TÉCNICO LISBOA

freeCycles - Efficient Data Distribution for Volunteer Computing

Rodrigo Fraga Barcelos Paulus Bruno

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. Paulo Jorge Pires Ferreira

Examination Committee

Chairperson:	Prof. Dr. Joaquim Armando Pires Jorge
Supervisor:	Prof. Dr. Paulo Jorge Pires Ferreira
Member of the Committee:	Prof. Dr. David Manuel Martins de Matos

September 2014

Acknowledgments

I leave here my deepest gratitude for all the time, patience, dedication, support, and guidance to Professor Paulo Ferreira, my supervisor.

I would also like to thank Professor David Matos, who was a member of the project course evaluation committee, presented interesting questions about the project and gave me a detailed review of the first chapters of this document.

Another big acknowledgment to the RNL (Rede das Novas Licenciaturas) administration team. The infrastructure provided by the RNL was fundamental for the performance evaluation.

I leave a final special acknowledgement to Cláudia Ávila, who was always very supportive and helpful during the whole process.

Resumo

A computação voluntária (VC) é uma forma de reutilizar recursos computacionais como poder computacional, largura de banda e espaço de armazenamento que estão, grande parte do tempo, subaproveitados. Além disso, desenvolvimentos recentes na área dos paradigmas de programação, mais especificamente em MapReduce, revelam o interesse em correr aplicações MapReduce em redes de grande escala, como a Internet. No entanto, as atuais técnicas de distribuição de dados usadas em computação voluntária, para distribuir grandes quantidades de informação (que são necessárias para as aplicações de MapReduce), estão subotimizadas e necessitam de ser repensadas.

Assim sendo, foi desenvolvido o freeCycles, uma solução de VC que suporta aplicações MapReduce e oferece duas principais contribuições: i) distribuição de dados mais eficiente (entre o servidor de dados, mappers e reducers) através da utilização do protocolo BitTorrent (que é utilizado para distribuir os dados iniciais, intermédios e resultados finais); ii) disponibilidade dos dados intermédios melhorada através de replicação de computação e de dados que previne atrasos na execução das aplicações MapReduce.

Neste trabalho são apresentados o desenho e implementação da nossa solução, freeCycles, assim como uma extensa avaliação que confirma a utilidade das contribuições acima mencionadas (distribuição de dados e disponibilidade dos dados intermédios melhoradas). Os resultados da avaliação confirmam ainda verosimilidade da VC para executar aplicações MapReduce.

Palavras-chave: Computação Voluntária, Computação em Rede, BitTorrent, Distribuição de Dados, Sistemas Entre-Pares e Redes Sobrepostas

Abstract

Volunteer Computing (VC) is a very interesting solution to harness large amounts of computational power, network bandwidth, and storage which, otherwise, would be left with no use. In addition, recent developments of new programming paradigms, namely MapReduce, have raised the interest of using VC solutions to run MapReduce applications on the large scale Internet. However, current data distribution techniques, used in VC to distribute the high volumes of information which are needed to run MapReduce jobs, are naive, and therefore need to be re-thought.

Thus, we present a VC solution called freeCycles, that supports MapReduce jobs and provides two new main contributions: i) improves data distribution (among the data server, mappers, and reducers) by using the BitTorrent protocol to distribute (input, intermediate and output) data, and ii) improves intermediate data availability by replicating tasks or data to volunteers in order to avoid losing intermediate data and consequently preventing big delays on the MapReduce overall execution time.

We present the design and implementation of freeCycles along with an extensive set of performance results which confirm the usefulness of the above mentioned contributions, improved data distribution and availability, thus making VC a feasible approach to run MapReduce jobs.

Keywords: Volunteer Computing, Desktop Grids, Availability, BitTorrent, Data Distribution, Peer-to-Peer.

Contents

List of Tables

List of Figures

Glossary

API	Application Programming Interface
BOINC	Berkeley Open Infrastructure for Network Computing
BT	BitTorrent
CCOF	Cluster Computing On the Fly
CDF	Comulative Distribution Function
CDN	Content Distribution Network
CPU	Central Processing Unit
FLOPS	FLoating-point Operation Per Second
FTP	File Transfer Protocol
GB	Gigabyte
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
INF	Inifinite
MBps	Megabyte per second
MB	Megabyte
MOON	MapReduce On Opportunistic Environments
Mbps	Megabit per second
Mb	Megabit
NAT	Network Address Translation
NA	Not Applicable
P2P	Peer-to-Peer
PC	Personal Computer
RAM	Random Access Memory
SCOLARS	Scalable COmplex LARge Scale Volunteer Computing
TCP	Transmission Control Protocol
UML	Unified Modeling Language

VC	Volunteer Computing
VM	Virtual Machine
VP	Volunteer Pool

Chapter 1

Introduction

With the ever growing demand for computational power, scientists and companies all over the world strive to harvest more computational resources in order to solve more and bigger problems in less time, while spending the minimum money possible. With these two objectives in mind, we think of volunteer computing (VC) as a viable solution to access huge amounts of computational resources, namely CPU cycles, network bandwidth, and storage, while reducing the cost of the computation (although some commercial solutions give incentives, pure VC projects can produce computation at zero cost).

With time, more computing devices (PCs, gaming consoles, tablets, mobile phones, and any other kind of device capable of sharing its resources) join the network. By aggregating all these resources in a global volunteer pool, it is possible to obtain huge amounts of computational resources that would be impossible, or impractical, for most grids, supercomputers, and clusters. For example, recent data from BOINC [4, 5] projects shows that, currently, there are 70 supported projects sustained by an average computational power of over 7 PetaFLOPS[1].

The creation and development of large scale VC systems enables large projects, that could not be run on grids, supercomputers, or clusters due to its size and/or costs associated, to run with minimized costs. VC also provides a platform to explore and create new projects without losing time building and setting up execution environments like clusters.

In addition, recent developments of new programming paradigms, namely MapReduce[2] [15], have raised the interest of using VC solutions to run MapReduce applications on large scale networks, such as the Internet. Although increasing the attractiveness of VC, it also brings the need to rethink and evolve current VC platforms' architectures and protocols (in particular, the data distribution techniques and protocols) to adapt to these new programming paradigms.

We present a system called freeCycles, a VC solution which has as its ultimate goal to aggregate as many volunteer resources as possible in order to run MapReduce jobs in a scalable and efficient way. Therefore, the output of this project is a middleware platform that enables upper software abstraction layers (programs developed by scientists, for example) to use volunteer resources all over the Internet

[1] Statistics from boincstats.com

[2] MapReduce is a popular programming paradigm composed of two operations: Map (applied for a range of values) and Reduce (operation that will aggregate values generated by the Map operation).

as they would use a supercomputer or a cluster. Therefore, using freeCycles, applications can be developed to solve problems that require more computational resources than what it is available in private clusters, grids or even in supercomputers.

To be successful, freeCycles must fulfill several key objectives. It must be capable of scaling up with the number of volunteers; it must be capable of collecting volunteers' resources such as CPU cycles, network bandwidth, and storage in an efficient way; it must be able to tolerate byzantine[3] fail-stop failures (which can be motivated by unreliable network connections, that are very common in large scale networks such as the Internet). Finally, were quire our solution to be capable of supporting new parallel programming paradigms, in particular MapReduce, given its relevance for a large number of applications.

MapReduce applications commonly have two key features: i) they depend on large amounts of information to run (for both input and output) and ii) they may run for several cycles (as the original algorithm for page ranking [6]). Therefore, in order to take full advantage of the MapReduce paradigm, freeCycles must be capable of distributing large amounts of data (namely input, intermediate, and output data) very efficiently while allowing applications to perform several MapReduce cycles without compromising its scalability.

In order to understand the challenges inherent to building a solution like freeCycles, it is important to note the difference between the main three computing environments: clusters, grids, and volunteer pools. Clusters are composed by dedicated computers so that every node has a fast connection with other nodes, node failure is low, nodes are very similar in hardware and software and all their resources are focused on cluster jobs. The second computing environment is a grid. Grids may be created by aggregating desktop computers from universities, research labs or even companies. Computers have moderate to fast connection with each other and grids may be inter-connected to create larger grids; node failure is moderate, nodes are also similar in hardware and software but their resources are shared between user tasks and grid jobs. At the end of the spectrum there are volunteer pools. Volunteer pools are made of regular desktop computers, owned by volunteers around the world. Volunteer nodes have variable Internet connection bandwidth, node churn is very high (since computers are not managed by any central entity), nodes are very asymmetrical in hardware and software and their computing resources are mainly focused on user tasks. Finally, as opposed to grids and clusters, volunteer computers cannot be trusted since they may be managed by malicious users.

Besides running on volunteer pools, freeCycles is also required to support MapReduce, a popular data-intensive programming paradigm, that was initially created for clusters and that depends on large amounts of data to run. Therefore, feeding large inputs to all volunteer nodes through unstable Internet connections, and coping with problems such as node churn, variable node performance and connection, and being able to do it in a scalable way is the challenge that address with freeCycles.

By considering all the available solutions, it is important to note that solutions based on clusters and/or grids do not fit our needs. Such solutions are designed for controlled environments where node

[3] A byzantine failure is an arbitrary fault that occurs during the execution of an algorithm in a distributed system. It may be caused, for example, by malicious users or some signal interference within the communication channel or even inside the computer components.

churn is expected to be low, where nodes are typically well connected with each other, nodes can be trusted and are very similar in terms of software and hardware. Therefore, solutions like HTCondor [40], Hadoop [44], XtremWeb [16] and other similar computing platforms are of no use to attain our goals.

When we turn to VC platforms, we observe that most existing solutions are built and optimized to run Bag-of-Tasks applications. Therefore, solutions such as BOINC, GridBot [37], Bayanihan [34], CCOF [26], and many others [3, 28, 45, 7, 10, 30, 29, 36] do not support the execution of MapReduce jobs, which is one of our main requirements.

With respect to the few solutions [25, 39, 27, 14] that support MapReduce, we are able to point out some frequent problems. First, data distribution (input, intermediate, and final output) is done employing naive techniques (that use point to point protocols such as FTP or HTTP). Current solutions do not take advantage of task replication (that is essential to avoid stragglers and to tolerate byzantine faults). Thus, even thought multiple mappers have the same output, reducers still download intermediate data from one mapper only. The second problem is related to intermediate data availability. It was shown [22] that if some mappers fail and intermediate data becomes unavailable, a MapReduce job can be delayed by 30% of the overall computing time. Current solutions fail to prevent intermediate data from becoming unavailable. The third problem comes from the lack of support for applications with more than one MapReduce cycle. Current solutions typically use a central data server to store all input and output data. Therefore, this central server is a bottleneck between every cycle (since the previous cycle needs to upload all output data and the next cycle needs to download all the input data before starting).

In order to support the execution of programming paradigms like MapReduce, that need large amounts of data to process, new distribution techniques need to be employed. In conclusion, current VC solutions raise several issues that need to be improved in order to be able to run efficiently new parallel programming paradigms, MapReduce in particular, on large volunteer pools.

freeCycles is a BOINC compatible VC platform that enables the deployment of MapReduce applications. Therefore, freeCycles provides the same features as BOINC regarding scheduling, tasks replication and verification. Besides supporting MapReduce jobs, freeCycles goes one step further by allowing volunteers (mappers or reducers) to help distributing both the input, intermediate output and final output data (through the BitTorrent[4] protocol and not point-to-point). Since multiple clients will need the same input and will produce the same output (for task replication purposes), multiple clients can send in parallel multiple parts of the same file to other clients or to the data server. freeCycles will, therefore, benefit from clients' network bandwidth to distribute data to other clients or even to the central server. As a consequence of distributing all data through the BitTorrent protocol, freeCycles allows sequences of MapReduce cycles to run without having to wait for the data server, i.e., data can flow directly from reducers to mappers (of the next cycle). Regarding the availability of intermediate data, freeCycles proposes an enhanced work scheduler that automatically chooses between replicating data or tasks in order to minimize the risk of loosing intermediate data.

By providing these features, freeCycles achieves higher scalability (reducing the burden on the data server), reduced transfer time (improving the overall turn-around time), and augmented fault tolerance

[4]The Official BitTorrent specification can be found at www.bittorrent.org

(since nodes can fail during the transfer without compromising the data transfer).

In sum, this work provides the following contributions: i) a BOINC compatible VC platform that uses the BitTorrent protocol to enhance data distribution in the context of MapReduce jobs; ii) a study on the performance and scalability of task and data replication in VC computing (from the data transfer perspective); iii) extensive performance comparison of freeCycles with other similar platforms, thus proving its benefits; iv) a research paper entitled *freeCycles: Efficient Data Distribution for Volunteer Computing* published in CloudDP'14 Proceeding of the Fourth International Workshop on Cloud Data and Platforms.

The remainder of this document is organized as follows: Chapter 2 is dedicated to state of the art, where we discuss several aspects of VC and analyse some data distribution systems and some VC platforms; Chapter 3 describes freeCycle's architecture; in Chapter 4, we address our system from the implementation point of view; Chapter 5 contains an extensive evaluation of our data distribution architecture and final solution. To finish the document, some conclusions and future work is present on the final chapter, Chapter 6.

Chapter 2

Related Work

By definition [33], Volunteer Computing or Cycle Sharing is a type of volunteer computing in which computer owners (known as volunteers) donate their computer resources such as processing power, storage capacity, or network bandwidth.

2.1 Why Volunteer Computing?

Notwithstanding, one might question why or when to use VC, to produce some computation, instead of using more traditional approaches: a cluster, a grid, or even a supercomputer.

To answer the question of whether or not VC is the best approach to place some computation, we would need to analyse several factors that depend not only on the computation itself but also on its stakeholders (the scientist, for example):

- Is the computation data intensive? In other words, is the time needed to send input data significantly less than the time it takes to process it? This is one of the most important restrictions of VC, the data/compute ratio must be low (less than 1GB per day of computing according to David Anderson[1]). As opposed to grids, clusters or supercomputers, volunteer computers have very limited Internet connection. Therefore, sending input data to volunteers spread around the globe will take much longer than sending data to very well connected nodes, as one might find in clusters or grids. In fact, the number of tasks running simultaneously is limited by the time it takes to process some input data, divided by the time needed to transfer the input data;

- How often do tasks need to communicate? As most VC platforms deliver tasks when volunteers ask for work (a pull model), having complex communication patterns between worker nodes might become prohibitive. This is not a problem for clusters or grids as long as a push model can be used (where work is pushed to workers). By using a push model, it is possible to assure that dependent tasks will run at the same time. However, when using a pull model, tasks might be

[1] David P. Anderson was the creator of SETI@home and now is the leader of the BOINC [4] project. His web page is available at: http://boinc.berkeley.edu/anderson. Presentation slides from one of his lectures where he talks about data/compute ratio can be found at http://gcl.cis.udel.edu/projects/msm/data/msm_02_03DavidAnderson.pdf.

delivered long after the previous ones and therefore it is very difficult to know if dependent tasks will run simultaneously;

- Is the data to process confidential? As VC will deliver tasks to remote volunteer computers, owned by anonymous volunteers, data (that might be confidential) will be disclosed. Although there are some algorithms that can be applied to ciphered data [1], most types of computation are still difficult to express using only ciphered data. Since clusters, grids, and supercomputers can have restricted access, this problem does no apply;

- Does the scientist/project owner has access or money to pay for a cluster, grid, or supercomputer? Most scientists might not have access or have very limited access to these resources. Moreover, most cloud providers will become very costly for running CPU intensive applications. This problem does not exist in VC, where computer owners donate their resources;

- How easy is to attract new volunteers? When using VC, it is important to be able to attract the public's attention so that more volunteers will join the computation. This is, perhaps, one of the most challenging issues in VC, how to bring more volunteer resources to the computation. Hence, when going for VC, it is very important to have a project able to motivate and draw the public's attention;

- How big is the computation? Another important factor when deciding where to deploy an application is the size of the computation. In our opinion, VC is most appropriate when some computation needs thousands or even hundreds of thousands of machines, i.e., for large computations. Running such computations with fewer machines would take years or even decades to finish the computation.

One important use of Volunteer Computing (VC) is to perform large-scale low-cost computation in the context of academic or research projects. Scientific fields that benefit from VC platforms are (for example): biology, climate study, epidemiology, physics, mathematics, and many more. According to BOINC statistics[2], there are more than seventy active projects using BOINC software with a total of 14.000.000 hosts (total number of hosts in 60 days) with an aggregated computational power of 7.940,256 TeraFLOPS (average floating point operations per second).

Over time, many volunteer computing platforms have been developed. Although all these systems have the same goal (to harvest volunteer resources) many solutions differ in several aspects such as: network architecture, task validation, data distribution, supported paradigms, security, or fault tolerance.

Given that our work is based on improving data distribution on MapReduce [15] jobs within a volunteer pool, we dedicate this section to studying the problem of data distribution in such volatile environments. We start by discussing possible data distribution architectures; then, we address the availability issues of intermediate data followed with a description and analysis of some of the most relevant data distribution systems; finally, we conclude by presenting some relevant VC platforms. This section finishes with a taxonomy summary of the analyzed systems.

[2] http://boincstats.com/

2.2 Data Distribution Architectures on Unreliable Environments

The data distribution problem refers to the issue of how to distribute data (input, intermediate, and output) in a volatile environment such as a volunteer pool. Current solutions are typically focused on Bag-of-Tasks and thus are only concerned about input and output data. However, when recent programming paradigms (that process large amounts of information) are to be used, MapReduce in particular, care must be taken in order to provide proper distribution of data.

Data distribution techniques can be divided into centralized and non-centralized (peer-to-peer), each of them with their advantages and disadvantages.

2.2.1 Centralized

Centralized data distribution is mainly used by systems that employ a centralized job management. Systems like BOINC [4], XtremWeb [16] and many more use centralized servers where all data is stored. When tasks are assigned to clients, clients can download the needed data from the centralized server. Output and/or intermediate data is also uploaded to the centralized servers.

The use of centralized servers provides several advantages compared to peer-to-peer techniques: better data management, better security against data tempering and better accounting. However, it has severe disadvantages (compared to peer-to-peer techniques): high burden on the centralized servers that could lead to high latency or high download/upload times, and having only one point of failure that could prevent clients from contributing.

2.2.2 Peer-to-Peer

Peer-to-Peer data distribution techniques implies that all (or almost all) the nodes play the client and server roles, i.e., uploading and downloading data to and from other nodes [35]. By distributing data with peer-to-peer techniques, one can send data to multiple nodes and receive data from multiple nodes. It is mainly used by systems where there is no centralized management (systems like MapReduce for Dynamic Environments [27] and P3 [36]). Thus, there is no central server where all data is stored; instead, data is stored on multiple nodes.

Using a distributed data distribution, two main difficulties arise: data availability and reliability. The first comes from the fact that volatile nodes can come and go (node churn). This means that data may be lost if no more copies were made (i.e. using replication). The second problem is reliability, i.e. data provided by nodes cannot be trusted (since malicious users may have changed it).

Despite all the problems that peer-to-peer networks may present, it is still a way to avoid centralization. Therefore, it is possible to scale up to millions of nodes and to provide very high download rates by using parallel connections to multiple nodes.

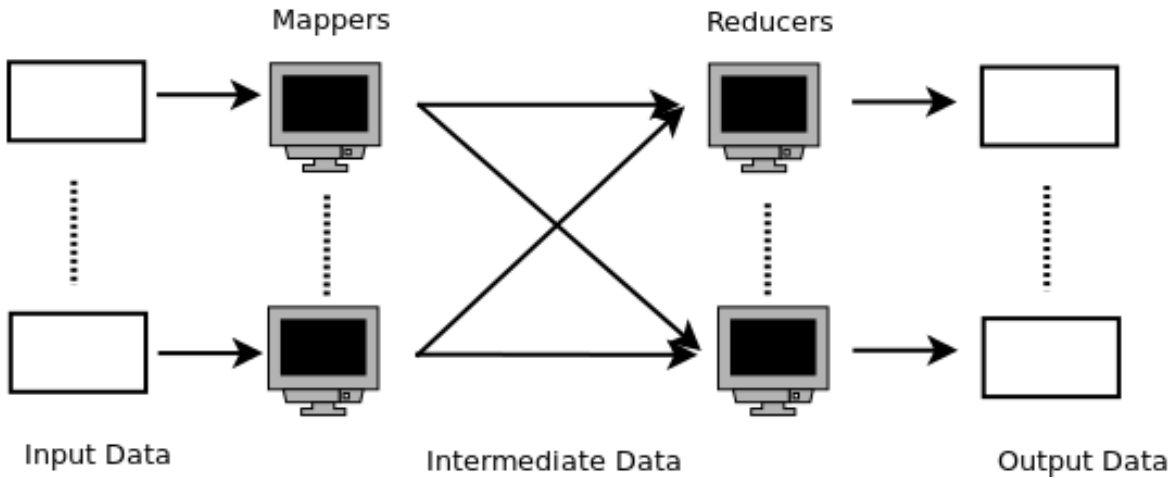


Figure 2.1: MapReduce Workflow

2.3 Availability of Intermediate Data

New emergent paradigms such as MapReduce, Dryad [21] and Pig [18] are gaining significant popularity for large-scale parallel processing. Namely, organizations such as Facebook, Yahoo!, and even Microsoft are intensively using them for data processing workflows.

