime can be either absolute or relative to the existence of the other data. In the latter case, a datum is obsolete when the reference data disappear.

`affinity`: defines the placement dependency between data. It indicates that data should be scheduled on a node where other data have been previously sent. The *affinity* attribute is stronger than *replica*. That is, if data A is r_a *replica* and is distributed to r_n nodes, then if a datum B has a placement dependency over A, it will be replicated over r_n nodes whatever the value of r_b or r_a is.

`transfer protocol`: specifies to the runtime environment the preferred transfer protocol to distribute the data. Users are more knowledgeable to select the most appropriate protocol according to their own criteria like the size of data and the number of nodes to distribute these data. For example a large file distributed to a large number of nodes would be preferably distributed using collaborative distribution protocol such as BitTorrent or Avalanche [41].

3.3 Application Programing Interfaces

We will now give a brief overview of the three main programming interfaces which allows the manipulation of the data in the storage space (BitDew), the scheduling and programming (ActiveData) and the control of file transfer (TransferManager).

To illustrate how APIs are put into action, we'll walk through a toy program which realizes a network file update and works as follows : one master node, the Updater, copies a file to each node in the network, the Updatee, and maintains the list of nodes which have received the file updated. The Listing 1 presents the code of the Updater[2], implemented using Java.

```
1  public class Updater {
2    //list of hosts updated
3    Vector updatees = new Vector();
4
5    public Updater(String host, int port, boolean master) {
6
7      //initialize  communications and APIs
8      Vector comms=ComWorld.getMultipleComms(host, "RMI", port, "dc","dr","dt","ds");
9      BitDew bitdew = new BitDew(comms);
10     ActiveData activeData = new ActiveData(comms);
11     TransferManager tranferManager = new TransferManager(comms);
12
13     if (master) {
14       //this part of the code will only run on the master
15       File fic = new File("/path/to/big_data_to_update");
16       Data data = bitdew.createData(fic);
17       bitdew.put(data, fic); //copy file to the data space
18       //attribute specifies that the data should be send to every node using the
             BitTorrent protocol, and has a lifetime of 30 days
19       Attribute attr = bitdew.createAttribute("attr update = { replicat=-1, oob=
             bittorrent, abstime=43200}");
20       activeData.schedule(data, attr); //schedule data
21       activeData.addCallback(new UpdaterHandler()); //install data life-cycle event
             handler
22     } else {
```

[2]For sake of clarity we have simplified some elements of syntax, however the full source code is available in the BitDew source package.

```
23        //this part of the code will be executed by the other nodes,
24        activeData.addCallback(new UpdateeHandler()); //install data life-cycle event
              handler
25      }
26    }
27  }
```

Listing 1: The Updater example.

```
1   public class UpdaterHandler extends ActiveDataEventHandler {
2     public void onDataCopyEvent(Data data, Attribute attr) {
3       if (attr.getname().equals("host")) {
4         updatees.add(data.getname());
5       }
6     }
7   }
8   public class UpdateeHandler extends ActiveDataEventHandler {
9     public void onDataCopyEvent(Data data, Attribute attr) {
10      if (attr.getname().equals("update")) {
11        //copy file from the data space
12        bitdew.get(data, new File("/path_to_data/to/update/"));
13        transferManager.waitFor(data); //block until the download is complete
14        Data collectorData = bitdew.searchData("collector");
15        //sends back to the updater the name of the host
16        activeData.schedule( bitdew.createData(getHostByName()),
17                             activeData.createAttribute("attr host = { affinity = " +
                                 collector.getuid() + "}");
18
19      }
20    }
21    public void onDataDeleteEvent(Data data, Attribute attr) {
22      //delete the corresponding file
23      if (attr.getname().equals("update"))
24        (new File("/path_to_data/to/update/")).delete();
25    }
26  }
```

Listing 2: Data life-cycle events handlers installed in the Updater example.

Before a user can start data manipulation, he firstly has to attach the host to the rest of the distributed system. For this purpose, a special class called CommWorld will set up the communication to the remote hosts executing the different runtime services (DC, DR, DT, DS). The result of this operation is a set of communication interfaces to services passed as parameters to the APIs constructor. After this step, the user does never have to explicitly communicate with service hosts. Instead the complexity of the protocol is kept hidden unless programmer wishes to perform specialized operations.

In the Updater example we have assumed that all D* services are executed on a single centralized node. However, in real world, it might be necessary to run several service nodes in order to enhance reliability and scalability or to adjust with an existing infrastructure where data are spread over multiple data servers. The BitDew approach to cope with distributed setup is to instantiate several APIs, each one configured with its own vector of interfaces to the D* pool.

Data creation consists of the creation of a slot in the storage space. This slot will be used to put and get content, usually a file, to and from that slot. A data object contains data meta-information: `name` is the character string label, `checksum` is an MD5 signature of the file, `size` is the file length, `flags` is a OR-combination of flags indicating whether the file is compressed, executable, architecture dependent, etc... The BitDew API provides methods which compute these meta-information when creating a datum from a file. Data objects are both locally stored within a database and remotely on the Data Catalog service. Consequently data deletion implies both local and remote deletion.

Once slots are created in the storage space, users can copy files to and from the slots using dedicated functions. However users have several ways of triggering data movement either explic-

itly or implicitly. Explicit transfers are performed via `put` and `get` methods, which copy data to the storage space slots. Implicit transfers occur as a result of affinity placement, fault tolerance or replication and are resolved dynamically by the Data Scheduling service.

This is precisely the role of the ActiveData API to manage data attributes and interface with the DS, which is achieved by the following methods: *i*) `schedule` associates a datum to an attribute and order the DS to schedule this data according to the scheduling heuristic presented in paragraph 3.4.3; *ii*) `pin` which, in addition, indicates the DS that a datum is owned by a specific node. Besides, ActiveData allows programmer to install *handlers*, those are codes executed when some events occur during data life cycle : creation, copy and deletion.