The execution of dataflow programs (programs where data flows through a sequence of operations) comprehends several phases (for example in MapReduce there are two phases, Map phase and Reduce phase). In between these phases, data must flow from one node to the next one. This data is called intermediate data.

Intermediate data is of extreme importance for the execution of MapReduce programs. First, this large-scale, distributed and short-lived data creates a computation barrier since the Reduce stage cannot start until all intermediate data is ready. Second, the problem is even worse when failures occur.

2.3.1 Characteristics of Intermediate Data

Intermediate data generated by dataflow programs has significant differences from data usually stored in distributed file systems or even local file systems. In order to better understand the characteristics of intermediate data, we now point out the most significant:

- **Size and Distribution** - intermediate data generated by dataflow programs has usually a very large number of blocks with different dimensions and distributed across a very large set of nodes;
- **Write Once, Read Once** - intermediate data follows a write once read once model where each piece of data is created by a stage and read by the next stage. After that, the data will never be accessed again. For example: in a Hadoop[3] job, the output of a particular mapper will belong to a certain region and therefore will be transfered to a specific reducer (the only one that will ever read that information);

[3]Hadoop is an Apache open-source software framework that implements the MapReduce programming model.It can be found at hadoop.apache.org.

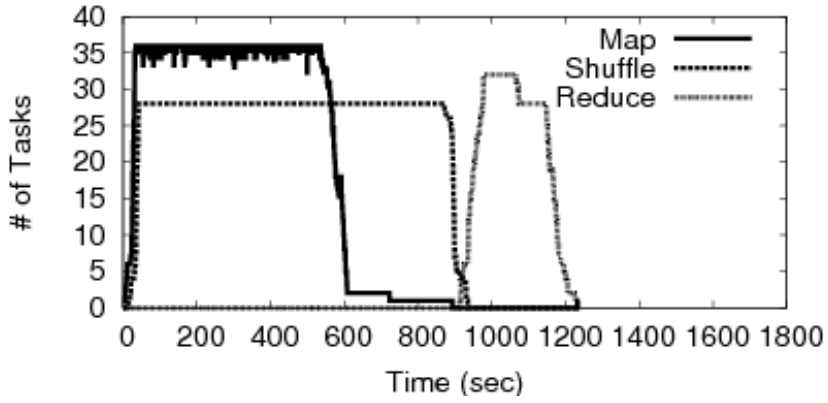


Figure 2.2: Normal behaviour of a Hadoop job

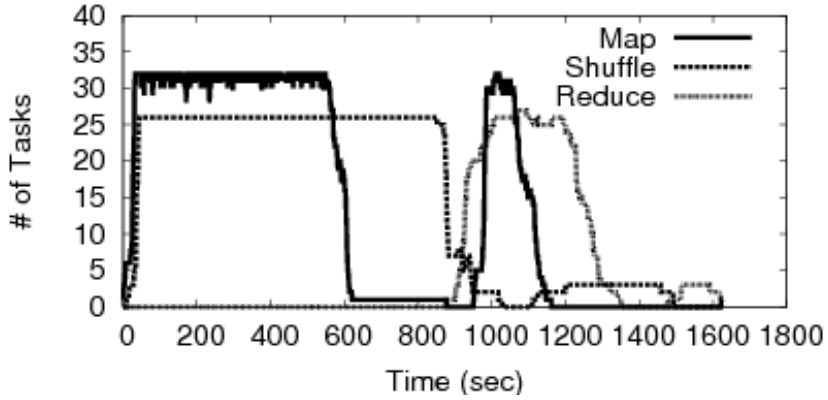


Figure 2.3: Behaviour of a Hadoop job under one Map task failure

- **Short Lived and Used Immediately** - intermediate data is short lived and used immediately since data produced from a stage will be immediately consumed by the next stage. For example: in MapReduce, the output of a particular mapper is transferred during the shuffle phase and then it is used immediately by the reducer task;

- **High Availability Requirements** - high availability of intermediate data is crucial in dataflow jobs since the next stage will not be able to start before all the intermediate data is ready and transfered to the target locations. For example: in MapReduce, the reduce phase will not start until all mappers have already finished and their output has been transfered to the target reducers.

2.3.2 Effect of Failures

As already discussed, faults can dramatically compromise a dataflow job. For instance, in a MapReduce job, a single map task failure will produce a big increase in the overall execution time as we can see in Figures 2.2 and 2.3 (results published in KO [22]). This big increase comes from the fact that both the shuffle phase (where the map outputs are sorted and sent to the reducers) and the reduce phase depend on the map phase. Therefore neither the shuffle nor the reduce phases can start while the map phase is still running.

By looking at both figures we can clearly see that the flow significant changes when there is just one fault. For the first scenario, where there is a normal execution, the job takes approximately 1200 seconds to finish while in the second scenario it takes approximately 1600 seconds. This means an increase of 33% of the overall completion time.

It is still important to note that these results were obtained in a cluster-like environment. In a real volunteer pool, where node volatility (churn) is very high, we might get much larger delays on the overall completion time due to multiple map task failures.

2.3.3 Possible Approaches for Intermediate Data Availability

We now present some approaches for storing intermediate data. Some will not be able to keep intermediate data availability under failures, and most of them are not suited for volunteer computing platforms.

- **Local File System** - typical approaches used by most dataflow programming frameworks store intermediate data on the local file system (or in memory) to avoid the costs of replicating intermediate data. However, this incurs in a large overhead if there is at least one fault. For example, in a Hadoop environment, mappers will host intermediate data locally only;
- **Distributed File Systems** - Distributed file systems could indeed be used to provide better availability of the intermediate data. Systems like HDFS [9], used by Hadoop, can be used to store all the intermediate data generated by the mappers. However, it would cause a severe impact on the performance of reading and writing data from and to HDFS. Previous studies [22] have shown that replicating Hadoop's intermediate data using HDFS will produce too much network interference that will slow down the shuffle phase.
- **Background Replication** - background replication is motivated by the idea that replicating intermediate data should proceed as a background mechanism to avoid interfering with the normal computation and dataflow. This kind of technique can be achieved by using background transport protocols like TCP Nice [41]. Using TCP Nice it is possible to replicate intermediate data without compromising the overall finishing time. However, as it uses only the spare network bandwidth available, it does not give any kind of guarantees about when data is already replicated. There could be scenarios where the replication process is not finished yet when the reduce phase starts.
- **Adaptative Replication** [22] - the final alternative seen so far is a mix of the ones mentioned above in the sense that it could take more or less aggressive replication depending on the cost of re-execution. However, this cost of re-execution, in other words, the amount of time to pay if a fault happens, is very hard to known since there are many unknown variables (like the size of each task, number of tasks left to finish and the number of new clients per second). Depending on the cost of re-executing a map task, one could go for an aggressive approach like in HDFS or could follow (if the cost is low) a background replication process like the one used by TCP Nice.

Despite the solutions just presented, we think that this subject is largely unexplored for the volatile environments. Whereas in cloud environments we might be worried about the failure of a few machines,

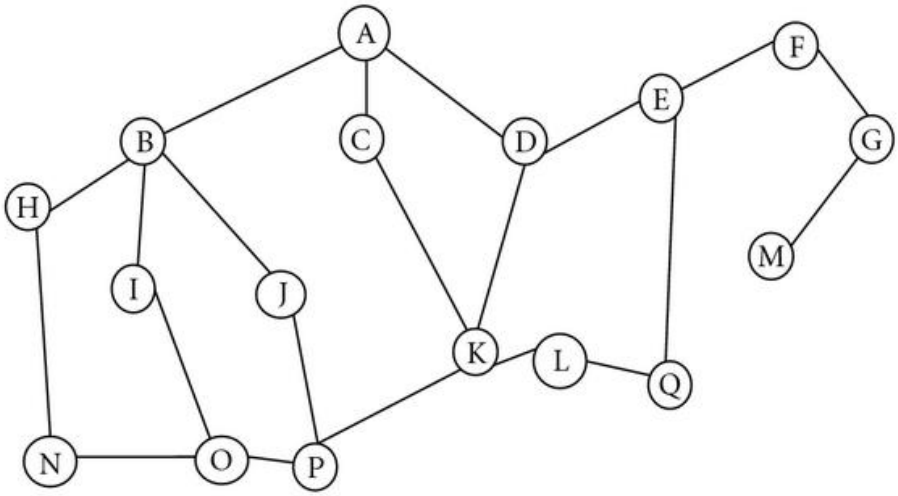


Figure 2.4: Gnutella 0.4 Network

in a volunteer pool, few machines failing or leaving would be a good scenario. Therefore, there is still room for improvements in how to use techniques that could improve the intermediate data availability in volatile environments.

2.4 Data Distribution Systems

In this section we discuss some data distribution systems. Even though these systems might not be designed for distributing data for VC platforms, valuable knowledge can still be extracted and applied to volunteer computing.

2.4.1 Gnutella

Gnutella [32] was the first decentralized file sharing system using a peer-to-peer network overlay. In its first version (0.4) , Gnutella's overlay network (see Figure 2.4) was completely unstructured, all nodes act as client and server (usually called servants). Gnutella provided search for files by flooding the network with query messages. However, this search mechanism raised scalability issues.

The next version of Gnutella (0.6) fixed the scalability issue by dividing nodes into super-peers (or ultrapeers) and leaf-peers (see Figure 2.5) and thus creating an additional network level (leading to a 2-level hierarchy network overlay). With this new version, flooding was restricted to super-peers and therefore network traffic was reduced.

Even in the second version, Gnutella still relies on flooding protocols to locate files and it results on a higher bandwidth consumption and higher search time. This is a direct consequence of using unstructured networks. Another important issue/disadvantage with the second version of Gnutella is the node asymmetry (since super nodes will be much more loaded than regular peers).

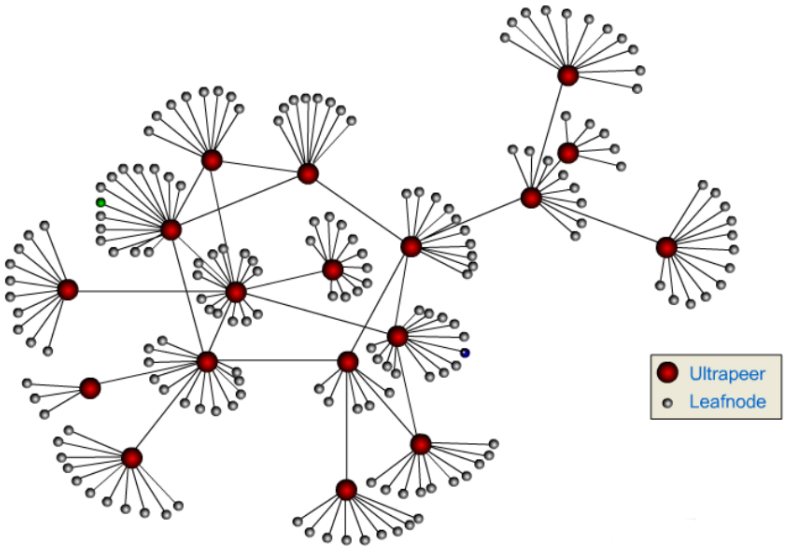


Figure 2.5: Gnutella 0.6Network

2.4.2 eDonkey2000

eDonkey2000 [20] (mainly known by one of its implementations, eMule) is a peer-to-peer file sharing protocol. This protocol assumes the existence of several servers that (like in the second version of Gnutella) are used to provide information about peers connected to other servers (search) and are used as a network entry point. In contrast to Gnutella, the servers used by eDonkey are dedicated and are not regular peers.

Using this protocol, clients share files among themselves and exchange information about other servers and peers. Clients behind firewalls are labelled as ”Low ID” and connections to these clients are made through servers. All clients are identified by a unique identifier, created and assigned upon the first time the client joins the network. Each client accounts the credit (amount of data downloaded) for all the other clients it has ever contacted. Clients prefer to upload data to the ones with more accounted credit.

As in BitTorrent (described in Section 2.4.7), files are divided into small parts that, in fact, are the only ones being shared. Despite having the same mechanism of sharing files in parts, eDonkey implementations, namely eMule, are not as efficient as BitTorrent. This mainly has to do with the slow credit increase in eDonkey2000 (nodes prefer to upload to others from whom they have received from, i.e., new peers will have difficulties starting to download).

2.4.3 BitDew

BitDew [17] is a programming framework that provides a programmable environment with automatic and transparent data management for computational Desktop Grids. It was created with the insight that desktop grid applications need to access large volumes of data and that little attention has been paid to data management in large-scale, dynamic, and highly distributed grids.

The programming model given to programmers is similar to the Tuple Space model in the Linda [19]

programming system; it works by aggregating storage resources from nodes and virtualizing them in a unique space where data is stored. However, as most computation platforms, BitDew needs stable nodes to work. It needs several services (Data Repository, Data Catalog, Data Transfer and Data Scheduler) to be working on centralized nodes. The unavailability of these nodes might compromise the data management system. This goes against our objectives of freeing volunteers from the centralized node (and the other way around, to reduce the load). Therefore, we do not use BitDew since it demands high availability of several management services, which is not possible in a volunteer pool setting.

2.4.4 FastTrack (Kazaa)

FastTrack [24] is another protocol for peer to peer file sharing. Its was most popular in 2003 while it was being used by Kazaa[4]. FastTrack also follows the distinction of super nodes and regular nodes. Super nodes are well connected nodes, with a public IP and with high availability. Regular nodes are users (clients).

Although this system had great impact with applications like Kazaa, iMesh[5] and Morpheus, it is a closed protocol which restricts us from analyzing it further.

2.4.5 Oceanstore

Oceanstore [23] is a global-scale persistent storage system. The system is comprised of untrusted servers where data is protected through redundancy and cryptographic techniques. Oceanstore creators envisioned a system where storage could be traded and where users may use the system to store all their data. Oceanstore runs on top of untrusted servers and uses cryptographic techniques so that anyone could provide secure storage.

Although Oceanstore seems like a good solution for storage, it is not suited for the high efficiency data distribution requirements needed for MapReduce since it relies on complex techniques for data consistency (file versioning), routing (distributed hash tables), data security (file encryption). Furthermore, we find these techniques unnecessary since: 1) data consistency is not a problem in MapReduce (there are no concurrent writes); 2) data securiry (confidentiality, integrity or authenticity) are not requirements for freeCycles; 3) routing is simplified by the reduced number of nodes (mappers and reducers normally do not go beyond few thousands, even in big companies like Google).

2.4.6 FastReplica

FastReplica [11] is a replication algorithm focused on replicating files in Content Distribution Networks (CDN). This algorithm was built to be used in a large-scale distributed network of servers. The authors point out that pushing large files to CDN servers is a resource-intensive problem; so, they present FastReplica as a solution for a more efficient way to replicate data through CDN servers.

[4]Kazaa's old web page can be found at www.kazaa.com.
[5]iMesh can be found at www.imesh.com.

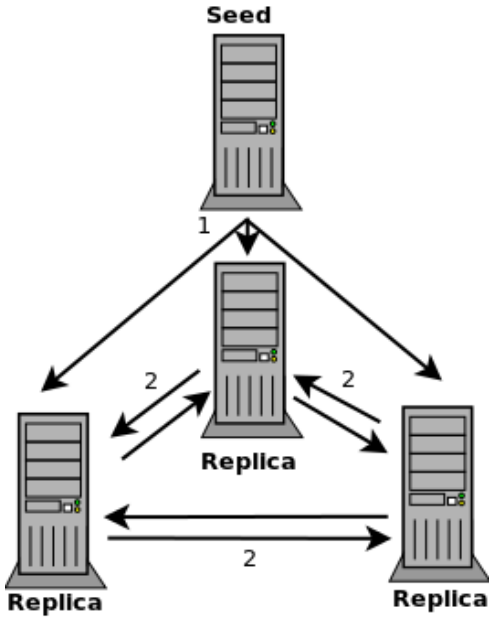


Figure 2.6: FastReplica Workflow

The algorithm is designed to work with a small set of nodes (ranging from 10 to 30 nodes) and comprehends two steps:

- **Distribution Step**
  1. the originator (holder of the content to distribute) opens a concurrent network connection to all server nodes that will have a copy of the file;
  2. the original file is divided into chunks (one for each server node);
  3. the originator sends each file chunk to each server so that each one knowns only one piece of the original file.
- **Collection Step**
  1. after receiving the corresponding chunk of the file, each server will open concurrent network connections to all other servers;
  2. each server sends its chunk of the file to all the other servers;
  3. each server has now all the file chunks and is able to reconstruct the original file.

Figure 2.6 presents a graphical representation for the FastReplica algorithm. The step number 1 stands for the distribution step while the step number 2 stands for the collection step.

FastReplica also supports replicating files for a large number of servers (using the previous algorithm multiple times). Throughout the initial tests, it was possible to show that FastReplica is able to reduce the time needed to replicate a file across different servers when compared to sequential transfer and multiple unicast transfers.

Despite the good results, FastReplica seems to be inadequate to our attain our goal. First, it relies on a push model (which is very common in CDNs). A push model would, for example, force mappers

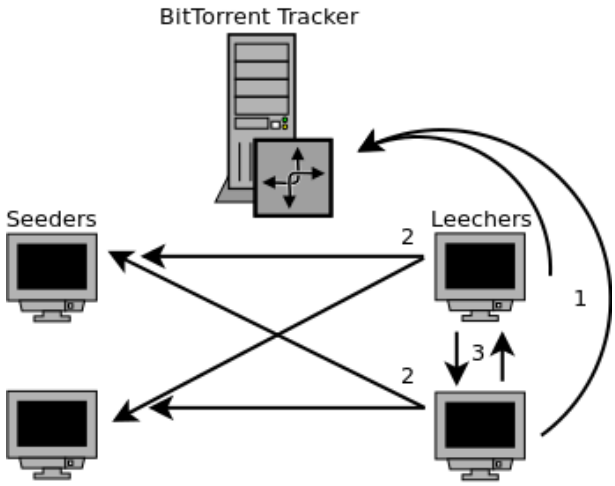


Figure 2.7: BitTorrent Workflow

to send data to reducers (that might not even exist yet when the map task is finished). Since the set of reducers that need to receive data from a particular mapper is not constant (due to assigning reducer task replicas or node failures), the problem of pushing information to a varying set of nodes seems quite hard.

The second reason is that there is no obvious way to cope (using the provided protocol) with node failures or how to handle replication of data. This is because FastReplica was designed for CDN servers and therefore its not suited to handle variable sets of nodes (which is the normal scenario in volunteer pools).

2.4.7 BitTorrent

BitTorrent is a peer-to-peer data distribution protocol used to distribute large amounts of data over the Internet. Nowadays, it is, perhaps, the most used protocol to share large files as it is widely used on the Internet. It was designed to avoid the bottleneck of file servers (like FTP servers).

BitTorrent can be used to reduce the server and network impact of distributing large files by allowing ordinary users to spare their upload bandwidth to spread the file. So, instead of downloading the whole file from a server, a user can join a swarm of nodes that share a particular file and thereafter, download and upload small file chunks from and to multiple users until the user has all the file chunks. Using this protocol it is possible to use computers with low bandwidth connections and even reduce the download time (compared to a centralized approach).

In order to find other users sharing the target file, one has to contact a BitTorrent Tracker. This tracker has to maintain some information about the nodes such as a node's ip, port to contact and available files. This information is kept so that when a node asks for a file, the tracker is able to return a list of nodes to whom the node should contact.

Figure 2.7 presents a graphical description of the protocol which goes through the following steps:

1. new leechers[6] contact the BitTorrent Tracker to know which nodes to contact for a particular file;

2. the nodes that already have some fraction of the file receive requests from the new peers;

3. as soon a leecher has some chunk of the file, it can start sharing it with other leechers.

As opposed to Gnutella, eDonkey and KaZaA, BitTorrent does not provide file search. Users share files by publicising `.torrent` files on web sites, blogs, etc. Each `.torrent` file contains enough information for a peer to be capable of finding other peers that have at least some file chunks from the target file.