The API provides functions to publish and search data over the entire network. Data are automatically published by hosts by means of a DHT. Moreover, the API also gives the programmer the possibility to publish any key/value pairs so that the DHT can be used for other generic purpose.

3.4 Runtime Environment

We review now the various capabilities provided by the BitDew service layer of the runtime environment.

3.4.1 Indexing and locating data

The data's meta-information are stored both locally on the client/reservoir node and persistently on the Data Catalog (DC) service node.

For each data published in the DC, one or several Locators are defined. A Locator object is similar to URL, it gives the correct information to remotely access the data: file identification on the remote file system (this could be a path, file name, or hash key) and information to set up the file transfer service (for instance protocol, login and password).

However, information concerning data replica, that is data owned by volatile reservoir nodes, are not centrally managed by DC but instead by a Distributed Data Catalog (DDC) implemented on top of a DHT. For each data creation or data transfer to a volatile node, a new pair data identifier/host identifier is inserted in the DHT.

The rationale behind this design is the following : as the system grows, information in the DDC will grow larger than information in the DC. Thus, by centralizing information in the DC, we shorten the critical path to access to a permanent copy of data. On the other hand, distributing data replica information ensures scalability of the system for two reasons : *i*) DHTs are inherently fault-tolerant; thus it frees the DC to implement fault detection mechanisms and *ii*) the search requests are distributed evenly among the hosts, ensuring effective load-balancing.

3.4.2 Storing and Transferring data

The strength of the framework depends on its ability to adapt to various environments in term of protocols (client/server vs. P2P), of storage (local vs. wide area), of security level (Internet vs. Grid). To provide more flexibility, we have separated data access in two different services : Data Repository (DR) is an interface to data storage with remote access and Data Transfer service (DT) is responsible for reliable out-of-band file transfer.

The Data Repository service has two responsibilities, namely to interface with persistent storage and to provide remote access to data. DR acts as a wrapper around legacy file server or file system, such as Grid Ftp server or local file system. In a Grid context, DR is the solution to map BitDew to an existing infrastructure.

BitDew does not propose new protocol to transfer data from node to node, instead, data are moved by out-of-band transfer.

The role of Data Transfer (DT) is to launch out-of-band transfers and ensure their reliability. If several transfers of the same data occur in parallel (for a broadcast, for example), it is the responsibility of the file transfer protocol to leverage this concurrency. This is finally what happens when collaborative file transfer are being used, but this is transparent to the system.

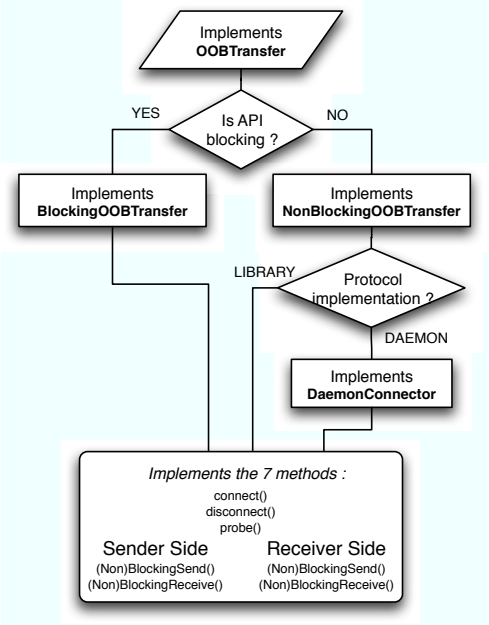


Figure 2: Flowchart to implement out-of-band transfer. To plug-in a new file transfer protocol, a programmer has to implement the OOBTransfer interface. Programmer chooses the blocking (resp. non blocking) interface if the method protocol are blocking (resp. non blocking). DaemonConnector is a helper interface for protocol provided as daemon instead of library. Finally, it is sufficient to write 7 methods : to open and close connection, to probe the end of transfer and to send and to receive file from the sender and the receiver sides.

Transfers are always initiated by a reservoir or client host to DT, which manages transfer reliability, resumes faulty transfers, reports on bandwidth utilization and ensures data integrity. Transfer management relies on a principle called *receiver driven transfer*. The sender of a datum will periodically pool the receiver to check the state of the transfer, because receiver can verify the size and the integrity, using the MD5 signature, of the received data. This mechanism, while simple, ensures support for a broad range of different data transfer protocols.

Figure 2 presents the framework to integrate existing or new file transfer protocols, client/server or P2P, with blocking or non-blocking communication, whose implementations are provided as libraries or as daemons. Note, that the former is very popular for P2P protocol where a daemon runs forever in the background and a GUI issues search and download order. So far, we support HTTP, FTP and BitTorrent, both as a library with Azureus[3] and as a daemon with BTPD and we tested the framework with SMTP, POP and edonkey.

[3]The Azureus BitTorrent Client: `http://azureus.sourceforge.net`

3.4.3 Scheduling data

Implicit data movement on the grid is determined by the Data Scheduling service (DS). The role of the DS service is to generate transfer orders according to the hosts' activity and data attributes.

Algorithm 1 presents the pseudo-code of the scheduling algorithm. Periodically, reservoir hosts contact the data scheduler with a description of the set of data hold in their local cache Δ_k. The data scheduler scans the list of data to schedule Θ, and according to data attributes, makes a scheduling decision which consists of a new set of data Ψ_k returned to the reservoir host. Reservoir host can safely delete obsolete data ($\Delta_k \setminus \Psi_k$), keep the cached data validated by the DS ($\Delta_k \cap \Psi_k$) and download newly assigned data ($\Psi_k \setminus \Delta_k$).

First step of the scheduling algorithm determines which data should be kept in reservoir cache. It is defined as the set of data both present in the reservoir cache Δ_k and in the DS data set Θ and whose lifetime, either absolute or relative, has not expired. In the second step, new data are scheduled to the reservoir host by filling Ψ_k. Two conditions trigger attribution of data to reservoir host. The first one is the dependency relations: if the reservoir cache Δ_k contains data which has a dependency relation with a datum missing from Δ_k, then this datum is added to Ψ_k. The second one is the replica attribute: if the number of active owners $\Omega(D_i^k)$ is less than the the value of *replica* attribute then this data is added to Ψ_k. The scheduling algorithm stops when the set of new data to download ($\Psi_k \setminus \Delta_k$) has reached a threshold.

Finally the Data Scheduler implements support for fault tolerance. For each data is maintained a list of *active* owners updated at each synchronization of reservoir hosts. Faults of owners are detected through timeout on the last synchronization. If a datum has the *fault tolerance* attribute, the faulty owner is removed from the list of active owners, otherwise the list is kept unchanged. As a consequence, the data will be scheduled again to a new host.

For now the scheduling has been designed to fulfill metadata specification without focus on performance. In future, we will investigate specific data repartition policies, cache management strategies and coordinated tasks scheduling heuristics.

3.5 Implementation

We have used the Java programming environment to prototype BitDew with Java RMI for the communication and Apache Jakarta Java JDO (`http://jakarta.apache.org`) with JPOX (`http://jpox.org`) which permits transparent objects persistence in a relational database. Each object is referenced with a unique identifier AUID, a variant of the DCE UID.

We have used two different database back-ends. MySQL (`http://mysql.com`), is a well-known open-source database, and HsqlDB (`http://hsqldb.org`) is an embedded SQL database engine written entirely in Java. Jakarta Commons-DBCP provides database connection pooling services, which avoids opening new connection for every database transaction. We have implemented data transfer with the client/server FTP protocol, respectively the client provided by the apache common-net package and the ProFTPD FTP server (`http://www.proftpd.org/`) and with the BTPD BitTorrent client (`http://www.murmeldjur.se/btpd/`). The distributed data catalog uses the DKS DHT [2].

Overall our first version of the software, while implementing most of the features described in the paper, includes less than 17000 lines of code. Initial release is available at `http://bitdew.net` under GNU GPL.

Algorithm 1 SCHEDULING ALGORITHM

Require: $\Theta = \{D_1, \cdots, D_m\}$ the set of data managed by the scheduler
Require: $\Delta_k = \{D_1^k, \cdots, D_n^k\}$ the data cache managed by the reservoir host k
Require: $\Omega(D_i^k) = \{k, \cdots\}$ the set of reservoir host owning data D_i
Ensure: $\Psi_k = \{D_1^k, \cdots, D_o^k\}$ the new dataset managed by the reservoir host k

$\Psi_k \leftarrow \emptyset$
{Step 1 : Remove obsolete data from cache}
for all $D_i^k \in \Delta_k$ **do**
  if $((D_i^k \in \Theta) \wedge (D_i^k.lifetime.absolute > now()) \wedge (D_i^k.lifetime.relative \in \Theta))$ **then**
    $\Psi_k \leftarrow \Psi_k \cup \{D_i^k\}$
    if $((D_i^k.faultTolerant == true)$ **then**
      update $\Omega(D_i^k)$
    end if
  end if
end for
{Step 2 : Add new data to the cache}
for all $D_j \in (\Theta \setminus \Delta_k)$ **do**
  {Resolve affinity dependence}
  for all $D_i^k \in \Delta_k$ **do**
    if $((D_j.affinity == D_i^k) \wedge (D_j \notin \Delta_k))$ **then**
      $\Psi_k \leftarrow \Psi_k \cup \{D_j\}$
      $\Omega(D_j) \leftarrow \Omega(D_j) \cup \{k\}$
    end if
  end for
  {Schedule replica}
  if $((D_j.replica == -1) \vee (D_j.replica < |\Omega(D_j)|)$ **then**
    $\Psi_k \leftarrow \Psi_k \cup \{D_j\}$
    $\Omega(D_j) \leftarrow \Omega(D_j) \cup \{k\}$
  end if
  if $(|\Psi_k \setminus \Delta_k| \geq MaxDataSchedule)$ **then**
    break
  end if
end for

return Ψ_k

4 Performance Evaluation

In this section, we present performance evaluation of the BitDew runtime environment. The experiments evaluate the efficiency and scalability the core data operation, the data transfer service, and the data distributed catalog. We also report on a Master/Worker bioinformatics application executed over 400 nodes in a Grid setup.

4.1 Experiments Setup

Experiments were conducted in 3 different testbeds. To measure precisely performances of basic data operations within an environment where experimental conditions are reproducible, we run

Cluster	Cluster Type	Location	#CPUs	CPU Type	Frequency	Memory
gdx	IBM eS-erver 326m	Orsay	312	AMD Opteron 246/250	2.0G/2.4G	2G
grelon	HP Pro-Liant DL140G3	Nancy	120	Intel Xeon 5110	1.6G	2G
grillon	HP Pro-Liant DL145G2	Nancy	47	AMD Opteron 246	2.0G	2G
sagittaire	Sun Fire V20z	Lyon	65	AMD Opteron 250	2.4G	2G

Table 1: Hardware configuration of the Grid testbed which consists in 4 Grid5000 clusters.

micro-benchmarks on the *Grid Explorer (GdX)* cluster which is part of the Grid5000 infrastructure [10].