This protocol has proved to be capable of scaling up to millions of users and providing high efficiency in distributing large amounts of data [31]. Thus, we intend to integrate this protocol in our solution since it enables a pull model where data can flow as volunteers need it (as opposed to FastReplica).

2.5 Relevant Volunteer Computing Platforms

In the next subsections we present and analyze several VC platforms. We start by giving a little reference to commercial (or paid) VC systems and then we move to pure VC systems, the ones that we think to be the most relevant to freeCycles.

2.5.1 Commercial Volunteer Computing

In pure VC, volunteers are assumed to be altruistic, i.e., volunteers work for the benefit of common good. However, it is not always like that. There might be platforms that give incentives to volunteers (although the concept of volunteerism becomes a little bit vague).

One way to increase participation on such systems is individual compensation for each finished work (some systems might even promote the trade of compensations in a market like fashion). A lottery system might be used as well. Using this system, a single worker, the one that is able to give the correct answer to the problem, gets the compensation.

A contract-based system can also be used to trade a particular service by some computation. For example, information providers such as search engines, news sites, and shareware sites, might require their users to run a Java applet in the background while they sit idle reading through an article, or downloading a file. While it is actually possible to hide forced worker Java applets inside web pages, for example, most users will consider it ethically unacceptable. Such sites should clearly allow users to contribute or to back-out.

One such company that uses browsers to produce computation is CrowdProcess [7]. Using CrowdProcess, web site owners can manage workloads that their own web site's clients will work on. In other words, CrowdProcess enables web sites owners to use the computational power of their web site visitors. Web site owners only have to add some code (generated by CrowdProcess) to their web pages.

[6] In the context of the BitTorrent protocol, a leecher is a node that does not have all parts of the file that it is downloading.
[7] crowdprocess.com

Another company focused on VC is CoinGeneration[8]. CoinGeneration simply buys computation from volunteers. To participate, each volunteer needs to install a client software that will consume some resources and will account the contribution. According to their web site, with time, volunteers will earn money, $1 for a day of computation, per thread.

Through our research for commercial VC platforms, we did not find any significant active platform. Some companies used the concept of VC in the decade of 1990 but most of them are now closed (some companies in this situation are Cell Computing, Popular Power, United Devices, and Entropia).

Nowadays, most commercial companies are focused on grids, how to use and optimise them to process large volumes of data.

2.5.2 BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) is a software platform built to help scientists creating and operating large scale VC projects. It is one of the most successful VC systems and it is, historically, one of the most responsible projects for raising the awareness of VC around the world. Furthermore, as we will see (in Chapter 3), it is one of the building blocks for our solution.

The big motivation for BOINC is the fact that the world's computing power and disk space is no longer concentrated in supercomputer centers but instead it is distributed in hundreds of millions of personal devices. The use of this unexploited computing power and storage will enable previously unfeasible research.

The BOINC platform is meant to be used by computer scientists to run their projects on a volunteer pool. Each project has independent servers. Clients chose which projects to participate (each client can participate in multiple projects at the same time).

The BOINC server comprises several functionalities that can be mapped to one or multiple servers:

- holding the input and output (**Data Server**);
- holding the information about jobs (**Database Server**);
- scheduling tasks to volunteers (**Scheduler**);
- perform task validation (**Validator**).

A strict Client-Server architecture is used. Communication from server to client or between clients do not exist (this way it is possible to avoid problems with firewalls and/or NAT). Therefore, the usual workflow is very simple (see Figure 2.8):

1. clients (volunteers) request work (1) and receive a workunit (a workunit is a file describing the task, it mentions the input, what output is expected and how to compute it);
2. clients will then download the input data from the Data Server (2);
3. the computation is done (3) and the output is produced;

[8] coingeneration.com

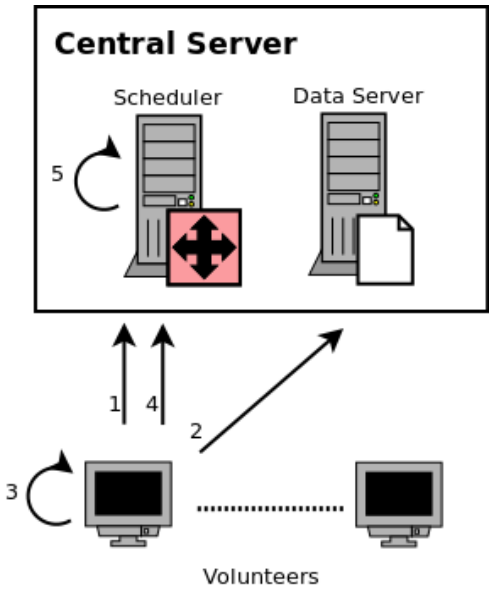


Figure 2.8: BOINC Workflow

4. the output is uploaded (4) to the BOINC Data Server;

5. the output is validated (5).

Majority voting is the provided validation mechanism. Each project is able to set the desired replication factor for tasks' execution. When results appear it is up to the application to decide whether to wait for more results or to accept a specific result.

As the goal of BOINC is to support the execution of applications over a large volunteer pool (millions of devices), a centralized approach can be very restrictive. Consequently, BOINC uses a failure and backoff mechanism that prevents the server from being overloaded with clients' requests. Using it, a failed request will make the client wait for some time (the amount of time gets bigger with the number of failed requests).

BOINC performs crediting/accounting. This means that it is possible to verify the amount of computation or disk space donated by every volunteer. Thus, it is possible to ”reward” the volunteer. This is an interesting feature to attract new volunteers.

In fact, BOINC has two ways of repaying clients: i) showing charts with the biggest donors in their web site, and ii) with a screensaver program (which normally displays information about the task that the volunteer is executing).

BOINC has been one of the most successful VC platforms. It is currently hosting around 70 projects with a combined total of more than 2.5 million volunteers donating their computation resources[9].

Despite all the good results, BOINC does not support MapReduce jobs (which is one of our requirements) and uses point-to-point data transfers (both for input and output) which do not take full advantage of the volunteer's bandwidth and produce a bigger overhead on the central server.

[9]Statistics from boincstats.com

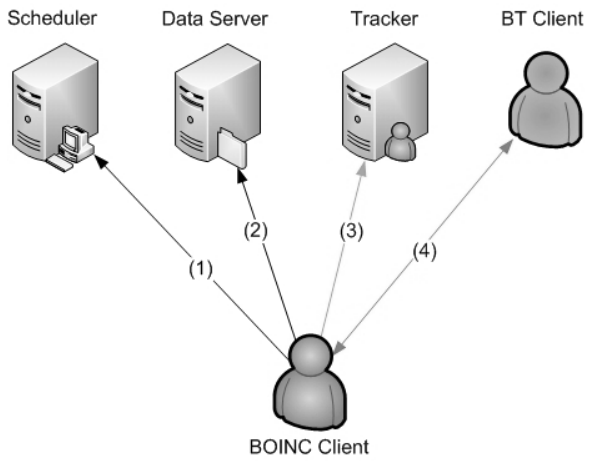


Figure 2.9: BOINC-BT Input File Transfer

2.5.3 BOINC-BT

BOINC-BT [12] is an optimization of the data distribution layer of BOINC using BitTorrent. The system was motivated by the following: i) BOINC uses a strict master/worker scheme that does not take advantage of the clients network bandwidth and, ii) BOINC data servers will eventually be a bottleneck for projects that consume large amounts of data.

BitTorrent was suggested as a way to avoid data servers' bottleneck by using the volunteer's bandwidth to help to distribute data. In order to use BitTorrent in the data distribution layer, BOINC-BT developers had to change the BOINC server and client implementation. BOINC-BT also added BitTorrent clients in both the server and in all clients. Clients without the BitTorrent client would use the traditional way and download the input files through HTTP (from BOINC data server).

BOINC-BT also added another service to the BOINC central server(s): a BitTorrent tracker. This tracker is used (as in the BitTorrent protocol specification) to disseminate information about clients that share a particular file.

For a particular input file to be shared with BitTorrent, a `.torrent` file is created pointing to the central tracker. This `.torrent` file is stored along with the file itself inside the project data server.

In order to start sharing the file, the BOINC server must start the BitTorrent client to act as a seed and announce itself to the tracker. Volunteers will then (after starting their BitTorrent client) be able to download and share the input file.

Volunteers get access to the `.torrent` file once they retrieve their work unit description file. The `.torrent` is referenced by this work unit file and then the client can download the `.torrent` from the project data server.

As shown in Figure 2.9, BOINC-BT file transfer is done in several steps: 1) the client contacts the scheduler asking for work and receives a workunit; 2) the client downloads the `.torrent` file from the data server (through HTTP); 3) the client starts his BitTorrent client (giving the `.torrent` file) that will contact the tracker to obtain information about other peers sharing the same file; 4) the BitTorrent protocol is used to download and upload file parts to and from other volunteers.

With BOINC-BT it was possible to achieve a reduction of the BOINC project data server's network bandwidth utilization by 90%. This, however, caused an initial spike of CPU utilization (about 50%) even for a little number of nodes sharing the file.

As a final comment, we can point out two disadvantages regarding our goals: i) the system does not support MapReduce jobs, and ii) only the input is distributed with the BitTorrent protocol.

2.5.4 SCOLARS

Scalable COmplex LARge Scale Volunteer Computing (SCOLARS) [14] is a VC platform that enables the execution of MapReduce applications over the large scale internet (what we have been calling a volunteer pool). It was designed with the intuition that day after day, more and more devices are connected to the internet. Thus, more decentralized architectures for task and data distributing as well as for result validation are needed to prevent current master/worker models from being a bottleneck. MapReduce was chosen because of its increasing importance and popularity in cloud computing environments. The project was built on top of BOINC since it is (according to the authors) the most successful and popular VC middleware.

This system makes two important contributions:

- **Inter-client transfers** is used to transfer map outputs to the target reducer(s). In a normal BOINC scenario, all map outputs would have to go back to the project data server and then when a new task (reducer) comes, the map output would have to be downloaded from the project data server. In order to reduce the burden on the project data server, SCOLARS supports the direct transfer of map outputs from the volunteer playing the map role to the volunteer playing the reduce role.

- **Hash-based task validation** is used instead of the BOINC default procedure. In a MapReduce execution, on top of an unmodified BOINC platform, all map outputs would have to be uploaded to the project data server in order to proceed with the task validation. To solve this enormous time and bandwidth expenditure, SCOLARS introduces the idea of validating the hash[10] of the intermediate output instead of the output itself. Therefore, volunteers playing the map role only have to upload the hash of the produced output (which is much smaller than the output itself).

By using these two features it is possible to use MapReduce paradigm keeping a low overhead on the central server. It is also possible to provide a more fault tolerant central server as the amount of time that volunteers depend on the server is reduced (by using the previously explained features).

SCOLARS is a BOINC compatible solution in the sense that volunteers with the original BOINC software installed can participate in projects hosted by SCOLARS' servers. They, however, will not be able to use inter-client transfers and the hash-based validation.

Figure 2.10 summarizes the execution model followed in SCOLARS. We describe it very briefly: 1,2) mappers request work from the server and then download the input from the data server; 3,4) mappers compute the map output and send the hash to the server; 5,6) task output validation is done on the

[10] We use the concept of hash to refer the result of applying a hash function to some data.

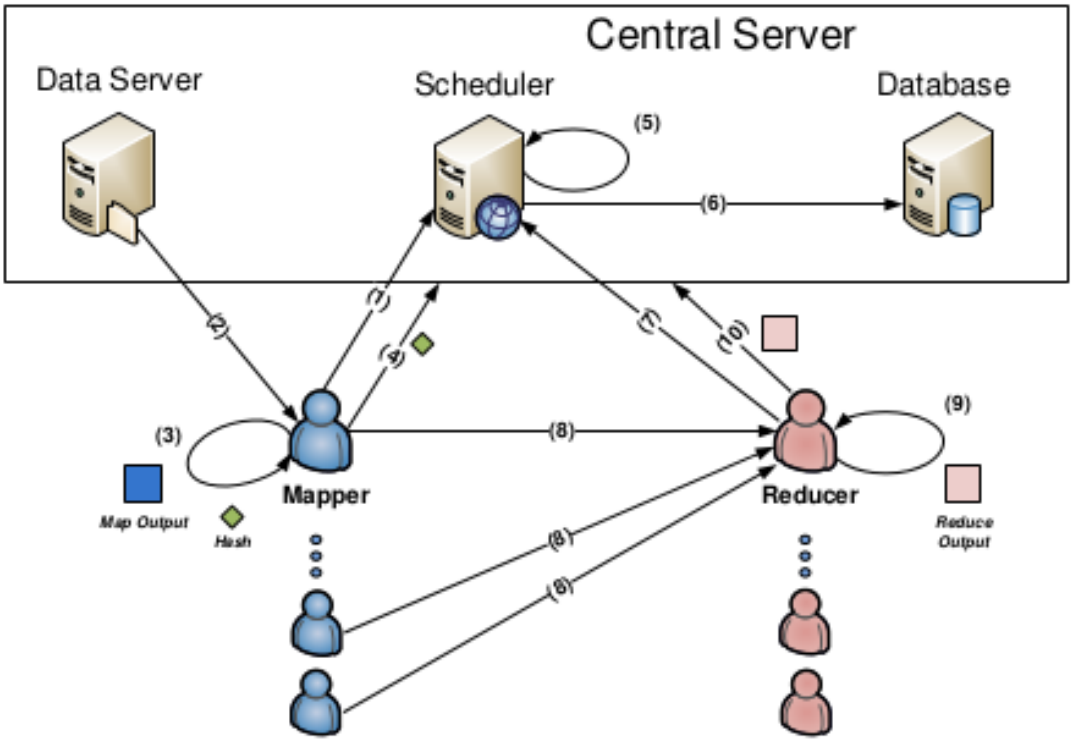


Figure 2.10: SCOLARS Execution Model

scheduler and, if the result is valid, the address of each mapper is stored on the database; 7) reducers request work and receive information indicating the location where to obtain the input from; 8,9) the input file is downloaded from mappers and the reduce operation is done; 10) finally, the reduce output is uploaded to the central server.

After evaluating the system [13], it was possible to show that SCOLARS performed better than the original BOINC system executing a MapReduce job (PlanetLab[11] was used to deploy volunteer nodes). For the word count application turnaround time it was possible to obtain a reduction of 65% compared to BOINC. As for the server network traffic, SCOLARS was able to cut the amount of data sent to clients by half. Regarding download traffic, SCOLARS was able to reduce the amount of data downloaded (from clients) by 92% compared to BOINC.

However, SCOLARS presents three main issues that do not fulfill our goals:

- all data transfers are done point-to-point and thus, do not take full advantage of the volunteers' network bandwidth (i.e., since multiple clients will have the same input and will compute the same output, for replication purposes, we could use the aggregated upload bandwidth to speed up data transfers);

- there is no explicit procedure to cope with the loss of intermediate data; this means that, if a mapper fails, the whole MapReduce job will be highly delayed (since the reduce phase will have to wait for a new map task);

[11] PlanetLab is a group of computers available as a testbed for computer networking and distributed systems research. It can be found at www.planet-lab.org

- there no is way to leverage the data already on the volunteers between MapReduce cycles, and therefore, MapReduce applications with multiple cycles will always depend on the central data server to download and upload all input and output data.

2.5.5 MOON

MapReduce On Opportunistic eNvironments [25] (MOON) is an extension of Hadoop, an open source implementation of MapReduce. According to its creators, MOON is a response to the bad task and data replication schema of MapReduce implementations for resources with high unavailability.

By analysing Hadoop, three main problems were found:

- First, the Hadoop Distributed File System (HDFS) relies on replication to provide reliable data storage. In order to maintain the same data availability on a volatile environment, one has to replicate the same data over a much bigger set of nodes which rapidly becomes prohibitive;

- Second, Hadoop does not replicate intermediate results. As in volatile environments the time between failures is much smaller (than in others environments such as clusters), it is highly probable that intermediate results might get lost;

- Third, the Hadoop task scheduler assumes that most tasks (mappers and reducers) will run smoothly until completion. However, in a volatile environment, task failure is the expected scenario. Therefore, the authors state that the Hadoop task scheduler is not well suited for volatile environments.

Note that within the context of this system, a volatile environment is an institutional grid and not a volunteer pool (as we have been describing so far). An institutional grid can be formed by a set of computers from a university, scientific laboratories, companies, etc.

Even though MOON presents both data and task scheduling, we only focus on data scheduling since task scheduling techniques are out of the scope of our work.

MOON relies on two sets of nodes: volatile volunteer nodes and dedicated computers. This big asymmetry is justified by the fact that assuring availability while keeping good access bandwidth can be easily achieved by placing one copy in a dedicated node and multiple replicas spread over the volatile nodes.

However, two important assumptions are made: i) it is assumed that the dedicated nodes have enough aggregate storage for at least one copy of all active data in the system (which can be huge), and ii) dedicated nodes do not fail.

Under these assumptions, MOON schedules data according to its availability requirements and free capacity on dedicated nodes. Figure 2.11 illustrates the algorithm to decide where to place data.

After evaluating the system, it was possible to achieve high data availability with much lower replication cost compared to Hadoop. However, the limited I/O bandwidth on the dedicated nodes caused an extra delay in the overall execution of the MapReduce jobs.

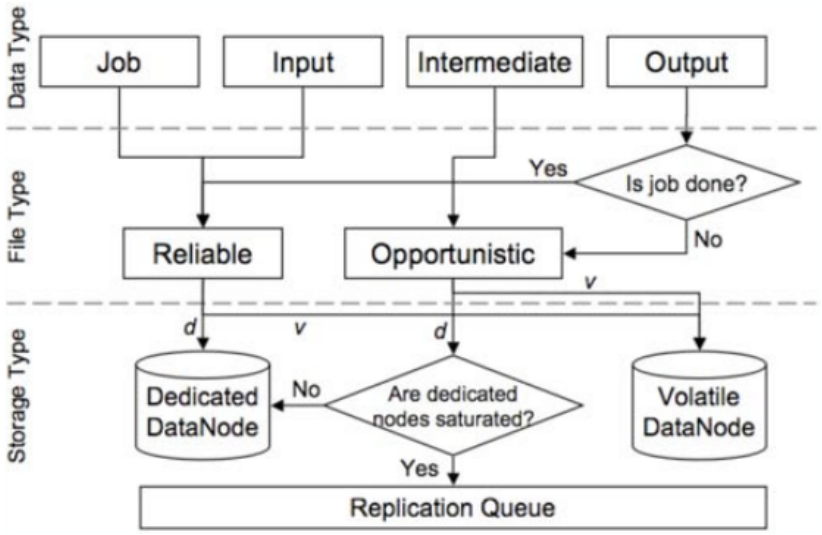


Figure 2.11: Decision process to determine where data should be stored.

As a final analysis against the requirements we specified, we can conclude that MOON does not overlap our goal system since: i) it was designed for Desktop Grids, where there are dedicated nodes to store all data (this assumption does not hold on volunteer pools); ii) data transfers are done point-to-point (which would incur in high burden on the central server and underutilization of the volunteer's network bandwidth).

2.5.6 MapReduce for Desktop Grid Computing

The solution presented by Tang et al. [39] is an implementation of the MapReduce programming paradigm focused on large-scale Desktop Grids. According to its authors, Desktop Grid middleware systems are mainly focused on Bag-of-Task applications with few I/O requirements and thus, are not suited for MapReduce computations that are characterized by the handling of large volume of input and intermediate data.

Therefore, this solution is presented as a way to cope with massive fault tolerance, replica management, barriers-free execution, latency-hiding optimization, and distributed result checking. Many of these challenges are solved utilizing the Bitdew [17] data management framework (the one analyzed in Section 2.4.3).

As in most Desktop Grid computational platforms, nodes are divided into two groups: i) stable or dedicated nodes, and ii) volatile nodes. Service node(s) run various independent services which compose the runtime environment: BitDew services and MapReduce services (equivalent to the JobTracker in a Hadoop cluster). Volatile nodes provide storage (managed by BitDew) and computational resources to run Map and Reduce tasks. The integration of BitDew with the MapReduce implementation begins right at job submission where input files are registered in the BitDew services and thereafter are available in all nodes.

We now describe how this system solves some of the promised features (we only show the ones that

fall within the scope of this project):

- **Latency Hiding** is achieved by pre-fetching data. As soon as a running task ends, new data is available to be processed avoiding unnecessary waiting time;
- **Collective File Operation** is used for input distribution and in the shuffle phase; it is achieved mainly through the BitDew framework (described in Section 2.4.3);
- **Barrier-Free Computation** is motivated by the fact that due to high node churn, in Desktop Grids, reduce tasks might have to wait for intermediate output re-execution (in case of a node performing a map task fails); to deal with this issue, the MapReduce paradigm was augmented with a new operation: a reduce operation over a sub-set of keys. Therefore, the barrier between Map and Reduce tasks is removed.