To analyze BitDew behavior on a platform close to Internet Desktop Grids, we conducted experiments with the DSL-Lab platform [4]. DSL-Lab is an experimental platform, consisting of a set of PCs connected to broadband Internet. DSL-lab nodes are hosted by regular Internet users, most of the time protected behind firewall and sharing the Internet bandwidth with users' applications. DSL-lab offers extremely realistic networking experimental conditions, since it runs experiments on the exact same platform than the one used by most of the desktop Grids applications. Technically, it's a set of 12 Mini-ITX nodes, Pentium-M 1Ghz, 512MB SDRam, with 2GB Flash storage.

The third testbed, used for scalability tests, is a part of Grid5000: 4 clusters (including GdX) of 3 different sites in France. Note that, due to the long running time of our experiments, we were not able to reserve the 2000 nodes of Grid5K. The hardware configuration is shown in table 1. All of our computing nodes are installed Debian GNU/Linux 4.0 as their operating systems.

As for software, we used the latest version of the software package available at the time of the experiment (July to December 2007). Java SDK 1.5 for 64 bits was the Java version we used for the experiments. BLAST used is NCBI BLAST 2.2.14 for Linux 64 bits.

4.2 Core data operation

We first report on the performance of basic data operations according to the database and communication components.

The benchmark consists of a client running a loop which continuously creates data slot in the storage space, and a server running the Data Catalog service. This basic operation implies an object creation on the client, a network communication from the client to server (payload is only few kilobytes) and an a write access to the database to serialize the object. Every ten seconds the average number of data creations per second (dc/sec) is reported. Table 2 shows the peak performance in thousands of dc/sec. This benchmark is representative of most of the data operations executed by the different D* services when the runtime environment manages data transfers, fault tolerance and replication.

The experimental configuration is as follows: with the *local* experiment, a simple function call replaces the client/server communication, with *RMI local* the client and server are hosted on the same machine and a RMI communication takes place between them; with *RMI remote* client and server are located in two different machines. We have used two different database engines

[4]DSL-lab: `http://dsllab.org`. Note that DSL-lab is currently an early prototype. At the time of the experiments, only 12 nodes were available.

	without DBCP		with DBCP	
	MySQL	HsqlDB	MySQL	HsqlDB
local	0.25	3.2	1.9	4.3
RMI local	0.21	2.0	1.5	2.8
RMI remote	0.22	1.7	1.3	2.1

Table 2: Performance evaluation of data slot creation (dc/sec) : the number is expressed as thousands of data creation per second.

	Min	Max	Sd	Mean
publish/DDC	100.71	121.56	3.18	108.75
publish/DC	2.20	22.9	5.05	7.02

Table 3: Performance evaluation of data publishing in the centralized and distributed data catalog : the number are expressed as pairs (dataID,hostID) create per second.

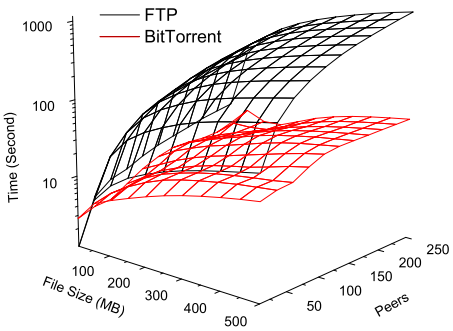
MySQL and HsqlDB; each database can be accessed either directly (*without DBCP*) or through (*with DBCP*) the use of the connection pooling service DBCP.

Preliminary results show that the latency for a remote data creation is about $500\mu sec$ when considering the HsqlDB database with DBCP. Using an embedded database provides a performance improvement of 61% over a more traditional MySQL approach, but comes at a price of manageability. Indeed, there exist numerous language bindings and third-party tools for MySQL, which can make the system more manageable. This lack of performance is certainly due to the networked client server protocol imposed by the MySQL Java JDBC connector. Moreover we can see that MySQL without the use of a connection pool is clearly a bottleneck when compared to the network overhead. However, a single service is able to handle more than 2 thousand data operations per second and we think that there is room for further performance improvements by using multi-threaded remote service invocations and by enclosing burst of data operations in a single remote invocations.

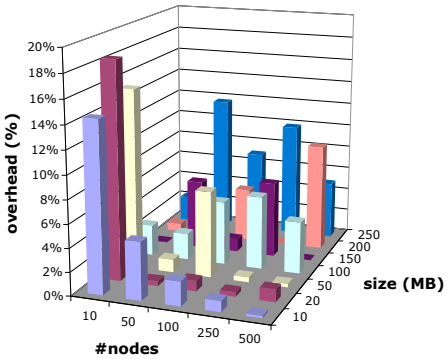
The next experiment evaluates the effectiveness of the Distributed Data Catalog (DDC) through the measurement of the DHT publish and search mechanism. The benchmark consists of an SPMD program running on 50 nodes. After a synchronization, each node will publish 500 pairs of dataID, hostID values, an operation which is executed every time a host completes a data transfer. We measure the time elapsed between the first data published to the last publication in the DHT, and we report in Table 3 the total time to create the data. One can observe that indexing 25000 data in the DDC takes about 108 sec. We conducted a similar experience with the DC and we found out that DDC is 15 time slower than DC. However, it is not fair to compare a DHT with a centralized approach as DC service do not implement fault-tolerance. Nevertheless, this result validates the design decision presented in paragraph 3.4.1 to rely both on a centralized Data Catalog service to provide fast access to data and a DHT for data replica hold by volatile nodes.

4.3 Data transfer

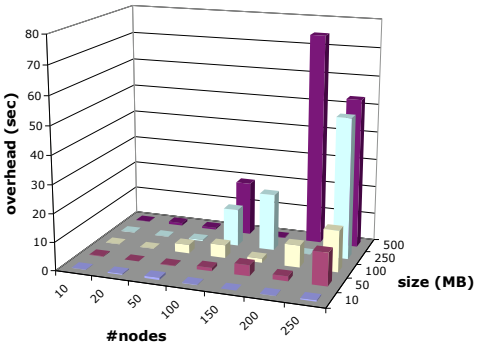
The following experiment evaluates the overhead of BitDew when transferring data, and Figure 3 presents the results. In a previous study [41], we have compared the BitTorrent and FTP protocols for computational Desktop Grids. Here BitDew issues and drives the data transfer, providing file transfer reliability. As a consequence, both the BitDew protocol and the file transfer



(a) BitDew performance with two file transfer protocols : BitTorrent and FTP.



(b) Overhead of BitDew with FTP against FTP alone in percentage of the communication time.



(c) Overhead of BitDew with FTP against FTP alone in seconds.

Figure 3: Evaluation of the BitDew overhead when transferring files.

protocol run together at the same time. As clusters have limited number of nodes and to provoke a greater impact of the BitDew protocol, we setup the experiment so that the D* services, the FTP server and the BitTorrent seeder run on the same node. To generate a maximum of BitDew traffic, we have configured the DT heartbeat to monitor transfer every 500 ms and the DS service to synchronize with the scheduler every second.

The benchmark works as follows: BitDew replicates data, whose size varies from 10 to 500MB, to a set of nodes whose size ranges from 10 to 250. Two file transfer protocols are measured: FTP and BitTorrent.

Figure 3a presents the completion time in seconds for the file distribution, that is the time between the beginning of the file replication and the last node ending the file transfer, averaged over 30 experiments. The first result shows that BitTorrent clearly outperforms FTP when the file size is greater than 20MB and when the number of nodes is greater than 10, providing more scalability when the number of nodes is large.

As expected measurement of BitDew running BitTorrent and FTP are similar to our previous study where FTP and BitTorrent alone were compared. To investigate precisely the impact of the

out-of-band transfer management of BitDew on file transfer performances, we choose to compare the performances of file transfer performed by the FTP protocol alone against BitDew and FTP.

We choose not to evaluate the BitDew overhead over BitTorrent for the following reasons : *i*) we have shown in [41] that BitTorrent exhibits varying and unpredictable performances, which would have affected the quality of the measures, *ii*) we want to generate high load on the server and it is well known that BitTorrent can adapt to low bandwidth servers. Also, one can argue that there exists efficient and scalable FTP server such as GridFTP server. However, for this experiment, the main bottleneck is the server bandwidth, so the efficiency of the FTP server does not affect the measure of the BitDew overhead.

Figure 3b shows the overhead of BitDew with FTP against FTP alone in percentage of the file transfer time. One can observe that the impact is stronger on small files distributed to a small number of nodes. Before, launching a data transfer, BitDew has to communicate with the DC service to obtain a location of the data, to the DR service to obtain a description of the protocol, and finally to the DR service to register the data transfer. Obviously these steps add extra latency. Figure 3c shows the overhead of BitDew with FTP against FTP alone in seconds. BitDew overhead increases with the size of the file and with the number of nodes downloading the file, which shows that the overhead is mainly due to the bandwidth consumed by the Bitdew protocol. For instance, distribution of a 500 MB file to 250 nodes, in approximately 1000 sec, generates at least 500000 requests to the DT service. Still, our experimental settings are stressful compared to real world settings. For instance the BOINC client contacts the server only if any of the following occurs: when the user's specified period is reached, whose default is 8.5 hours or when a work unit deadline is approaching and the working unit is finished. By analyzing the logs of the BOINC based XtremLab[5] project, we have found that, after filtering out clients which contact the server less than two times in a 24 hours period, the mean time between two requests is 2.4 hours. Thus, even a very responsive periodical heartbeat of 1 minute generate an equivalent workload on the DT service if the number of clients exceeds 30000, implying a reasonable degradation on file transfer performance less than 10%.

4.4 Fault tolerance data operation

The next experiment aims at illustrating a fault tolerance scenario. We run the experiment in the DSL-Lab environment. The scenario consists of the following : we create a datum with the following attribute : replica = 5, fault tolerance = true and protocol = "ftp", which means that the runtime will constantly tries to maintain the number of data replica, even in the presence of host failures. At the beginning of the experiment, the data are owned by 5 nodes. Every 20 seconds, we simulate a machine crash by killing the BitDew process on one machine owning the data, and we simultaneously simulate a new host arrival by starting BitDew on an other node.

We measure the elapsed time between the arrival of the node and the schedule of data to this node, as well as the time to download the data. Figure 4 shows the Gantt chart of the experiment and the bandwidth obtained during the download. One can observe a waiting time of 3 seconds before the download starts, which is due to the failure detector. BitDew maintains a timeout to detect host failure, which is set to 3 times of the heartbeat period (here 1 second). We can also observe a great variation in the communication performance between the hosts. This can be explained by the difference of service quality between the various Internet Service Providers and by the fact that bandwidth consuming applications might be running at the same time.

[5] `http://xtremlab.lri.fr`

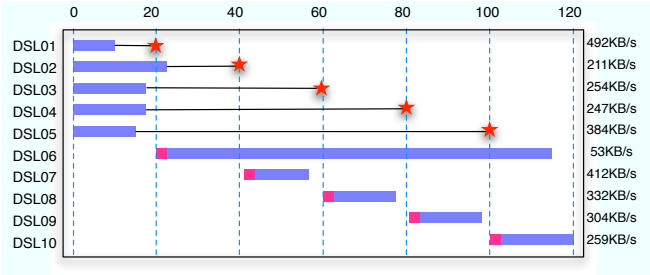


Figure 4: Evaluation of Bitdew in a faulty scenario. The Gantt chart presents the main events: red box the waiting time, blue box the downloading time and red star indicates a node crash. The rightmost part of the graph presents the bandwidth obtained during the file transfer.