However, it is important to note that the last described feature, that tries to remove the barrier between Map and Reduce tasks, inserts a new step for aggregating the partial outputs. In order to use it and take advantage of it, all MapReduce applications must be re-designed.

Regarding the evaluation of this system, no direct comparison is made with other computing platforms, not even Hadoop. Furthermore, a cluster was used as test-bed, which can distort most results. The only results presented, that could be used to compare with other solutions (like Hadoop), are from the word count application throughput versus the number of nodes. The system was able to produce 1200MB/s with 512 nodes. However, we believe that it could be useful more evaluation and comparisons of this system with others.

We also point out that we believe that this system is not suited for large scale VC. This is mainly because it uses the BitDew (see 2.4.3) framework that would produce high overhead in a VC setting and assumes high availability of a particular set of nodes (dedicated nodes to store all data).

2.5.7 MapReduce for Dynamic Environments

The system developed by Marozzo [27] is presented as a solution to exploit the MapReduce model for creating and running data-intensive applications in dynamic environments. According to its authors, current approaches are too focused on cluster environments and thus do not take into account frequent node churn/failure (that can happen to both worker and master nodes). The main goal of this system is to help in the transition of the MapReduce paradigm to Grid-like and peer-to-peer dynamic scenarios.

Within this system there are two classes of nodes: master and slave. The role is assigned dynamically and, at each time, there is a limited set of nodes playing the master node while all the others are playing the slave role. Moreover, each master node can act as a backup node for other master nodes.

A user can submit a job to one of the master nodes. The selected master node will then coordinate the job as in a normal MapReduce application. This master will periodically checkpoint this job in his backup nodes so that, in case of a master failure, one of the backup nodes can restart the job from the last checkpoint.

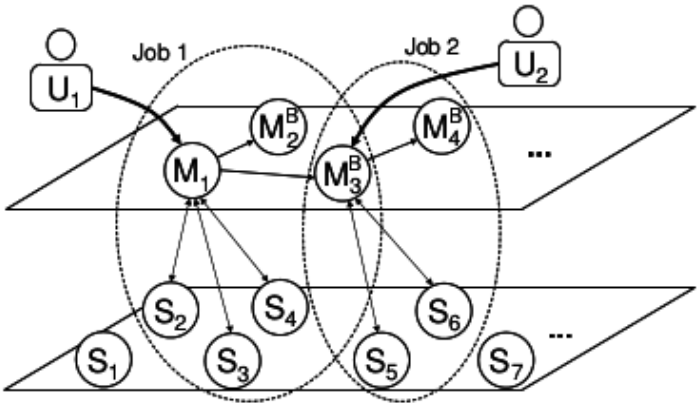


Figure 2.12: Execution Environment

Figure 2.12 presents a simple execution example where there are two jobs (Job 1 and Job 2) submitted by two users (User 1 and User 2). Job 1 uses Master 1 for the master role and Slaves 2, 3 and 4 as slaves while Job 2 uses Master 3 as master and Slaves 5 and 6 as slave nodes. Masters 2 and 3 are used as backup masters for Master 1. Master 4 is used as backup master for master 3.

Regarding the goals we proposed for freeCycles, three issues can be found concerning this system: i) data transfer is done point-to-point (which fails to fully utilize the volunteer's network bandwidth); ii) the system does not provide a way to ensure or at least try to maintain the availability of the intermediate data; iii) the system does not reuse output data to perform subsequent MapReduce cycles, i.e., each MapReduce cycle is independent and therefore, output data (from reducers) can not go directly to the mappers of the next cycle.

2.5.8 XtremWeb

XtremWeb [16] is presented as a generic global computing system. It aims at building a platform for experimenting with global computing capabilities by using spare resources around the world. XtremWeb is a Desktop Grid system so it is mainly planned for aggregating clusters and smaller grids.

The authors identified two essential features (requirements) for the XtremWeb design: multi-application (so as to allow multiple institutions or enterprises to setup their own global computing applications or experiments), and high-performance.

XtremWeb's architecture follows a three-tier design: worker, client, and coordinator. The worker is the regular node that shares resources by allowing others to use its CPU, network bandwidth, and storage capacity. The client is the job submitter. Clients are allowed to submit task requests to the system which will execute them on workers.

There is no direct communication between clients and workers (not even file transfers). Instead, a coordinator stands between clients and workers. The role of this third tier is to decouple clients from workers and to coordinate task execution on workers. Within the usual job execution workflow, the coordinator is responsible for: 1) accepting task requests coming from several clients; 2) distributing

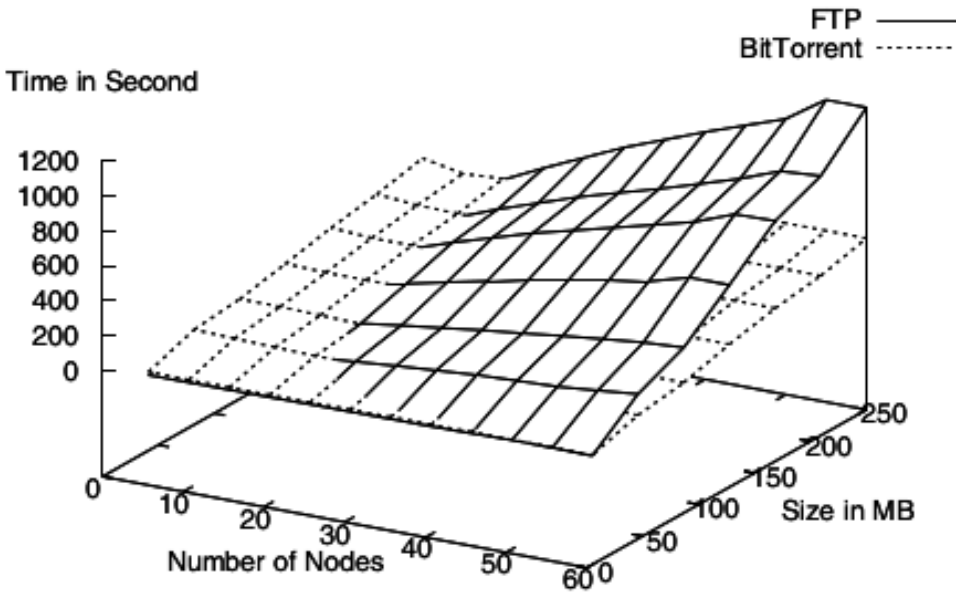


Figure 2.13: Performance comparison of FTP and BitTorrent [42]

tasks to the workers according to a scheduling policy; 3) transferring application code and input files to workers if necessary; 4) supervising task execution on workers; 5) detecting worker crash/disconnection; 6) re-launching crashed tasks on any other available worker; 7) collecting and stores task results; 8) delivering task results to client upon request.

Enhancements with BitTorrent for Input File Distribution

Several years after the appearance of XtremWeb, the authors developed a prototype (based on the same system) to evaluate the BitTorrent protocol for computational Desktop Grids[43, 42]. This prototype had two additional entities within the coordinator tier: the data catalogue and the data repository. The data catalog (centralized service) is used to keep track of data and their location (just like a BitTorrent tracker). On the other hand, the data repository is where the data is stored.

In order to assess the possible gains of applying BitTorrent for input file distribution, the authors compared two protocols: FTP and BitTorrent. By analysing the Figure 2.13 it is possible to see that BitTorrent outperforms FTP transfers (i.e. transfers are faster) when the number of nodes or the size of the file increases. It is also possible to conclude that, for FTP, the file transfer time increases linearly with the number of nodes and the size of the files. However, using BitTorrent, the time to transfer the file keeps almost constant while the number of nodes increases. Thus, BitTorrent is much more scalable than FTP.

As for our final observation about XtremWeb, although it uses BitTorrent to distribute data, it does not support MapReduce jobs; thus, it does not meet one of the most important requirements we proposed.

2.6 Summary

In this section, we present and discuss the table summarizing each of the VC platforms previously described according to a set of relevant parameters (see Table 2.1). We draw our comparison based on the following six parameters:

VC System	Environment	Input Distribistribution	MapReduce Support	Intermediate Output Replication	Intermediate Output Distribution	Input Distribution for Subsequent MapReduce Cycles
BOINC	Volunteer Pool	Direct transfer from central server (using HTTP)	No	NA	NA	NA
SCOLARS	Volunteer Pool	Direct transfer from central server (using HTTP)	Yes	No	Direct transfer from volatile node	All data between MapReduce cycles must go through the central data server.
BOINC-BT	Volunteer Pool	Using the BitTorrent protocol	No	NA	NA	NA
MOON	Desktop Grid	Direct transfer from dedicated node	Yes	Yes	Direct transfer from dedicated or volatile node	The system uses the underlying replicated system to keep all output data stored on multiple nodes.
work by Tang [39]	Desktop Grid	Using the BitDew framework	Yes	Yes (through Map task replication)	Using the BitDew framework	Using BitDew, the system stores all output data on several nodes.
work by Marozzo [27]	Desktop Grid	Direct transfer from volatile node	Yes	No	Direct transfer from volatile node	All data between MapReduce cycles must go through one single node (the one responsible for the job).
XtremWeb	Desktop Grid	Using the BitTorrent protocol	No	NA	NA	NA

Table 2.1: Volunteer Computing Platforms Taxonomy

- Environment: target environment for the platform. Systems' environment might be classified as Volunteer Pool (for pure VC systems) or Desktop Grid (for systems mainly focused on Desktop Grids);
- Input Distribution: how the system distributes input data;
- Support for MapReduce applications: whether or not the system supports MapReduce applications;
- Intermediate Output Replication: whether or not the system replicates intermediate output data. This parameter is not applicable to systems that do not support MapReduce applications;
- Intermediate Output Distribution: how the system distributes intermediate output data; This parameter is not applicable to systems that do not support MapReduce applications;
- Input Distribution for Subsequent MapReduce Cycles: how input data for subsequent MapReduce cycles is performed. This parameter is not applicable to systems that do not support MapReduce application.

As the main goal of freeCycles is to provide efficient data distribution for new parallel programming paradigms, in particular, MapReduce running over volunteer pools, we conclude, by analysing the presented table, that no current system fulfills all our goals. The system closer to our goals is the one presented in Tang [39] which is built on top of BitDew. However, as we have seen, BitDew was designed to run on Desktop Grids and thus is not suited for volunteer pools. As for pure VC platforms, SCOLARS [14] is the one closer to our goals; however, it uses direct transfers for all data and therefore fails to fully utilize all the volunteers' upload bandwidth.

Regarding the support for multiple MapReduce cycles and how different systems provide input data to subsequent cycles, no system provides an efficient (in our opinion) approach to the problem. All systems provide multiple MapReduce cycles by repeating the same procedure several times. This becomes prohibitive in systems (SCOLARS for example) where a central data server is used (since all data always have to go to the central data server). Solutions that use automatic replication (such as MOON [25]) do not have this problem (since data is stored on multiple nodes). However, this does not solve the problem. Data still has to be moved to some intermediary node(s) before the next cycle begins.

In the next chapters, we present and evaluate our solution, freeCycles, that solves all the pointed issues and fulfills the all the proposed goals.

Chapter 3

freeCycles

freeCycles is a MapReduce enabled and BOINC [4] compatible VC platform. It provides two main new contributions: i) efficient data distribution (for input, intermediate output and output) by using the BitTorrent protocol and ii) enhanced task scheduling that minimizes the loss of intermediate data. Along this chapter, we start by describing how and which techniques are uses by freeCycles to achieve the proposed goals. Next, we present our data distribution algorithms (used to provide fast and scalable data transfers in the context of MapReduce jobs). We finish this chapter with some techniques, used by freeCycles, that address the availability problem of intermediate data.

3.1 Architecture Overview

freeCycles brings together several concepts from different areas: VC (BOINC), programming models (MapReduce), and data distribution protocols (BitTorrent). BOINC is a VC platform that is used to harvest computational resources from volunteer nodes. MapReduce is a programming model typically used in clusters. BitTorrent is a widely used and very popular peer-to-peer data distribution protocol, used to share data over the internet.

Throughout the previous chapters, we have motivated our solution that takes advantage of these techniques to fulfill the goals we proposed. Now, and before going into more detail on our solution, we give a brief overview about the architectural components and how we join them together these different concepts: BOINC, MapReduce, and BitTorrent. We start by showing how BitTorrent can be used as a data distribution protocol in BOINC, and after, how to build a MapReduce framework on top of the VC platform already using BitTorrent for data distribution.

As described in Section 2.5.2, BOINC (and therefore freeCycles) is divided in two major components: client and server. The server is responsible for delivering and managing tasks (work units) while the client is responsible for executing and reporting results. At a very high level, the server is composed by a scheduler (that manages and delivers tasks) and by a data server (that is where the input and output are staged). On the other hand, the client is responsible for managing the running tasks and help with the communication with the central server. The normal worklflow for the client runtime can be described

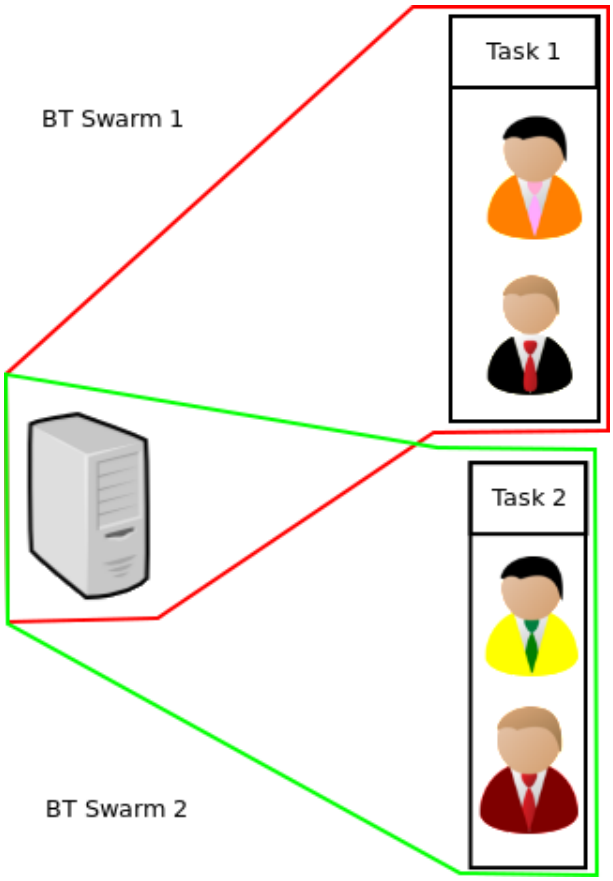


Figure 3.1: BitTorrent Swarms. Central server on the left, volunteers on the right.

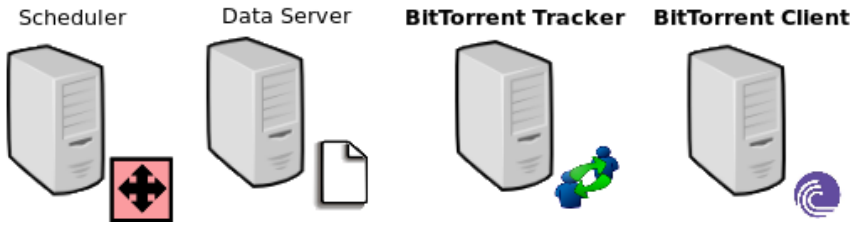


Figure 3.2: Server-Side Architecture

as: downloading input, starting the task, and uploading the output.

3.1.1 Using BitTorrent as Data Distribution Protocol

BitTorrent is a peer-to-peer protocol that allows nodes around the world to exchange data. In our project, we use it do spread input and output data related to executing tasks. Using BitTorrent, each volunteer and central server will become part of a swarm (see Figure 3.1), a network of nodes that share a particular file (the input or output files in our case). This way, each volunteer might download files from multiple sources (therefore alleviating the burden on the central server).

In order to introduce BitTorrent as the data distribution protocol, two new entities were added to the central server (see Figure 3.2): a BitTorrent Tracker and a BitTorrent Client. As previously described in Section 2.4.7, the tracker is essential to provide information so that new nodes are able to find nodes

that have a particular file. The BitTorrent client (which is also added to the client runtime component) is responsible for handling the BitTorrent protocol and all the necessary communication to perform the transfers.

Once the components are placed both on the server and client sides of freeCycles, `.torrent` files can be used as input and output of normal tasks. I.e., instead of sending normal data, both server and client exchange `.torrent` files. These files (which are tiny) are then used by the BitTorrent client to perform the real peer-to-peer transfer.

The validation step is also changed. In spite of downloading all outputs from all volunteers, the server can now download the `.torrent` files only, that have a hash of the real task output, and validate them. When some `.torrent` file is validated, it is sent to the BitTorrent client (at the server) that will therefore download the real output file.

3.1.2 Supporting MapReduce Jobs

The MapReduce programming model is usually provided as a framework, i.e., the programmer extends or overrides some functionality but the execution is controlled by the framework. To introduce MapReduce in freeCycles we used the same approach.

From the server point of view, freeCycles uses normal tasks but redefines some steps, namely the task creation and task validation. Task creation had to be modified to introduce different replication factors for map and reduce tasks (more details on Section 3.3). Task validation was improved to detect:

- when all map tasks of a specific job are complete and validated. Once such situation happens, the validator triggers the shuffle phase;
- when all reduce tasks of a specific job are complete and validated. Once this situation happens, the reducers' output (`.torrent` files) are passed to the BitTorrent client, on the server, to start downloading the real job's outputs.

The shuffle phase is a necessary step in the MapReduce workflow. As each mapper might produce output to every reducer, there is the need to organize the inputs for the reduce tasks. Figure 3.3 shows this operation. This operation is performed by the central server once it detects that all map tasks are validated. It is important to note that this operation manages `.torrent` files only. This way, the shuffle phase is extremely fast and the scalability of the system is not compromised. Once this operation is finished, new reduce tasks can be delivered to new volunteers.

With respect to the client side, we augmented the already available runtime to provide similar functionality to what is usually offered by typical MapReduce implementations, such as Hadoop. The main steps of the algorithm run by the client can be described as follows:

```
1. Download input using the BitTorrent protocol
2. Load the input into sorted Key-Value pair structures
3. If (task is a map):
4.   For all Key-Value pairs:
```

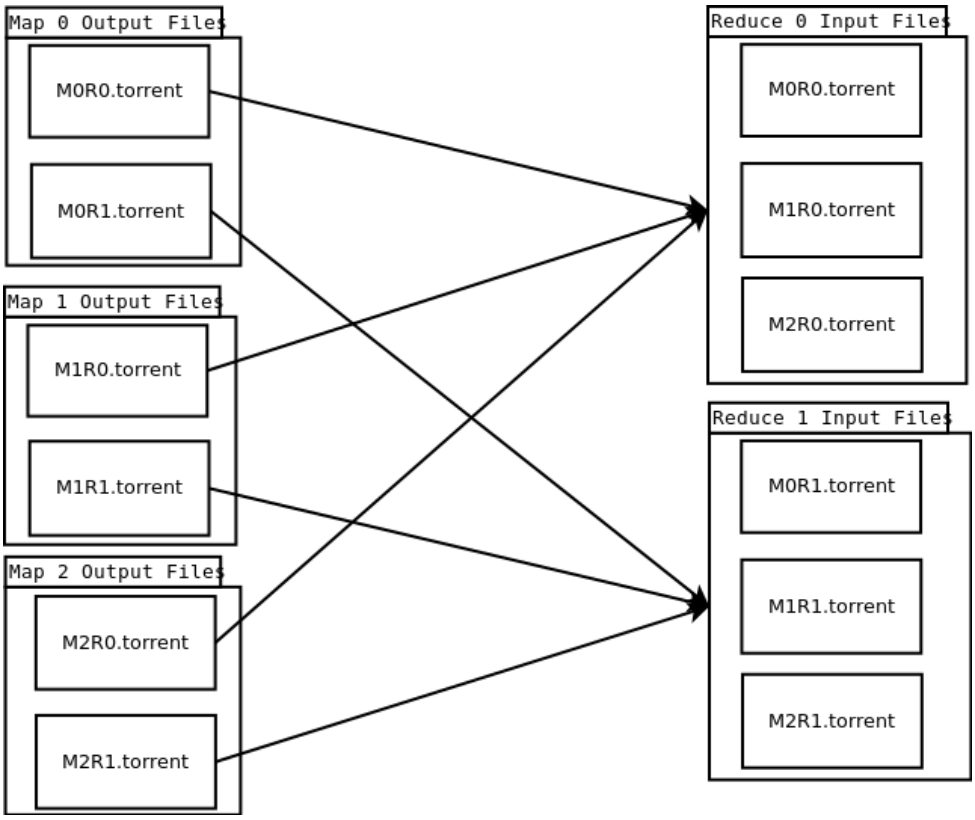


Figure 3.3: Shuffle Phase Example

```
5.     call the user defined map method
6.   create a .torrent file with the specific intermediate data for each reducer
7. Else:
8.   For all Key-Value pairs:
9.     call the user defined reduce method
10.   create a .torrent file with the output
11. Pass resultant .torrent files to the BitTorrent client
```

3.1.3 MapReduce Jobs with Multiple Cycles

Using freeCycles, it is possible to run multiple MapReduce jobs with multiple cycles each. To provide multiple cycles we had to update the client runtime and the validator.

The client was modified to split the reducer output into N splits (N being the number of mappers for the next cycle). Instead of producing only one output file, each reducer creates N smaller output files.

The validator (server side) was changed to trigger an operation, very similar to the shuffle operation, that will prepare the input for the map tasks of the next cycle. This operation is performed when all reduce tasks are validated and the MapReduce job needs more cycles.

Using the aforementioned improvements, freeCycles is able to run multiple MapReduce cycles very fast, data between cycles does not need to go to the central server, it flows directly from reducers to mappers.

3.2 Data Distribution Algorithm

Having described the components on (client-side) volunteers and on the server, we now detail how we use the BitTorrent file sharing protocol to coordinate input, intermediate and final output data transfers. Still on our data distribution algorithm, we show how freeCycles is able to run multiple MapReduce cycles without compromising its scalability (i.e. avoiding high burden on the data server).

3.2.1 Input Distribution

Input distribution is the very first step in every MapReduce application. Input must be split over multiple mappers. To do so, each mapper downloads a `.torrent` file pointing to the corresponding input file in the central data server.

For each input file, the server plays as initial seed (the origin). If we take into consideration that each map task is replicated over at least three volunteers (for replication purposes), then, when a new map task begins, the volunteer will have at least one seed (the data server) and, possibly up to the task replication factor minus one, additional volunteers sharing the file (each volunteer shares all the input file chunks, that it has, using the BitTorrent protocol).

Therefore, we can leverage the task replication mechanisms to share the burden of the data server. Even if the server is unable to respond, a new mapper may continue to download its input data from other mappers. The transfer bandwidth will also be bigger since a mapper may download input data from multiple sources.

Input data distribution is done as follows (see Figure 3.4):

1. a new volunteer requests work from the central server scheduler and receives a workunit;[1]
2. the new mapper downloads the `.torrent` file[2]from the data server (a reference of the `.torrent` file is inside the workunit description file);
3. the new mapper contacts the BitTorrent tracker to know about other volunteers sharing the same data;
4. the volunteer starts downloading input data from multiple mappers and/or from the server.

3.2.2 Intermediate Output Distribution

Once a map task is finished, the mapper has an intermediate output ready to be used. The first step is to create a `.torrent` file. From this point on, the mapper is able to share its intermediate data using the BitTorrent protocol: the BitTorrent client running at the volunteer node automatically informs the BitTorrent tracker, running at the central server, that some intermediate files can be accessed through

[1] A workunit is a concept used in BOINC to refer a computational task that is shipped to volunteer resources. A workunit contains all the information needed to run a task.

[2] A .torrent file is a special metadata file used in the BitTorrent protocol. It contains several fields describing the files that are exchanged using BitTorrent. A `.torrent` file is unique for a set of files to be transfered (since it contains a hash of the files).

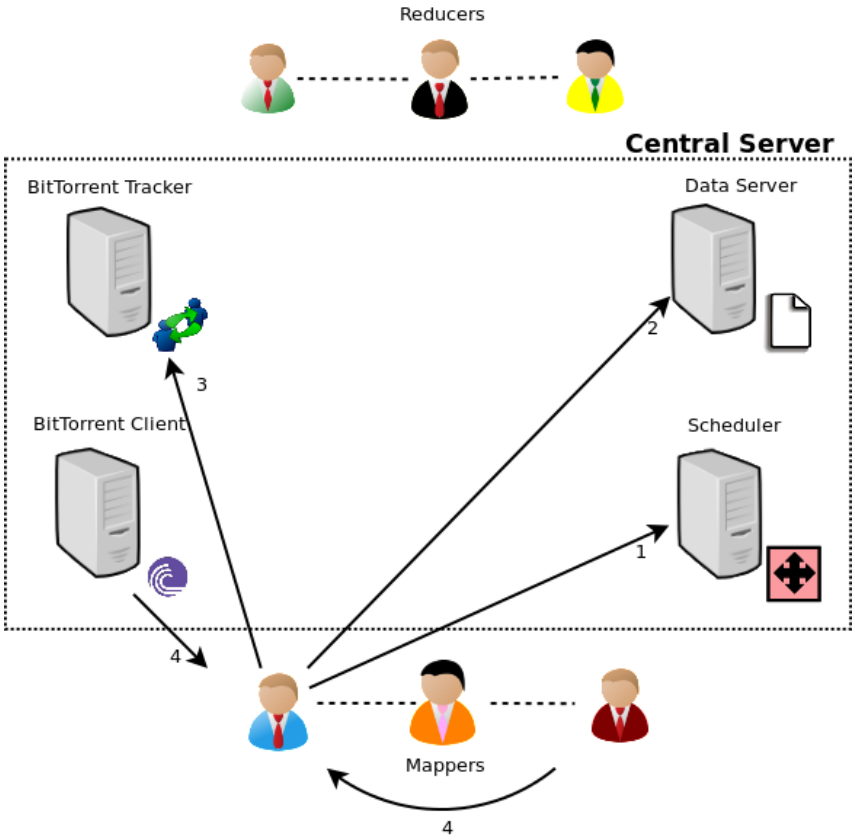


Figure 3.4: Input Data Distribution

this volunteer . The second step is to make the server aware of the map task finish. To this end, the mapper contacts the server and sends the `.torrent` file just created for this intermediate output.

As more `.torrent` files arrive at the server, the server is able to decide (using a quorum of results) which volunteers (mappers) have the correct intermediate files comparing the hashes (that came inside the `.torrent` files). When all the intermediate outputs are available, the server shuffles all `.torrent` files and prepares sets of inputs, one for each reduce task. When new volunteers request work, the scheduler starts issuing reducer tasks. These reducer tasks contain references to the `.torrent` files that were successfully validated and that need to be downloaded. Once a reducer has access to these `.torrent` files, it starts transferring the intermediate files (using the BitTorrent protocol) from all the mappers that completed the map task with success. Reduce tasks start as soon as all the needed intermediate values are successfully transfered.

Figure 3.5 illustrates the steps for the intermediate data distribution:

1. the mapper computes a `.torrent` file for each of its intermediate outputs;
2. the mapper informs the central BitTorrent tracker that it has some intermediate data ready to share;
3. a message acknowledging the map task finish and containing the computed intermediate output hashes (`.torrent` files) is sent to the central scheduler;
4. the server validates and shuffles all intermediate output hashes;

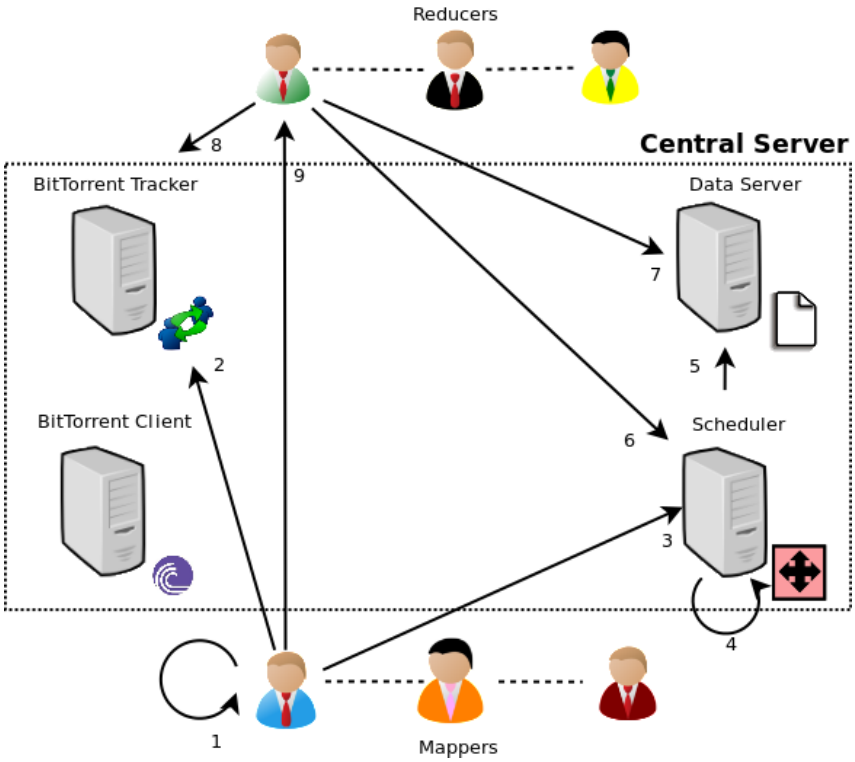


Figure 3.5: Intermediate Data Distribution

5. then all `.torrent` files are stored in the data server;

6. when a new volunteer (reducer) asks for work, a workunit is delivered with references to several `.torrent` files (one for each intermediate output);

7. the reducer downloads the `.torrent` files from the data server;

8. it then contacts the BitTorrent tracker to know the location of mapper nodes that hold intermediate data;

9. the reducer uses its BitTorrent client to download the intermediate data from multiple mappers.

3.2.3 Output Distribution

Given that map and reduce tasks are replicated over at least three volunteers, it is possible to accelerate the upload of the final output files from the reducers to the data server.

The procedure is similar to the one used for intermediate outputs. Once a reduce task finishes, the reducer computes a `.torrent` file for its fraction of the final output. Then, it informs the BitTorrent tracker that some output data is available at the reducer node. The next step is to send a message to the central scheduler containing the `.torrent` file and acknowledging the task finish. Once the scheduler has received enough results from reducers, it can proceed with validation and decide which `.torrent` files will be used to download the final output. All the trustworthy `.torrent` files are then used by the BitTorrent client at the central server to download the final output.

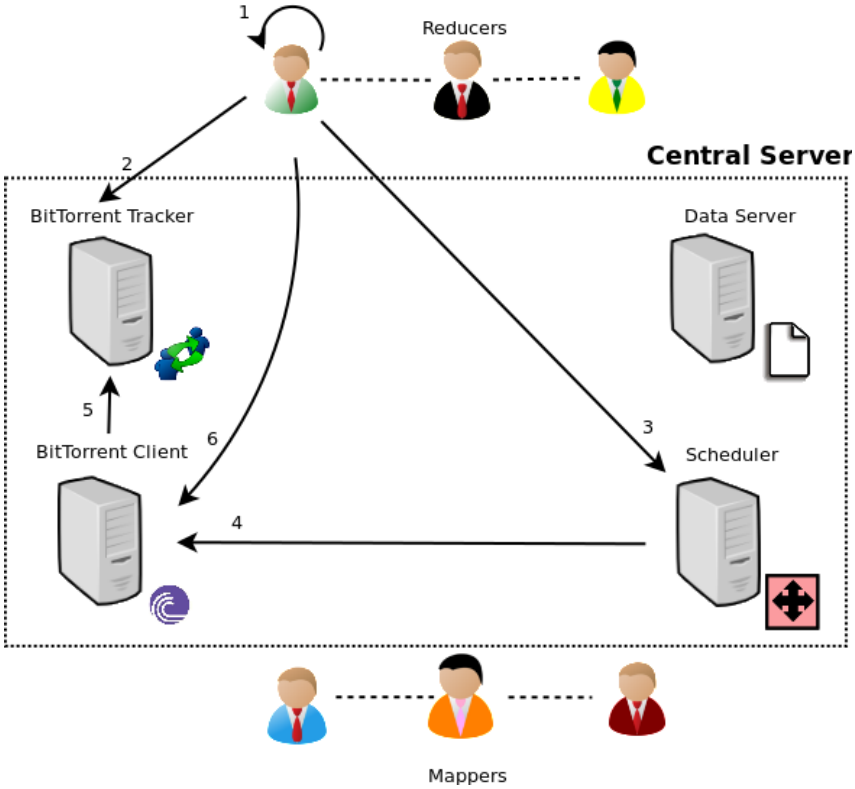


Figure 3.6: Output Data Distribution

Using BitTorrent to transmit the final outputs results in a faster transfer from volunteers to the data server, a lower and shared bandwidth consumption from the volunteer's perspective, and an increased fault tolerance (since a volunteer node failure will not abort the file transfer).

Output data distribution works as follows (see Figure 3.6):

1. `.torrent` file is generated for the fraction of the final output;
2. the reducer informs the BitTorrent tracker that it has some data to share;
3. a message is sent to the central scheduler acknowledging the task finish and reporting the `.torrent` file;
4. the central server is able to validate results and gives some `.torrent` files to the BitTorrent client at the server;
5. the BitTorrent client contacts the BitTorrent tracker to know the location of the reducers;
6. final output files are downloaded from multiple volunteer nodes.

3.2.4 Multiple MapReduce Cycles

Using the data distribution techniques just described, where the central server and all volunteers have a BitTorrent client and use the BitTorrent Tracker to find peers with data, it is very easy to use freeCycles for running applications that depend on multiple MapReduce cycles. The difference between our solution

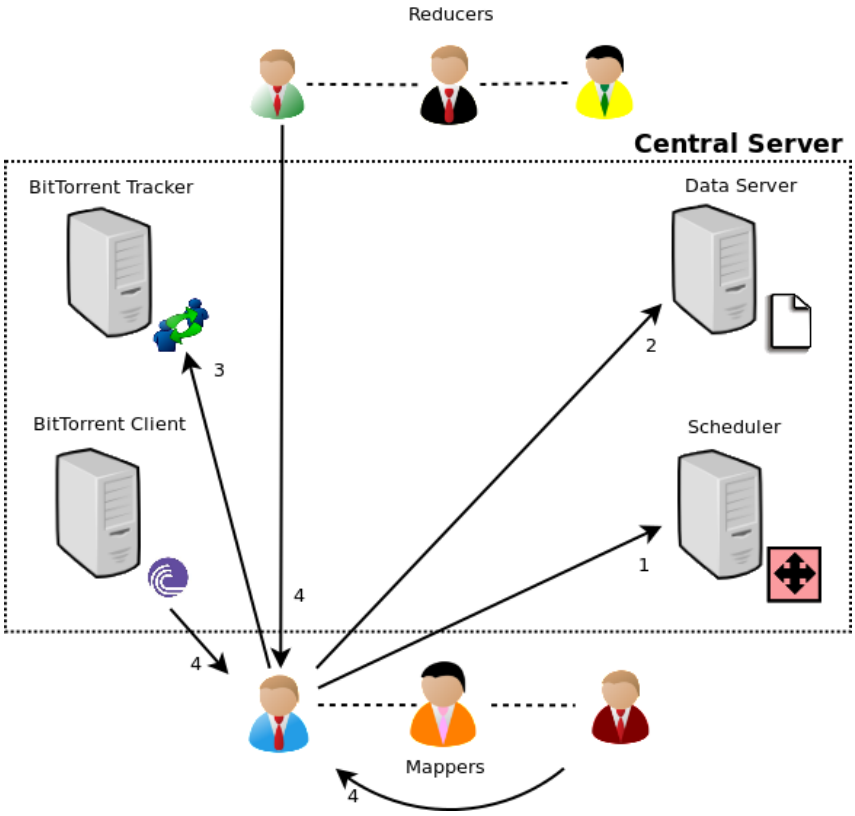


Figure 3.7: Data Distribution Between MapReduce Cycles

and previous ones (namely SCOLARS [14]) is that output data need not to go to the central server before it is delivered to new mappers (i.e., data can flow directly from reducers to mappers from one cycle to the other).

From mappers's perspective, when a workunit is received, volunteers ask the BitTorrent tracker for nodes with the required data. It does not differentiate between the single cycle scenario (where the volunteer downloads from the central server) or the multiple cycle scenario (where the volunteer downloads from reducers of the previous cycle). Regarding the central server's perspective, the scheduler only needs to know that some map tasks depend on the output of some reduce tasks (more details in the next section).

Figure 3.7 shows how volunteers and server interact to feed the new cycle with the output from the previous one (assuming that all steps in Figure 3.6 are finished):

1. when new volunteers ask for work, the scheduler delivers new map tasks with references to the `.torrent` files sent by the reducers (of the previous cycle);
2. the new mappers download the `.torrent` files from the data server;
3. after retrieving the `.torrent` files, the mappers ask the BitTorrent tracker for nodes that are sharing the output data;
4. mappers from the next cycle can now download output data from multiple reducers (from the previous cycle).

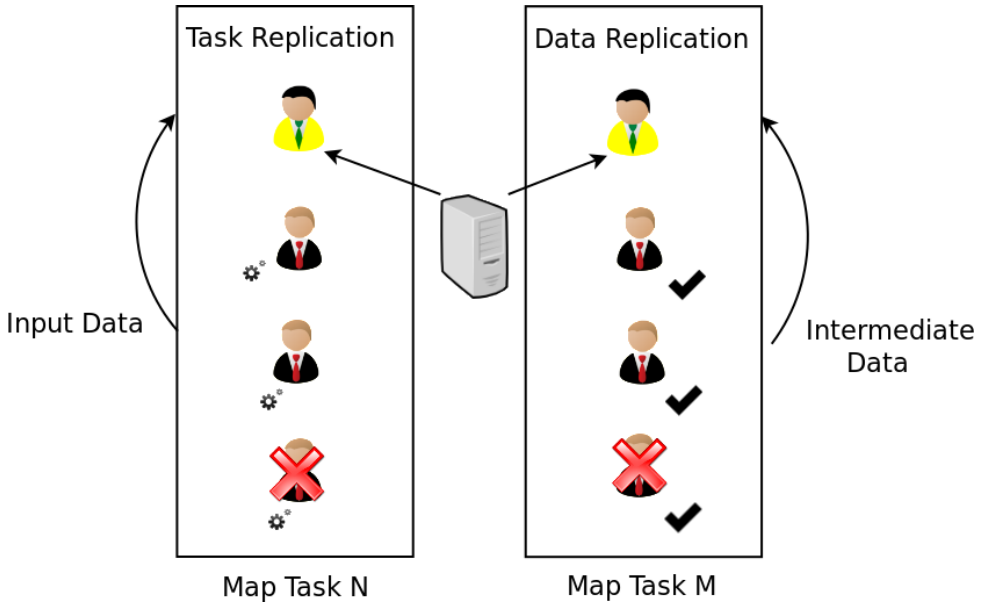


Figure 3.8: Task and Data Replication

3.3 Availability of Intermediate Data

Previous studies [22] show that the availability of intermediate data is a very sensitive issue for programming paradigms like MapReduce. Note that, when using embarrassingly parallel applications, there is no intermediate data and therefore this problem does not apply.

The problem is that, for performance reasons, typical MapReduce implementations (targeted to clusters) do not replicate intermediate results. However, when applied (MapReduce) to volunteer computing, where node churn is very high, such lack of replication leads to a loss of intermediate data. It was shown that loosing a single chunk of intermediate data incurs into a 30% delay of the overall MapReduce execution time. To cope with this problem, freeCycles presents two methods:

1. replicate map tasks aggressively when volunteers designated to execute a particular map task take too long to report results. By imposing a shorter interval time to report to the central server, we make sure that we keep at least a few replicas of the intermediate output;

2. replicate intermediate date when there are intermediate outputs that have already been validated (by the central server) and some of the volunteers that reported these results take too long to report. Therefore, volunteers might be used to replicate intermediate data to compensate other mappers that die while waiting for the reduce phase to start. These tasks would simply download `.torrent` files and use them to start downloading intermediate data. Once the reduce phase starts, these new volunteers can also participate in the intermediate data distribution phase, just like the other volunteers that performed the map tasks. Replicating only the intermediate output is much faster than replicating a map task since: i) the computation does not have to be performed; ii) intermediate data is normally smaller than input data. These two methods are applied before the reduce stage starts.

Figure 3.8 shows the two possible scenarios. On the left, one working volunteer fails. As soon

as a new volunteer asks for work, the scheduler (central server) will assign a map task replica (task replication). On the other hand, on the right, a volunteer fails after finishing its map task, whose output is already validated. In this situation, it is much faster to recover the failure replicating only the resultant intermediate data.