5 Programing a Master/worker Application

In this section, we present an example of a Master/worker application developped with BitDew. The application is based on NCBI BLAST (Basic Local Alignment Search Tool), BLAST compares a query sequence with a database of sequences, and identifies library sequences that resemble the query sequence above a certain threshold. In our experiments, we used the *blastn* program that compares an amino acid query sequence against a protein sequence database. The input DNA sequences used were taken from the GeneBank database and the DNA databases were taken from the National Center for Biotechnology Information.

```
attribute Application = { replication = -1, protocol = "BitTorrent"
}
attribute Genebase = { protocol = "BitTorrent", lifetime = Collector , affinity =
    Sequence
}
attribute Sequence = { fault tolerance = true, protocol = "http", lifetime =
    Collector , replication = x
}
attribute Result = { protocol = "http", affinity = Collector , lifetime = Collector
}
Collector attribute {
}
```

Listing 3: Attributes Definition

In a classical MW application, tasks are created by the master and scheduled to the workers. Once a task is scheduled, the worker has to download the data needed before the task is executed. In contrast, the data-driven approach followed by BitDew implies that data are first scheduled to hosts. The programmer do not have to code explicitly the data movement from host to host, neither to manage fault tolerance. Programming the master or the worker consists in operating on data and attributes and reacting on data copy.

With this application, there exists three sets of data : the *Application* file, the *Genebase* file, and the *Sequence* files. The *Application* file is a binary executable files which has to be deployed on each node of the network. The replication attribute is set to -1, which is a special value which indicates that the data will be transferred to every node in the network. Although the size is small (4.45 MB), the file is highly shared, so it is worth setting the protocol to BitTorrent.

Each task depends on two data: the *Genebase* data is a compressed large archive (2.68 GB), and the *Sequence* which is the parameter of the task. The previous experience has shown that FTP is an appropriate protocol to distributes sequence which are small text files, unique

to each tasks, and BitTorrent is efficient to distribute *Genebase* shared by all the computing nodes. We define an affinity between a *Sequence* and a *Genebase*, which means that BitDew will automatically schedule transfer of *Genebase* data wherever a *Sequence* is present. This ensures that only nodes actually participating in the computation will download and store the *Genebase* files. Once the *Genebase*, the *Application* and at least one *Sequence* files are present in the worker's local cache, the worker can launch BLAST computation.

At the end of the computation, the tasks will produce a *Result* file which has to be retrieved by the master node. The master creates an empty *Collector* and `pin` this data. Each worker set an affinity attribute from *Result* data to the *Collector* data. By this way, results will get automatically transferred to the master node. At the end of the experiment, it is wise to delete data and to purge the workers' local cache. However, some files are large and should be kept persistent on workers' cache for the next execution. An elegant way is to set for every data a relative lifetime to the *Collector*. Once the user decides that he has finished his work, he can safely delete the *Collector*, which will obsolete remaining data.

Setting the *fault tolerance* attribute for the sequence data ensures that the tasks will be rescheduled if the host failed. The replication attribute of the sequence also affects the scheduling of data and tasks on the hosts. For instance, to implement the scheduling strategy presented in [26], one would simply keep a replication to 1 when the number of tasks is less than the number of hosts and dynamically increase the value of *replication* attribute when there are more hosts available than remaining tasks.

In Figure 5, we use the protocols FTP and BitTorrent respectively as transfer protocol. The x axis represents the number of workers used in our experiment, the y axis represents the total execution time: the time to broadcast the Genebase and Sequence to query, more the execution time of BLAST application for searching gene sequence in Genebase. When the number of workers is relatively small (10 and 20), the performance of BitTorrent is worse then FTP. But when the number of workers still increases from 50 to 250, the total time of FTP increases considerably, in contrast the line for BitTorrent is nearly flat.

For further experiments, we run our M/W application with BitDew on a part of Grid5000: 400 nodes of 4 clusters in 3 different sites (see 1). Breakdown of total execution time, in transfer time, unzip time, execution time is shown in Figure 6. The last 2 columns show the mean time for 4 clusters. Obviously, the transfer protocols used by BitDew play an important role over application performance because most of the time is spent for transferring data in network. In this case, using BitTorrent protocol to transfer data can gain almost a factor 10 of time for delivering computing data.

6 Related Work

The main efforts to enable data-intensive application on the Grid were initiated by projects that address the issues of data movement, data access and metadata management on the Grid. Representative example includes GridFTP [3] and GFarm [39]. GridFTP is a protocol to support secure, reliable and high performance wide-area data transfers on the Grid, by implementing striped transfers from multiple sources, parallel transfers, tuned TCP usage, failure detection and GSS authentication. The GFarm file system enable parallel processing of data intensive application. OGSA-DAI [7] is an effort to provides middleware for data access and data integration in the Grid. Metadata management is one of the key technique in Data Grids [37]. Metadata Catalog Service provides mechanism for storing and accessing descriptive metadata and allows users to query for data items based on desired attributes [13]. To provide high availability and scalability, metadata replica can be organized in a highly connected graph [23] or distributed in