To finish, it is important to notice that if a map task is not already validated, it is not be safe to replicate the intermediate output. If there is some intermediate output available (but not validated), replicating it would possibly replicate erroneous data which would get to wrongly validated data.

3.4 Summary

To finish this chapter we give a small summary of freeCycles. freeCycles is a VC platform that promises to improve the data distribution and data availability that are not optimized, in current platforms, for MapReduce workflows.

freeCycles uses BOINC, a successful VC platform as base platform and uses the BitTorrent protocol to improve data distribution, harnessing volunteers' upload bandwidth to distribute data. Intermediate data availability is also improved by replicating both tasks and data.

Chapter 4

Implementation

In this section we explain how freeCycles is implemented and how to use our VC solution to create and distribute MapReduce applications over a volunteer pool. We further present a detailed description of all its components and how they cooperate with each other.

During this project, we tried to keep the implementation simple, effective, and reusable. Right from start, when we thought about how to implement freeCycles, in particular, how to use BitTorrent for data distribution, two alternatives were present:

- use a stable BOINC release and change its core, namely the data communication protocol. The default protocol implementation is provided by libcurl[1]. To successfully distribute data using BitTorrent, we would have to rewrite all functions calls to libcurl by some BitTorrent client library. These modifications would have to propagate to client and server sides;

- keep all changes at a project level. A BOINC server can host different projects at the same time. Each project can reimplement several functionalities (which we will detail later). By redefining several functionalities, it is possible to use BitTorren to distribute data.

Analysing both solutions, we concluded that the first one would bring four big disadvantages:

- in order to keep all code consistent and working, the amount and complexity of code to change would be huge;

- the solution would not be reusable, i.e., reusing our modifications in future versions of BOINC would be impossible;

- it would not let a single BOINC server host both BitTorrent (freeCycles) and non BitTorrent (regular BOINC) projects;

- in order to contribute, each volunteer would have to use our specific versions. Volunteers with regular BOINC installations would not be able to participate in projects hosted by freeCycles.

[1] cURL is a multi protocol file transfer library. cURL's website can be found at http://curl.haxx.se/

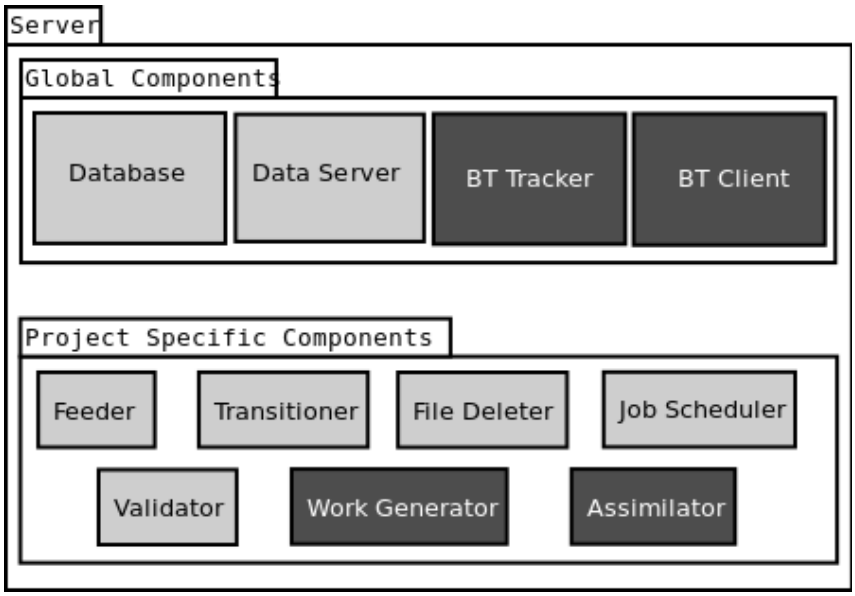


Figure 4.1: freeCycles Server-side Implementation

Therefore, in order to keep our code reusable for future VC application developers, and compatible with current BOINC versions, we decided to implement freeCycles right on top of BOINC, as a project. In fact, assuming a working BOINC server, to start running MapReduce jobs on volunteer nodes and using BitTorrent to distribute data, one only has to:

1. create a BOINC project. This is an automated task, a project owner only has to use a script provided by BOINC and answer some simple questions about the project (the name, for example);
2. replace and/or extend the provided implementations for some of the project specific daemons (work generator and assimilator);
3. use and/or adapt the provided script to prepare input data and the project configuration file;
4. reuse and/or extend our sample MapReduce application and provide a Map and a Reduce function implementations;
5. start the project.

Although implemented as a project, freeCycles needs to change both sides, the server and the client. In the next sections of this chapter we will provide a detailed description of how freeCycles is integrated with the already existing environment.

4.1 Server

freeCycles uses a central server to coordinate task delivery. Each project is tied to a single server but a server can host multiple projects. Each server can be described as being composed by two types of components: i) global components, shared between projetcs, and ii) project specific components.

In order to implement freeCycles, we introduced several global and project specific components. Figure 4.1 presents a graphical representation of a freeCycles server. Light gray boxes are the components inherited from BOINC (and therefore, remain unchanged). Dark gray boxes represent the components that were included (in the case of the BitTorrent client and BitTorrent tracker) or modified (work generator and assimilator) by our solution. We now proceed with a detailed description of these components and how they cooperate with each other.

4.1.1 BitTorrent Tracker

A BitTorrent tracker is a central entity in any BitTorrent swarm (network of nodes sharing a particular file). The tracker is responsible for holding information that helps nodes to find other nodes sharing a particular file.

The tracker is introduced as a global component, meaning that multiple projects using BitTorrent can used it at the same time. We have not programmed a new BitTorren tracker, but instead, we used a popular open source implementation, opentracker[2]. opentracker revealed to be a simple, lightweight, and efficient BitTorrent tracker.

4.1.2 BitTorrent Client

Since we decided not to change BOINC's core implementation, we implemented a small BitTorrent client and placed it as a global component, available for any project using BitTorrent. Our implementation uses a very efficient and scalable BitTorrent implementation library, libtorrent[3].

Our BitTorrent client is responsible for periodically check a particular folder for `.torrent` files and download the corresponding files and share them afterwords. Using this implementation, projects simply need to drop `.torrent` files and periodically check if the wanted file is already there.

4.1.3 Auxiliary Scripts

freeCycles provides two auxiliary scripts, one that helps preparing a MapReduce job and one to perform the shuffle operation.

The job setup script performs several steps:

1. given the number of mappers, reducers, and cycles, it creates a job configuration file, assigning ids to all tasks, and presenting paths to all inputs and outputs. This file is to coordinate both the work generator and the assimilator;
2. splits the input in equally sized chunks, one for each mapper;
3. creates a `.torrent` file for each chunks;
4. stages all chunks and respective `.torrent` files on the BitTorrent client folder.

[2]opentracker can be found at https://erdgeist.org/arts/software/opentracker/. opentracker is currently being used by one of the world's largest tracker sites, The Pirate Bay.

[3]libtorrent can be found at http://www.rasterbar.com/products/libtorrent/.

The shuffle script is responsible for executing the shuffle step. This step, as described back in Section 3.1.2, performs many small file operations. This script also uses the job configuration file to know hot to organize intermediate data into reducers' input.

4.1.4 Work Generator

The work generator is a project specific module. As the name suggests, its main purpose is to create tasks for volunteers. In order to know which tasks to deliver, the work generator keeps, in memory, all MapReduce jobs' state (this state is first read from the job configuration file). All changes in the state of each MapReduce job are propagated to the configuration file, marking which tasks are delivered.

This is also the module responsible for keeping tasks replicated. If some task gets too much time to be validated, the work generator will replicate the tasks, delivering a replica as soon as a volunteer asks for work (more details on Section 3.3).

4.1.5 Assimilator

This last server side component is the one that assimilates several task results and prepares the task output. In other words, this module performs several operations after a tasks is validated. In our scenario we use this module to trigger the shuffle phase.

The assimilator also keeps an in memory data structure about the MapReduce jobs' state. It also updates it when the shuffle phase is triggered. After the state file is updated with the shuffle phase, the work generator will start delivering reduce tasks.

This is also the component that stages the final output, i.e., places the output files on their final location. This is specially important for the cooperation between the work generator and the assimilator. Both components respect the paths described on the job configuration file and therefore, all input and output files can be easily reached using the paths described on it.

4.2 Client

Using freeCycles, all volunteer nodes run the client software which is responsible for: i) performing the computation (map or reduce task), and ii) sharing its input and output data (which might be input, intermediate, or final output data) with all other volunteers and possibly, the central server.

To be and remain compatible with current BOINC clients, our project is implemented as a regular BOINC application. Therefore, all volunteers will be able to join a MapReduce computation without upgrading their VC software. If it would not be a regular application, volunteers would have to upgrade their client software in order to fully explore freeCycles' capabilities namely, use BitTorrent to share files. Previous solutions (e.g. SCOLARS) do not use this approach and modify the BOINC client. Therefore, it cannot be used without forcing users to upgrade their client software.

freeCycles' client side application is meant to be used as a framework, i.e., developers would simple call freeCycles' code to register the map and reduce functions. All other issues related to managing map

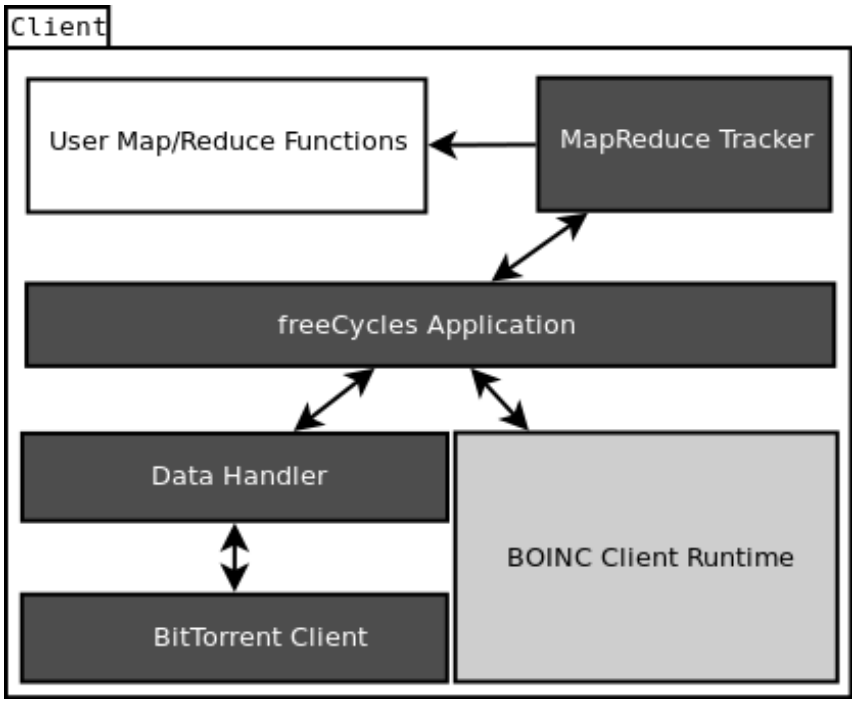


Figure 4.2: freeCycles Client-side Implementation

and reduce task execution, downloading and uploading data is handled by our implementation.

Notwithstanding, VC application developers might analyse and adapt the application code to specific application needs (e.g. if one needs to implement a special way to read/write input/output data). Other optimizations like intermediate data partitioning or combining intermediate results might be as well easily implemented.

Figure 4.2 presents a graphical representation of the freeCycles client implementation. The light gray box, BOINC Client Runtime, is the core component in the client software. All volunteers that are already contributing using BOINC will have this component and it is, therefore, our only requirement. Dark grey boxes are the components offered by our project. We now proceed with a detailed description of these components, the ones offered by our project.

4.2.1 freeCycles Application

freeCycles Application is the central component and the entry point of our application. It coordinates the overall execution by: i) asking for input from the Data Handler, ii) preparing and issuing the MapReduce computation and iii) sending output data (via the Data Handler); Additionally, this component is responsible for interacting with BOINC Client Runtime (initialize BOINC runtime, obtain task information and acknowledging the task finish);

4.2.2 Data Handler

Data Handler is the data management API. It is the component responsible for downloading and uploading all the necessary data and is the only one that needs to interact with the communication protocol

(BitTorrent). This API, however, does not depend on the communication protocol and therefore, can be used for multiple protocols. In fact, for debug purposes, we implemented this API for BitTorrent and for the local file system (to test the application locally).

4.2.3 BitTorrent Client

The BitTorrent Client is the low level and protocol specific component. It was implemented using an open source BitTorrent library (libtorrent). This client uses a specific directory to place the downloaded files (the ones that are being shared via BitTorrent).

As one volunteer might be running several tasks at the same time, we force all BitTorrent clients (one for each task) to use the same download directory. This way, it is possible to reuse data from one task to the other (if it happens that a client receives a task that needs output from other, one just completed on the same node). In order to have multiple multiple BitTorrent clients we had to configure libtorrent to accept multiple connections from per node, i.e., one client can connect to multiple clients on the IP address.

It is also important to note that the scheduler at the server will never assign two replicas replicas of the same task to the same node. Therefore, there is no problem in multiple BitTorrent clients sharing the same download directory since they will never download the same file at the same time.

The final note about the BitTorrent client is about how we create `.torrent` files. These files are created by multiple volunteers and their content has to match (for validation purposes), all fields have to have the same value. As described in the BitTorrent protocol, each `.torrent` file contains a set of key value pairs (creator, date of creation, hash of the file, number of bytes of the file, ...). For the validation process succeed, we had to force all fields to have the same values. For example, for the data of creation, we used the value zero, that represents the first of January of 1970.

4.2.4 MapReduce Tracker

The MapReduce Tracker introduces the logic related to MapReduce applications. It uses a previously registered function to run map or reduce tasks and manages all key and value pairs needed for the computation.

For test purposes, we did not use any combining operation[4]. Combining map output keys would reduce the size of the intermediate data and therefore would not help demonstrating our data distribution improvements. Nevertheless, a VC application developer might easily program and use a combiner in freeCycles. The developer would only need to create a function that receives a key value data structure, containing intermediate keys, and use the function to i) produce a new structure or ii) change the existing structure.

To create and use one application, one would only need to provide a map and a reduce function implementations (white box from Figure 4.2). These functions are then registered in our MapReduce Tracker module and thereafter, are called for all keys and value pairs.

[4]Combining is a MapReduce optimization technique that processes Map output keys to filter or aggregate them, therefore, lessening the number of intermediate keys to process by the reducers.

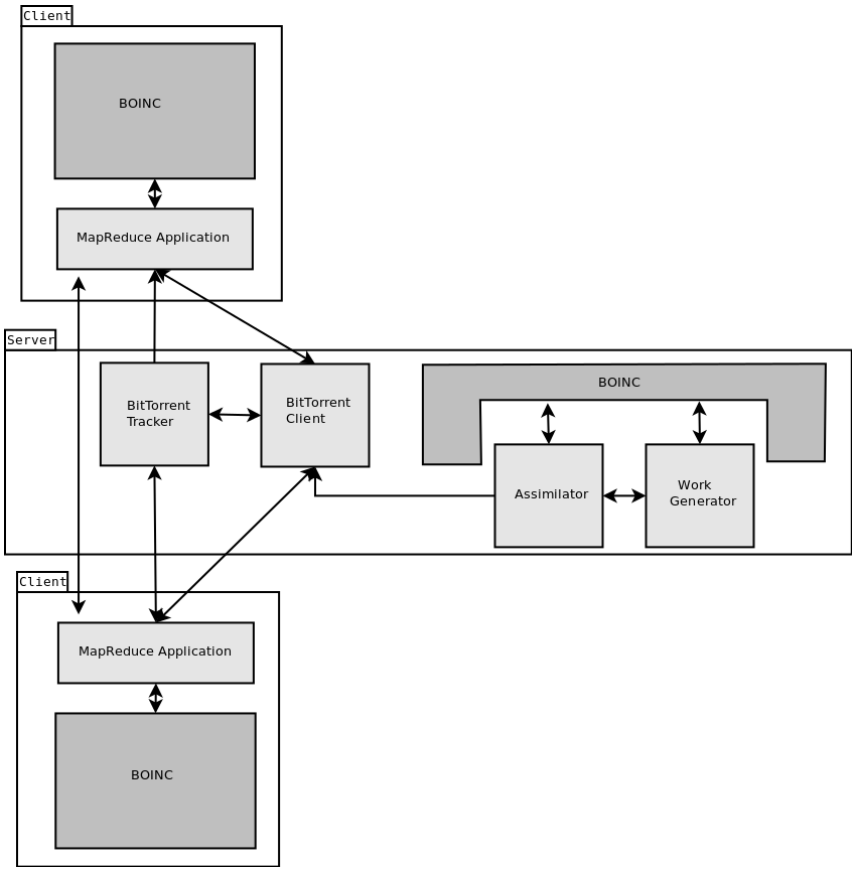


Figure 4.3: Integration of freeCycles with BOINC

The client side application code is divided into two modules: the Data Handler (handles all data downloading and uploading), and the MapReduce Tracker (controls the map or reduce task execution). By using such division of responsibilities, we provide project developers with modules that can be replaced (e.g. one could decide to use our Data Handler implementation with BitTorrent to distribute data in other types of computations, not just MapReduce).

4.3 freeCycles Integration with BOINC

Having described all the components introduced by freeCycles, we now give a global overview about how these components interact with the already existing BOINC components.Figure 4.3 shows the all the connections between the introduced and modified components and BOINC.

The communication between the new components and BOINC is done in two places: i) at the client, where the BOINC run time interacts with our MapReduce application, and ii) in both the assimilator and work generator, components that are called by the BOINC scheduler.

The remaining connections in Figure 4.3 are between new components. All MapReduce applications (which contain a built-in BitTorrent client) contact each other, the BitTorrent tracker and the BitTorrent client at the server. Finally, the work generator communicates with the assimilator via the job configuration file (described in Section 4.1.5) and with the BitTorrent client (to start the download of the final

output).

4.4 Summary

Throughout this chapter we discussed freeCycles' implementation. We started with some design decisions and then we showed how freeCycles can be used to create and distribute a MapReduce job over the Internet. Then we moved into a detailed description of all the components that freeCycles offered and ended on how these components interact with the existing BOINC platform.

To conclude our discussion about our implementation, we emphasize, once more, that freeCycles tries to maintain the implementation as simple and modular as possible. We hope that this way we enable future utilizations of this work.

Chapter 5

Evaluation

Our evaluation focuses three different aspects, protocol evaluation, platform evaluation, and replication (when nodes fail) evaluation. Therefore, this chapter is organized in three different sections, each one concerning a different aspect:

- Data Distribution Protocol Evaluation - data protocols found in BOINC [4] (HTTP), SCOLARS [14] (FTP) and freeCycles (BitTorrent) are compared with simulated MapReduce data transfers. This section emphasises the performance and scalability wise superiority of the BitTorrent protocol versus its contenders;
- Volunteer Computing Platforms Evaluation - performance and scalability comparison, using several benchmark applications, of freeCycles versus BOINC, SCOLARS and Hadoop (cluster deployment);
- Task and Data Replication Evaluation - section dedicated to study the performance and scalability implications of data and task replication on a volunteer pool setting. Simulations used different node churn rates (number of nodes joining and leaving the network in a certain interval of time) to better simulate different real world environments.

5.1 Data Distribution Protocol Evaluation

In this section we present a performance comparison between the data distribution protocols used in BOINC (HTTP), SCOLARS (FTP), and freeCycles (BitTorrent) in the context of a MapReduce application. All three protocols were evaluated using a simulated MapReduce data flow comprehending three phases: i) Input Distribution: central server sends initial input to all mappers; ii) Intermediate Output Distribution: mappers send intermediate outputs to reducers; iii) Output Distribution: reducers send final output to the central server.

All tasks (maps and reduces) are replicated: map tasks with a replication factor of 5 (high replication factor to simulate aggressive replication in order to keep intermediate data available); reduce tasks with a replication factor of 3 (minimum for majority voting).

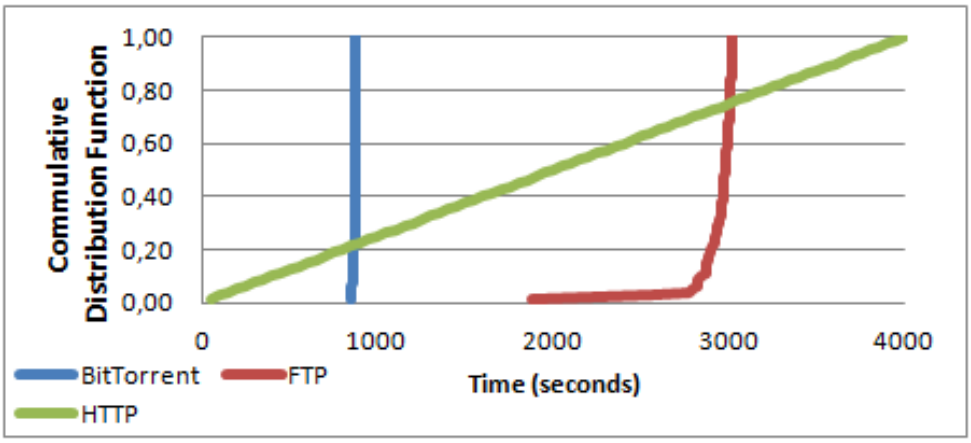


Figure 5.1: CDF for input distribution

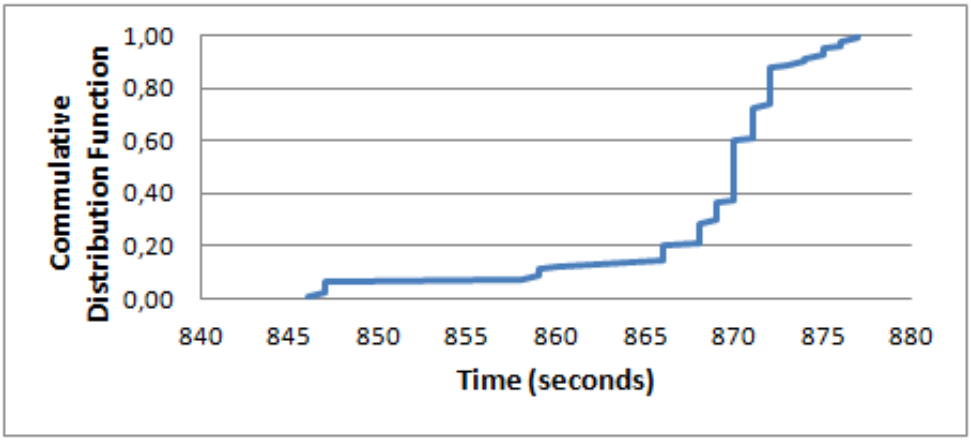


Figure 5.2: CDF for BitTorrent during input distribution

5.1.1 Evaluation Environment

We used a total of 92 nodes (92 cores with 2GB of RAM, scattered in several physical nodes): 80 mappers (16 different map tasks replicated 5 times) and 12 reducers (4 different reduce tasks replicated 3 times). To simulate Internet connection bandwidths, all nodes had their download and upload bandwidths limited to 100Mbps and 10Mbps respectively (except for the experiment in Figure 5.5). All the files (16 input files, 16 intermediate outputs and 4 final outputs) were different with 64MB each (except for the experiments in Figures 5.4 and 5.5).

5.1.2 BitTorrent versus HTTP versus FTP

Figure 5.1 presents the Cumulative Distribution Function (CDF) for all the three protocols during input distribution. It is clear that BitTorrent is able to distribute the same amount of data much faster than FTP and HTTP. From Figure 5.2 (that shows a zoomed region of Figure 5.1), we see why BitTorrent is able to finish all the transfers earlier: as soon as one mapper has some part of the file, it can participate in the input distribution. Therefore, the more mappers have the input file (or at least some parts), the faster it gets to other mappers download the file. We do not present the CDFs for intermediate output distributed

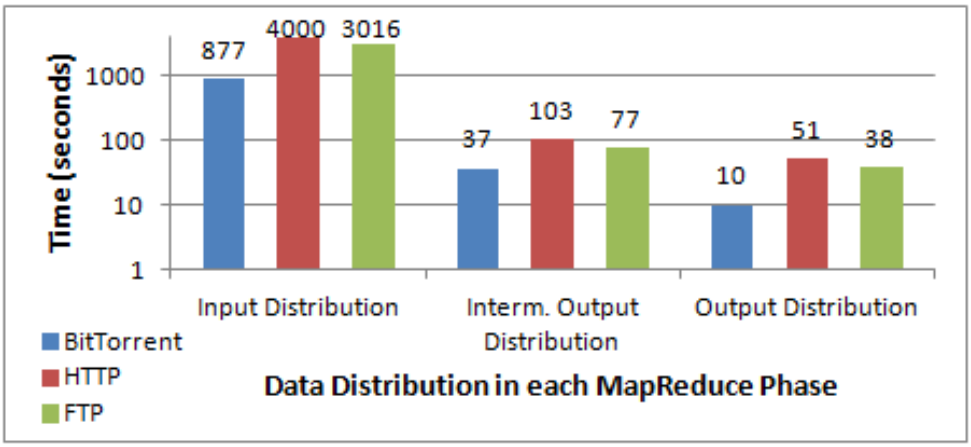


Figure 5.3: Performance evaluation of the protocols used in BOINC (HTTP), SCOLARS (FTP) and freeCycles (BitTorrent)

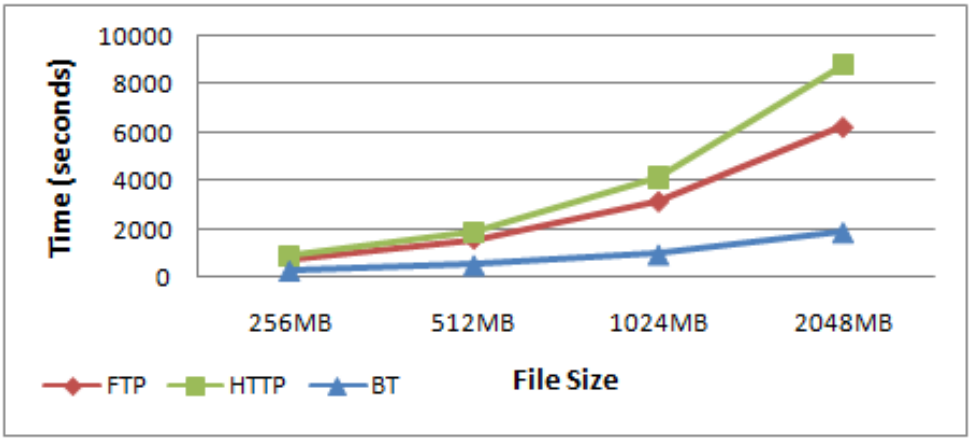


Figure 5.4: Performance comparison while varying the input file size

since results are similar.

From Figure 5.3, we can conclude that the BitTorrent protocol outperforms the other two protocols in all the data distribution phases. If we take into consideration that SCOLARS only uses FTP to distribute intermediate data, and still uses HTTP to distribute input and output data, we achieve a total of 4128 seconds spent distributing data. Looking at freeCycles, with BitTorrent distributing all data, we achieve a total of 924 seconds: freeCycles reduces the time spent data by 77,6% compared to SCOLARS.

5.1.3 Varying Input File Size and Upload Bandwidth

We further analysed the behaviour of the three protocols while varying: i) the file input size, and ii) the upload bandwidth (for a constant download bandwidth). In the first scenario (Figure 5.4) we repeated the experiment in Figure 5.3 for different file sizes. The second scenario (Figure 5.5) is similar to the first one but, instead of varying the file size, we used different upload bandwidths (keeping the download bandwidth at 100Mbps). For both simulations, we only present the sum of the time it took to distribute all data (input, intermediate and output). It is then possible to conclude that BitTorrent is more scalable while increasing the file size and decreasing the upload bandwidth (which is an important result since

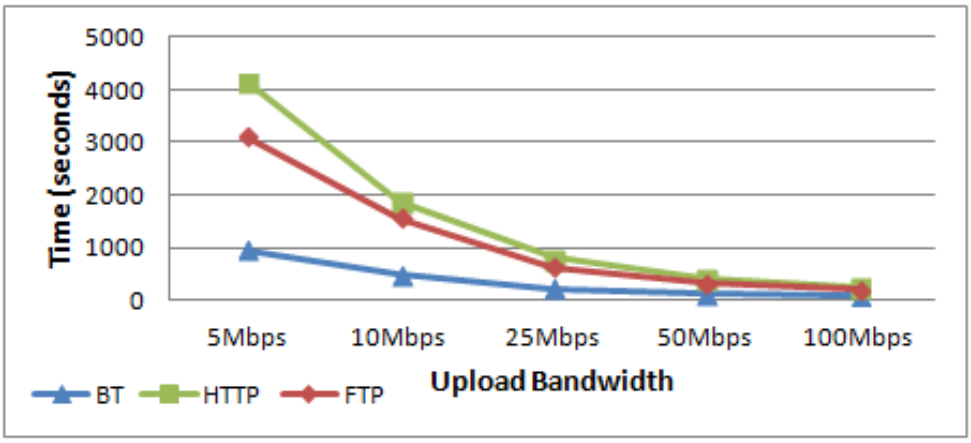


Figure 5.5: Performance comparison while varying the node upload bandwidth

ISPs provide much lower bandwidth for upload than for download).

5.2 Volunteer Computing Platforms Evaluation

We now proceed with an extensive evaluation of our system. We compare freeCycles with SCOLARS (a BOINC compatible MapReduce VC system), and BOINC, one of the most successful VC platforms, and therefore, a good reference for performance and scalability comparison. We use several representative benchmark applications and different environment setups that verify the performance and scalability of freeCycles.

5.2.1 Evaluation Environment

To conduct our experiments, we use a set of university laboratory computers. These computers, although very similar in software and hardware, are, most of the time, running tasks from local or remote users. It is also important to note that all tasks run by remote users have a higher niceness.[1] With this setting of priorities, our computations will compete with local user tasks (as in a real volunteer computing scenario).

In order to be more realistic, most of our evaluation was performed with throttled upload bandwidth. This is an important restriction since Internet Service Providers tend to limit users' upload bandwidth (the ratio between download and upload bandwidth usually goes from 5 to 10 or even more).

During this experimental evaluation, all MapReduce workflows used 16 map tasks and 4 reduce tasks both with a replication factor of 3. All map and reduce tasks run on different nodes (to simulate what would probably happen in a regular VC project). Thus, we used a total of 60 physical nodes (48 map nodes and 12 reduce nodes).

[1] The nice value is process attribute that increases or decreases the process priority for accessing the CPU. The higher this value is, the lower the priority is when deciding which process should run next.

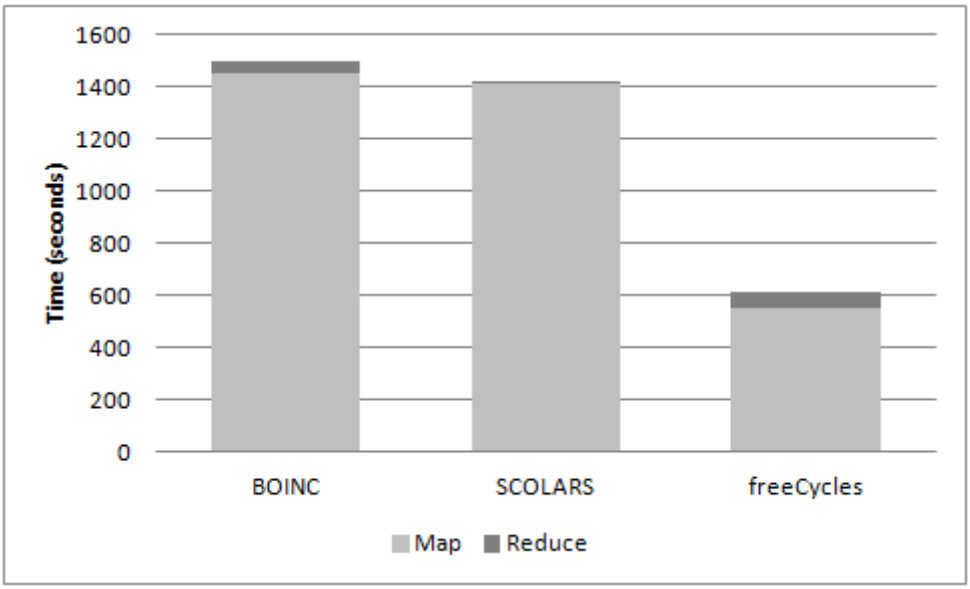


Figure 5.6: grep Benchmark Application

5.2.2 Application Benchmarking

For this dissertation, we use a set of representative benchmark applications to compare our solution with SCOLARS and BOINC.

From the data handling perspective (and according to some previous work [2]), each one of the three selected benchmarks belongs to a different MapReduce application class: small intermediate output (grep), medium intermediate output (word count) and large intermediate output (terasort).

While running these benchmarks, several variables are set: i) the MapReduce application was setup to 16 map tasks and 4 reduce tasks; ii) the replication factor is 3 (then we have 48 map nodes plus 12 reduce nodes); iii) all nodes have their bandwidth throttled to 10Mbps; iv) a 512MB data file was used as input. These parameters hold for every benchmark application and for all the three evaluated systems.

grep

Our first benchmark application is grep. As from the original Unix system, grep is a program that searches plain-text data for lines matching regular expressions. For this evaluation, we built a simple implementation of grep that was used to match a single word. The word was selected so that it was possible to have very small intermediate data.

From Figure 5.6, it is possible to see that freeCycles is able to run the benchmark application in less than half the time took by BOINC and SCOLARS. The application turnaround time is clearly dominated by the input distribution time. Since freeCycles uses BitTorrent, it uses available upload bandwidth from volunteers to help the server distributing the input.

A slight overhead can be noticed in the reduce phase for our solution. This comes from the fact that the intermediate output is so small that the time to distribute all the intermediate and final output was dominated by the BitTorrent protocol overhead, namely: contact the central tracker, contact several nodes, and wait in queues.

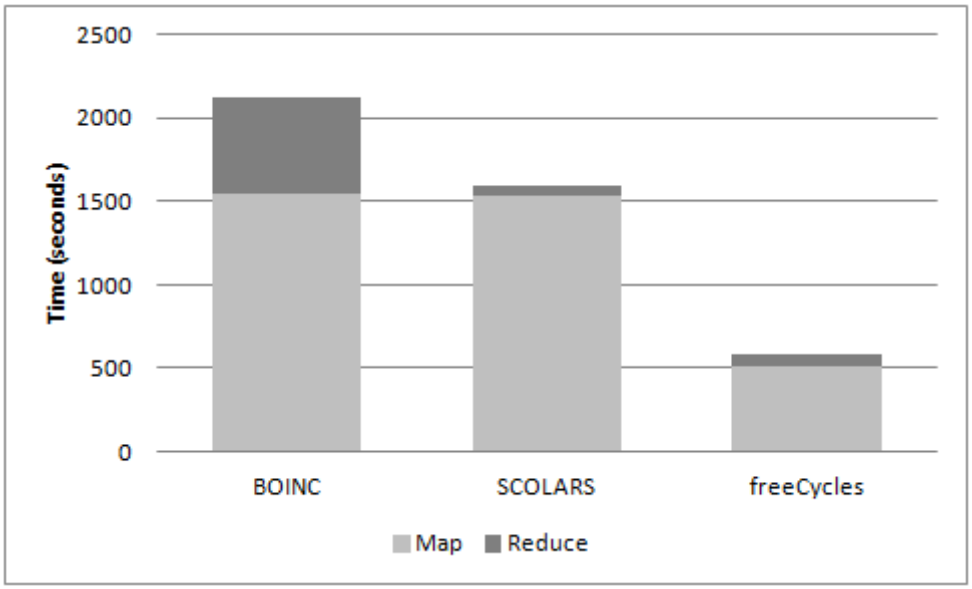


Figure 5.7: Word Count Benchmark Application

Word Count

Our second benchmark application is the famous word count. This program simply counts the number of occurrences for each work in a given input text. In order to maintain a reasonable size of intermediate data, we do combine (operation that merges intermediate data still on the mappers) output data from mappers.

Figure 5.7 shows the results of running word count.As intermediate data is larger than in the previous application (186MB of generated intermediate data versus 2MB, in grep), BOINC has a worse performance compared to SCOLARS and freeCycles. SCOLARS is able to be much more efficient than BOINC in the reduce phase (since it allows intermediate results to travel from mappers to reducers, avoiding the central server). Nevertheless, freeCycles continues to be the best system mainly by having a very small input data distribution time.

Terasort

The last benchmark application is Terasort. Terasort is yet another famous benchmark application for MapReduce platforms. At a very high level, it is a distributed sorting algorithm that: i) divides numbers in smaller chunks (with certain ranges), and ii) sorts all chunks. We produced a simple implementation of this algorithm to be able to compare the performance of freeCycles with other systems.

Despite being a very popular benchmark, this application is also important because it generates large volumes of intermediate and output data.

Looking at Figure 5.8, it is possible to see a big reduce phase in BOINC. This has to do with the large intermediate data generated by terasort. As SCOLARS implements inter-client transfers, it cuts much of the time needed to perform the intermediate data distribution (which is the dominating factor). As intermediate data is larger than in the previous application, our system suffered a slight increase in the reduce phase.

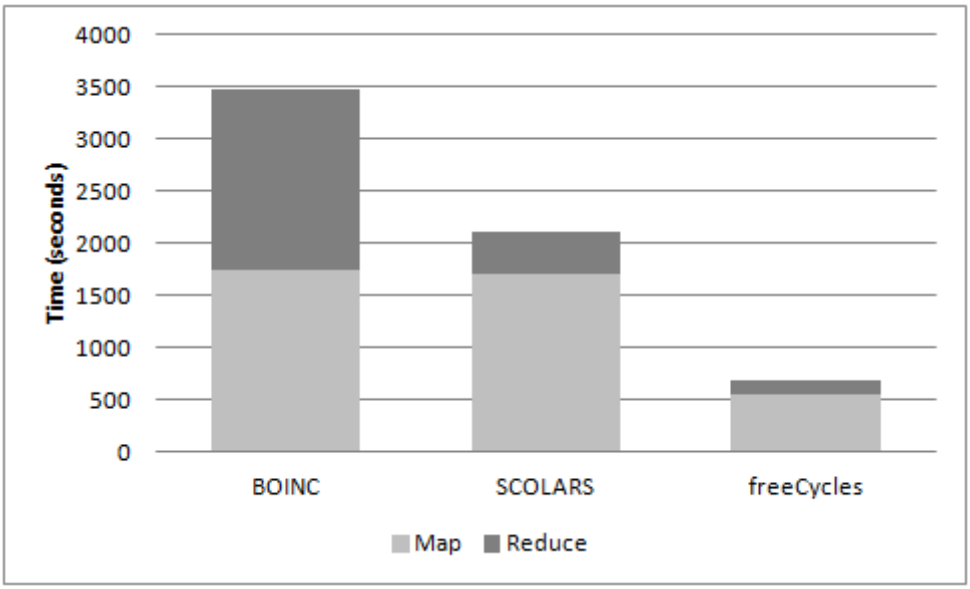


Figure 5.8: Terasort Benchmark Application

Despite the size of intermediate data (512MB), freeCycles is able to distribute intermediate data very efficiently. It is almost 5 times faster than BOINC and 3 times faster than SCOLARS.

5.2.3 Varying Input File Size

For the next experiment, we change the input file size. In the previous scenarios 512MB files were used. For this experiment, we start with a smaller input file (256MB) and increase the file size until 2048MB.

The reason for this experiment is to show how VC systems behave when large input files are used. Typical VC projects use small input files that require lots of computation. However, as already said, MapReduce applications need large input files that, most of the time, will be consumed quickly.

The input file is divided by all mappers. In our implementation, we equally divide the file using the number of lines as parameter. Therefore, all mappers will have a very similar input file size to process (and consequently, will have similar execution times).

For this experiment we use the word count application (as it has medium intermediate file sizes). We keep all nodes with their upload bandwidth limited to 10Mbps and we maintain the MapReduce configuration of 16 mappers and 4 reduces (all replicated 3 times).

Figure 5.9 shows the results for this experiment. BOINC is the system with worse execution times. SCOLARS is able to lower the time by using inter-client transfers, which effectively reduces the reduce operation time. Our system, freeCycles, has the best execution times, beating the other two systems with great advantage (4,5 times faster than BOINC and 3 times faster than SCOLARS).