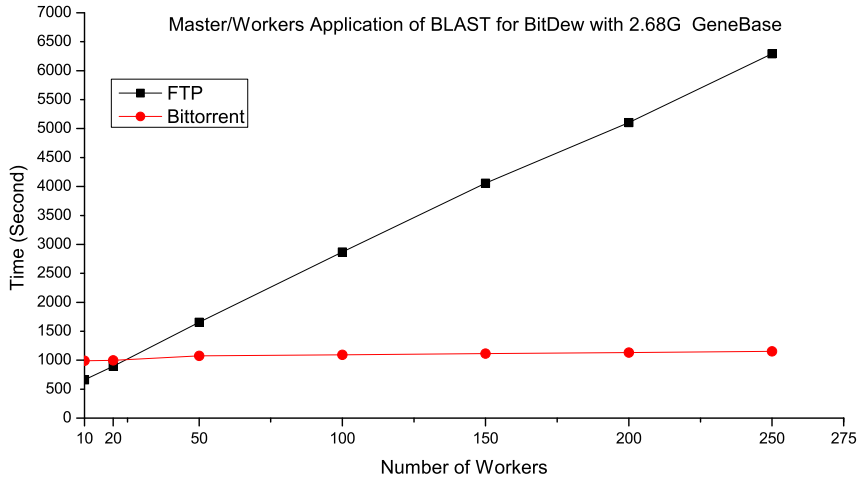


Figure 5: BitDew performances on a Master/Worker application. The two lines present the average total execution time in seconds for the BLAST application with a large Genebase of 2.68GB, executed on 10 to 250 nodes and with file transfer performed by FTP and BitTorrent

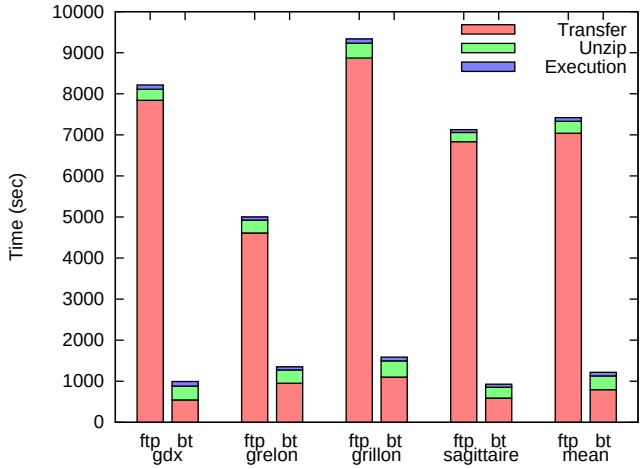


Figure 6: Breakdown of total execution time in time to transfer data, time to unzip data and Blast execution time by cluster. The rightmost values is the time average on the whole platform.

P2P network [33]. Beside descriptive information, metadata in BitDew also expresses *directive* information to drive the runtime environment. However Desktop Grid differs significantly from traditional Grid in terms of security, volatility and system size. Therefore specific mechanisms must be used to efficiently distribute large files and exploit local resource storage.

Several systems have been proposed to aggregate unused desktop storage of workstation within a LAN. Farsite [1] builds a virtual centralized file system over a set of untrusted desktop computers. It provides file reliability and availability through cryptography, replication and file

caching. Freeloader [40] fulfills similar goals but unifies data storage as a unique scratch/cache space for hosting immutable datasets and exploiting data locality. Nevertheless these projects offer a file system semantic for accessing data that is not precise enough to give users (or an agent that work on behalf of the user) control over data placement, replication and fault tolerance. We emphasize that BitDew provides abstractions and mechanisms for file distribution that work on a layer above these systems, but can nevertheless work in cooperation with them.

Oceanstore [27], IBP [9], and Eternity [5] aggregate a network of untrusted servers to provide global and persistent storage. IBP's strength relies on a comprehensive API to remotely store and move data from a set of well-defined servers. Using an IBP "storage in the network" service helps to build more efficient and reliable distributed application. Eternity and Oceanstore uses cryptographic and redundancy technologies to provide deep storage for archival purpose from unreliable and untrusted servers. In contrast with these projects, BitDew specifically adds wide-area resource storage scavenging, using P2P technics to face scalability and reliability issue.

JuxMem [8] is a large scale data sharing service for the Grid which relies on a P2P infrastructure to provide a DSM-like view of the data space. It is built over the JXTA middleware and features data replication, localization, fault tolerance, and a specialized consistency model. Bitdew differs in its approach to build a flexible and modular runtime environment which allows one to integrate a new protocol for data transfer, for DHT's, or to access remote storage. We believe that a key requirement for a Grid data sharing service is its ability to integrate with other Grid standards and utilities.

As BitDew's data distribution mechanism is built by using BitTorrent protocol, one could argue that performance over a wide-area could be severely limited by the transmission of redundant data blocks over bottleneck links. However, recent techniques involving network coding and file swarming have alleviated these concerns. The Avalanche protocol [19, 18], which effectively ensures that only unique and required datum is transmitted through these links, could be easily integrated within BitDew's framework.

7 Conclusion

We have presented BitDew, a programmable environment for large-scale data management and distribution that bridges the gap between Desktop Grid and P2P data sharing systems. We have detailed a programming model that provides developers with an abstraction for complex tasks associated with large scale data management, such as life cycle, transfer, placement, replication and fault tolerance. While maintaining a high level transparency, users still have the possibility to enhance and fine tune this platform by interacting directly with the runtime environment.

BitDew's runtime environment is an evolution of traditional Desktop Grid architectures which takes advantage of local storage of the resources. We have proposed a flexible and modular environment which relies on a core set of independent services to catalog, store, transfer and schedule data. This runtime environment has been designed to cope with a large number of volatile resources, and it has the following high-level features: reliable data transfers, automatic replication and transparent data placement. To achieve scalability, the BitDew architecture can apply P2P protocols when a centralized approach might induce a performance bottleneck. We have conducted a performance evaluation of a distributed data catalog implemented with the DKS DHT and evaluate BitDew's protocol overhead over the FTP file data transfer. We have presented a master/worker application with a performance evaluation which show the potential by relying on an efficient data distribution subsystem.

Desktop grids can integrate BitDew in three complementary ways. First, BitDew can serve as a multi-protocol file transfer library, featuring concurrent, reliable and P2P transfers. BitDew

would be a means of leveraging future enhancements of P2P protocols without modifying the Desktop Grid system. Second, a Desktop Grid could be enhanced with a distributed storage service based on BitDew, which would allow data management tasks (for example, lifetime and replication management) that are currently impossible to perform on existing DG systems. Finally, BitDew could facilitate the execution of data-intensive applications. This is the subject of our future works, which will aim at building a Data Desktop Grids system, providing the following features: sliced data, collective communication such as gather/ scatter, and other programming abstractions, such as support for distributed MapReduce operations.

Acknowledgment

Authors would to thank Derrick Kondo for his insightful comments and correction throughout our research and writing of this report.

Experiments presented in this paper 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 `https://www.grid5000.fr`).

Experiments presented in this paper were carried out using the DSL-Lab experimental testbed, an initiative supported by the French ANR JCJC program (see `https://www.dsllab.org`) under grant JC05_55975.

References

[1] A. Adya and all. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, 2002.

[2] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications. In *The 3rd International CGP2P Workshop*, Tokyo, 2003.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of Super Computing (SC05)*, 2005.

[4] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *proceedings of the 5th IEEE/ACM International GRID Workshop*, Pittsburgh, USA, 2004.

[5] R. Anderson. The Eternity Service. In *Proceedings of Pragocrypt '96*, 1996.

[6] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[7] M. Antonioletti and all. The Design and Implementation of Grid Database Services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17:357–376, February 2005.

[8] G. Antoniu, L. Bougé, and M. Jan. JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[9] A. Bassi, M. Beck, G. Fagg, T. Moore, J. S. Plank, M. Swany, and R. Wolski. The Internet BackPlane Protocol: A Study in Resource Sharing. In *Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002.

[10] R. Bolze and all. Grid5000: A Large Scale Highly Reconfigurable Experimental Grid Testbed. *International Journal on High Peerformance Computing and Applications*, 2006.

[11] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A Peer-to-Peer Enhancement for the Network File System. In *Proceeding of International Symposium on SuperComputing SC'04*, 2004.

[12] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, 2003.

[13] E. Deelman, G. Singh, M. P. Atkinson, A. Chervenak, N. P. C. Hong, C. Kesselman, S. Patil, L. Pearlman, and M.-H. Su. Grid-Based Metadata Services. In *SSDBM04*, Santorini, Greece, June 2004.

[14] Enabling Grids for E-Science in Europe.

[15] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing Platform. In *CCGRID'2001 Special Session Global Computing on Personal Devices*, 2001.

[16] Y. Fernandess and D. Malkhi. On Collaborative Content Distribution using Multi-Message Gossip. In *Proceeding of IEEE IPDPS*, Rhodes Island, 2006.

[17] D. Gelernter. Generative Communications in Linda. *ACM Transactions on Programming Languages and Systems*, 1985.

[18] C. Gkantsidis, J. Miller, and P. Rodriguez. Anatomy of a P2P Content Distribution System with Network Coding. In *IPTPS'06*, California, U.S.A., 2006.

[19] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive View of a Live Network Coding P2P System. In *ACM SIGCOMM/USENIX IMC'06*, Brazil, 2006.

[20] C. Gkantsidis and P. Rodriguez. Network Coding for Large Scale Content Distribution. In *Proceedings of IEEE/INFOCOM 2005*, Miami, USA, March 2005.

[21] A. Iamnitchi, S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *proceeding of 15th IEEE International Symposium on High Performance Distributed Computing HPDC 15*, Paris, 2006.

[22] A. Iamnitchi, S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *HPDC 2006*, Paris, 2006.

[23] H. Jin, M. Xiong, S. Wu, and D. Zou. Replica Based Distributed Metadata Management in Grid Environment. *Computational Science - Lecture Notes in Computer Science, Springer-Verlag*, 3994:1055–1062, 2006.

[24] K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *5th International Workshop in Grid Computing (Grid 2004)*, Pittsburgh, 2004.

[25] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello. Characterizing Result Errors in Internet Desktop Grids. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2007.

[26] D. Kondo, A. Chien, and H. Casanova. Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. In *ACM Conference on High Performance Computing and Networking (SC'04)*, Pittsburgh, 2004.

[27] J. Kubiatowicz and all. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[28] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 104–111, Washington, DC, 1988. IEEE Computer Society.

[29] J. Luna, M. Flouris, M. Marazakis, and A. Bilas. Providing security to the Desktop Data Grid. In *2nd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid'08)*, 2008.

[30] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*. MIT, 2002.

[31] E. Otoo, D. Rotem, and A. Romosan. Optimal File-Bundle Caching Algorithms for Data-Grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2004. IEEE Computer Society.

[32] PPDG. From Fabric to Physics. Technical report, The Particle Physics Data Grid, 2006.

[33] A. Reinefeld, F. Schintke, and T. Schatt. Scalable and Self-Optimizing Data Grids. *Annual Review of Scalable Computing, Singapore University Press*, 6:30–60, 2004.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 2001.

[35] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[36] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[37] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of SuperComputing'03*, Phoenix, Arizona, USA, November 2003.

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[39] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proc. of the 2nd IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid'02)*, 2002.

[40] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. L. Scott. FreeLoader: Scavenging Desktop Storage Resources for Scientific Data. In *Proceedings of Supercomputing 2005 (SC'05)*, Seattle, 2005.

[41] B. Wei, G. Fedak, and F. Cappello. Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005*, Seatle, 2005.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399