It is also interesting to note that, as the input file size increases, all three solutions have different execution time increases. This has to do with the fact that, as the file size goes up, the total number of bytes uploaded goes up as well. However, as freeCycles uses volunteers to distribute data, each volunteer will have a little more data to share. On the other hand, BOINC data server will have to upload all data resulting in an almost doubled execution time when the input file size is doubled. SCOLARS has an

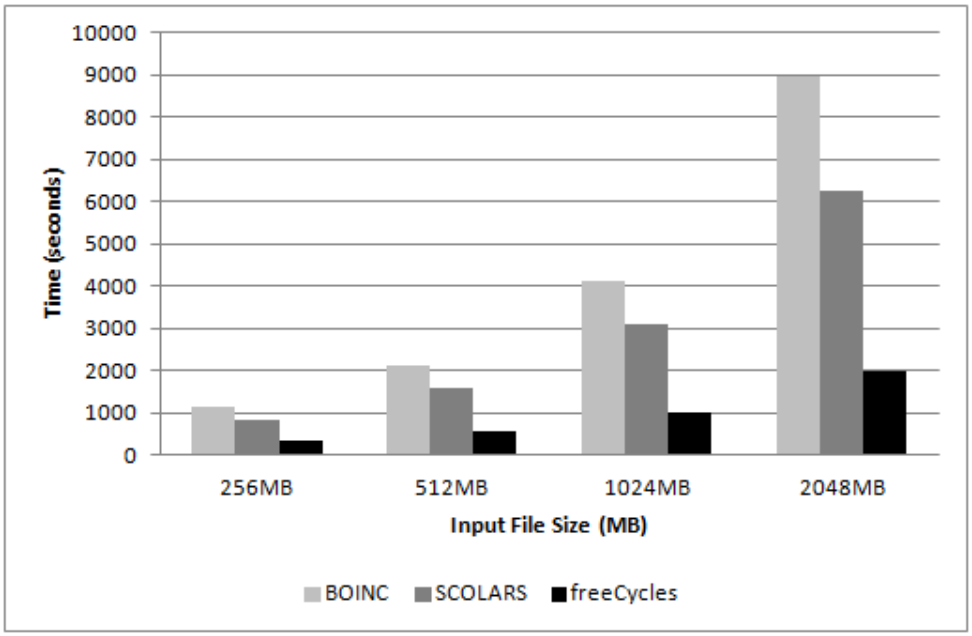


Figure 5.9: Performance varying the Input File Size

intermediate behaviour since it spares some upload bandwidth from volunteers to spread intermediate data.

5.2.4 Varying Upload Bandwidth

Another important restriction on real-life volunteer computing projects is the speed at which a data server can export computation. In other words, upload bandwidth is a very limited resource that determines the number of volunteer tasks that can be delivered at any moment.

In order to evaluate the behaviour of different systems with different upload bandwidth, we present this evaluation scenario where we repeated the execution of the word count application with a 512MB input file. We keep the 16 map and 4 reduce configuration for all runs.

We use a wide range of upload bandwidth. We start with 5Mbps and 10Mbps, reasonable values for a home Internet connections. Then we move to higher bandwidths that come closer to what is expected for a grid or cluster.

With Figure 5.10, we can see that as the upload bandwidth goes up, the difference between different solutions decreases. The reason for this is that the input file size was kept constant (at 512MB). Hence, freeCycles, for example, always have a small overhead that comes from the fact of using BitTorrent (delay to contact the central tracker, contact several volunteers and, delay in priority queues). This overhead is particularly noticeable in the last scenario (with 100Mbps) where SCOLARS performs better (since it has far less overhead).

Yet, when we move to more realistic scenarios, with upload bandwidths such as 5Mbps or even 10Mbps, freeCycles is able to perform much better than the other solutions. This is possible since freeCycles uses all the available upload bandwidth at the volunteer nodes to distribute all the data (input, intermediate and final output). On the other end of the spectrum there is BOINC, that uses a

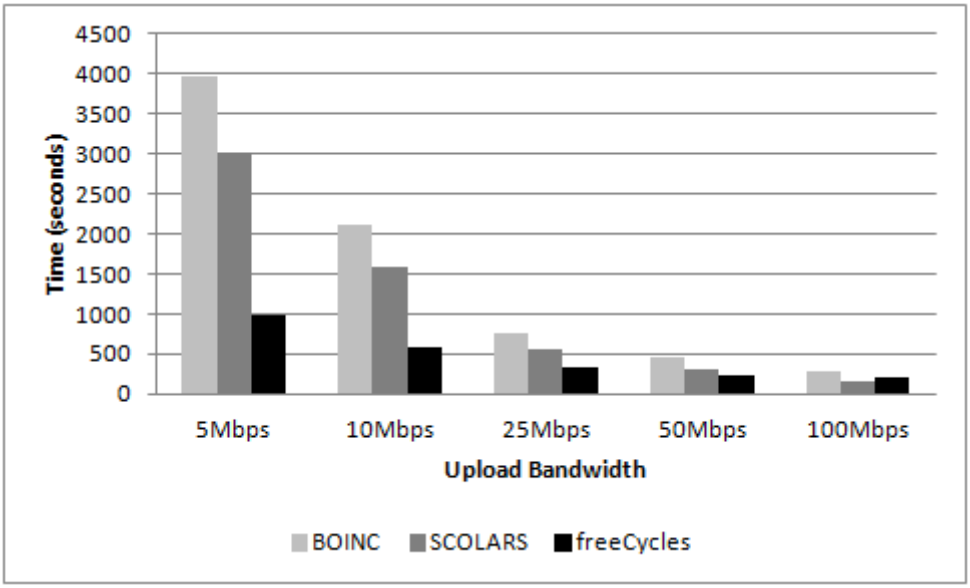


Figure 5.10: Performance varying the Upload Bandwidth

central data server to distribute all data (and thus, becomes as critical bottleneck).

With the same data from this experiment, we divided the total number of uploaded mega bytes (which is the same for the three platforms) by the time it took to distribute that amount of data. The result is an approximate average of the total upload bandwidth affectively used. We performed this division for all the bandwidths in Figure 5.10 (5, 10, 25, 50, and 100 Mbps) and the results are shown in Figure 5.11.

Figure 5.11 shows that, for some node upload bandwidths (the actual link bandwidth), freeCycles can, harvesting volunteer's upload bandwidth, obtain up to three (replication factor) times the total bandwidth that BOINC is able to use. It is also possible to see that, when the node upload bandwidth goes up, our solution is not able to increase the usable upload bandwidth. This has to do with the fact that we do not increase the input file size (which is always 512MB) as we increased the node upload bandwidth. Therefore, with higher bandwidths, the overhead of using BitTorrent becomes noticeable. Hence, it is possible to conclude that BitTorrent should be used when the ratio of the number of bytes to upload by the server upload bandwidth is high.

5.2.5 Iterative MapReduce Applications

For this next experiment, our goal is to measure the performance of the three solutions (BOINC, SCOLARS, and freeCycles) when a MapReduce application needs to run for several cycles (as the original Page Ranking algorithm).

For that purpose, we implemented a simple version of the Page Ranking algorithm which is composed by two steps: i) each page gives a share of its rank to every outgoing page link (map phase); ii) every page sums all the received shares (reduce phase). These two steps can be run iteratively until some criteria is verified (for example, if the value has converged).

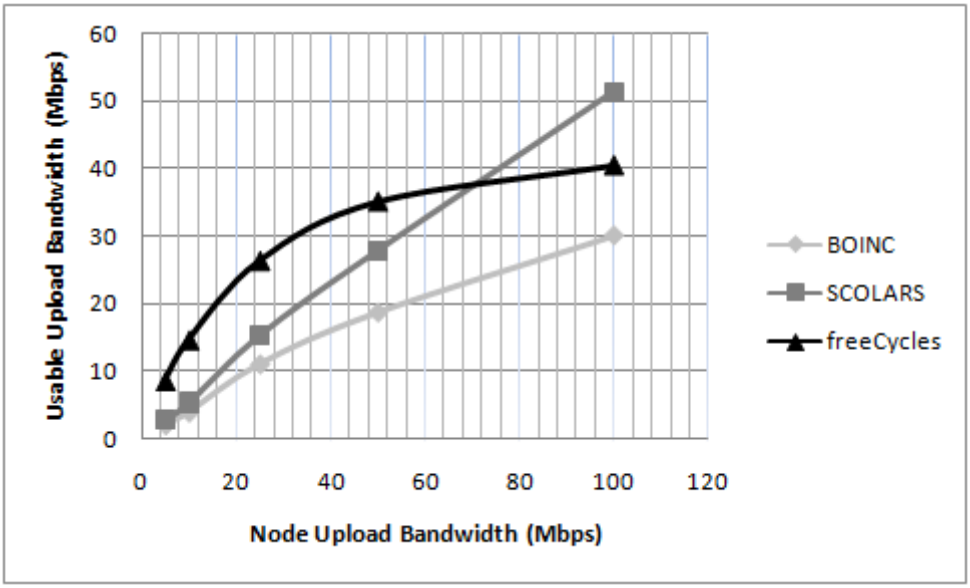


Figure 5.11: Aggregate Upload Bandwidth

We maintained the setup used in previous experiments: upload bandwidth throttled to 10Mbps, input file size of 512MB, 16 map tasks, 4 reduce tasks and 3 replicas for each task. To limit the size of the experiment (and since no extra information would be added) we use only two iterations.

Results are shown in Figure 5.12. BOINC has the highest execution times since all the data has to go back to the server after each map and reduce task. This creates a high burden on the data server which contributes to a higher overall time. It is also interesting to note that, in this experiment, intermediate data is almost 50% bigger than input and output data. This is why the reduce phase takes longer than the input phase.

SCOLARS performs much better than BOINC since intermediate data flows from mappers to reducers. However, between cycles, all data must go to the central data server and therefore, N iterations will result in N times the time is takes to run one iteration (as in BOINC).

freeCycles presents the most interesting results. The first map phase (from the first iteration) still takes a significant time to finish. Subsequent phases (first reduce, second map and second reduce) execute much faster. However, the big difference between our solution and previous ones is that output data need not to go to the central data server. Thus, input data distribution for the next iterations is much faster since multiple reducers, can feed data to the next map tasks avoiding a big bottleneck on the data server.

5.2.6 Comparison with Hadoop Cluster

Another interesting experiment to perform is to compare the execution times for the three benchmark applications on: i) volunteer pool (freeCycles) and Hadoop cluster (Hadoop is a MapReduce implementation by Apache).

To that end, we use a cluster of 60 machines (the same number of machines as in our volunteer pool) in which 10 machines also play as datanodes (datanodes are nodes that are hosting the distributed

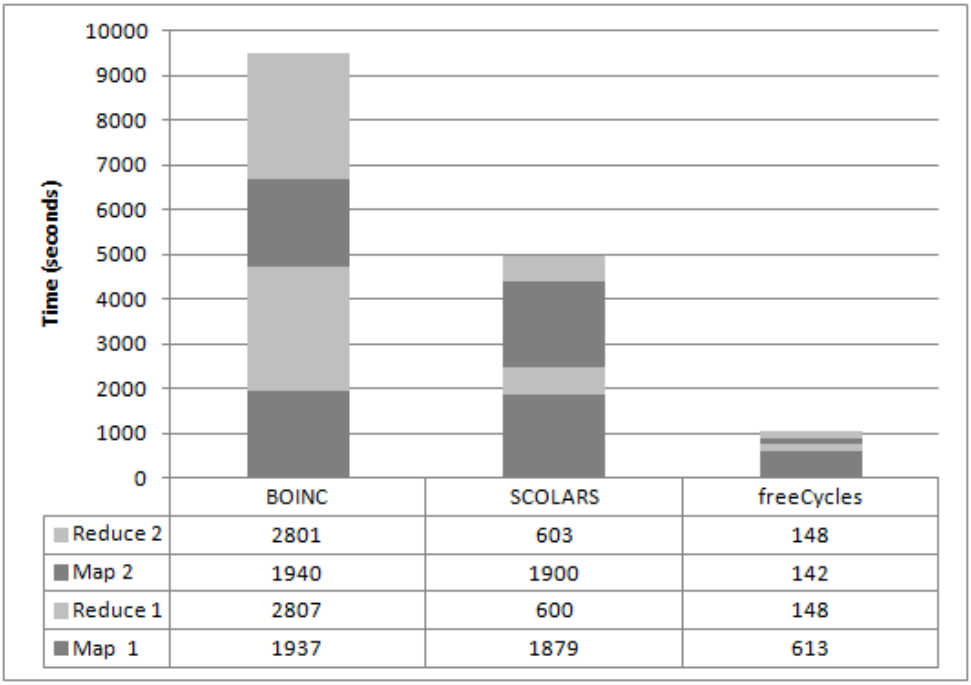


Figure 5.12: Two Page Ranking Cycles

Benchmark	Hadoop Cluster	Volunteer Pool
grep	102 sec	610 sec
Word Count	106 sec	578 sec
Terasort	92 sec	686 sec

Table 5.1: Benchmark Execution Times

file system, HDFS). All nodes have 4 cores, a total 8GB of RAM and are interconnected with Gigabit Ethernet.

For this experiment, we run the three application benchmarks: grep, word count, and terasort. We did not build these applications for Hadoop. Instead, we use the implementation provided with the platform. Regarding the freeCycles platform (Volunteer Pool), we use the same implementation as in previous experiments. All applications run with 16 mappers and 4 reduces (replicated 3 times).

Results are shown in Table 5.1. From these results, it is possible to conclude that a MapReduce application deployed on a cluster runs approximately 6 times faster than the same application deployed on a volunteer pool. This is the performance price that it takes to port an application to volunteer computing.

Nevertheless, it is important to note two factors that motivate this big performance discrepancy (between the used cluster and the volunteer pool): i) we limited the bandwidth to 10Mbps on the volunteer pool while the cluster nodes where connected via 1000Mbps; ii) input data was previously distributed and replicated among 10 datanodes (in the Hadoop cluster) while our freeCycles deployment only had one data server.

5.3 Task and Data Replication Evaluation

In this last section of the evaluation chapter, we conduct several experiments that enable us to draw some conclusions about task and data replication necessary to keep intermediate data available.

freeCycles introduces two ways to increase the availability of intermediate data: i) replicate map tasks when some replicas take too long to report; 2) replicate intermediate data when some map task is finished and validated and some of the replicas that completed the task fail (i.e., take too long to report). These two methods are applied before the reduce phase starts.

We now describe the experiments that show how freeCycles's performance is affected by the introduction of this extra replication.

5.3.1 Evaluation Simulator

For the following experiments, we used not only freeCycles, but also a simulator. We developed a simulator specially designed to replicate the behaviour of our system. Our simulator incorporates the logic of BOINC (our base VC platform), MapReduce (both on the client and server side) and BitTorrent (used to share information).

The main motivation for using a simulator instead of the real system is the possibility of obtaining more results: 1) with more nodes, 2) with more data, 3) in situations that would be harder to simulate on a cluster, 4) in less time (all the simulations performed would have taken weeks of continuous computation on the real system).

It is clear that, by using a simulator, we do not produce real results (but a lower bound instead). However, we expect to see a relation between the simulated system and the real system (as we show below).

Figure 5.13 presents a very simplified UML class diagram for our simulator's implementation. For the sake of simplicity, we only show the most important classes and their aggregation and specialization relations. The MapReduce task scheduling is done by the Server. This entity is responsible for devilering WorkUnits when a volunteer asks for work. This is also the entity that keeps information about the location of each file (it implements the logic of a BitTorrent tracker). Finally, the server is also responsible for assuring that all tasks are being replicated (and therefore, responsible for increasing the replication level when some nodes fail).

Each node (both Server and Volunteer) is responsible for keeping and updating their BitTorrent data transfers. Each node keeps references to the nodes that asked for a specific file and will, periodically, send some bytes to the destination node. Each Volunteer also has a local scheduler that knows which work needs to be done and which input files need to be downloaded before proceeding with some computation. All nodes are updated periodically.

Before delving in the results, it is important to explain how we obtain results from our simulator. Each simulation operates over a large set of entities: volunteers, a scheduler, a BitTorrent tracker, data transfers, data processing operations, just like in a real system. To produce useful results, we provide each simulation with a set of values to reproduce a specific evaluation environment. Therefore, each

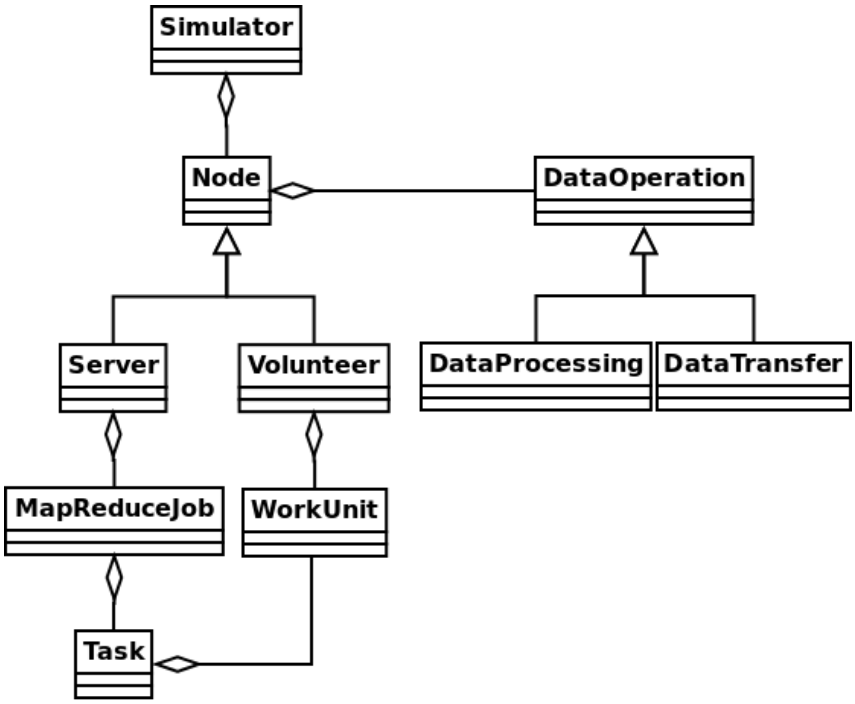


Figure 5.13: Simplified UML Class diagram for the Simulator Implementation

simulation receives as input he following values:

- number of nodes - this is the number of volunteers that are already in the system, i.e., that will periodically ask for work;
- session time - the approximate session time for each volunteer. For each volunteer, the real session time is calculated using a Normal distribution centered (mean) on the given session time and variance equal to the session time divided by 2 (future versions of this simulator could take into account that some nodes can have very long session times while others may have very short sessions);
- new volunteer rate - this is the rate by which new volunteers will join the system. In a real world scenario, volunteers come and go. This value specifies how fast new volunteers come join the system;
- upload bandwidth - assuming a real volunteer pool, over the Internet, upload bandwidth is very limited. This is the biggest bottleneck when concerning data transfers on the Internet. This value applies to all nodes, just as we did on previous experiments. During this experimental evaluation, all runs had their upload bandwidth limited to 10Mbps;
- processing power - this value expresses how many MBs a node can process per second (throughput). This is used to represent the application that is running. Different applications might take a different amount of time to produce results on the same amount of data. This value is applied for all nodes and therefore, it is not used to model nodes with different computational power. As in previous experiments, all nodes have the same processing capabilities;

- number of map tasks - the number of different map tasks, i.e., without replication;
- map task replication factor - the number of replicas for each map task;
- number of reduce tasks - the number of different reduce tasks, i.e., without replication;
- reduce task replication factor - the number of replicas for each reduce task;
- report timeout - a node is assumed to be failed if it does not report back to the server in the given amount of time. In BOINC (an therefore in freeCycles), this is a project configuration parameter. We use this value to start replication tasks or intermediate data when some volunteer does not contact the scheduler for at least this amount of time;
- input file size - the total amount of input data. The given input file size will be divided in splits so that each mapper processes an approximately the same amount of data;
- intermediate file size - the total amount of intermediate data. Since we are not using the concept of application, the simulator needs to know how much data each map task will produce;
- output file size - the total amount of output data. This value is needed for the same reason as the previous one.

5.3.2 Comparison with freeCycles

Before describing the obtained results, it is important to analyse how close our simulator is to the real system, freeCycles. To that end, we perform some experiments using the same application benchmarks as before: word count, terasort and grep. Since these benchmarks are representative of typical MapReduce applications (see Section 5.2.2), a good relation between the real and the simulated results would gives us an idea about the confidence degree for our simulator. In this experiment we use a MapReduce workflow with 16 map tasks, 4 reduce tasks with a replication factor of 3 (both map and reduce tasks) and a 512MB input file.

For the three application benchmarks (see Figure 5.14), our simulator finished faster than the real system, freeCycles. This, however, is expected since the simulator simplifies reality in several ways: it does not consider the time packages take to reach the destination (latency), it does not take into account the queueing delays (that are present in every BitTorrent node), it assumes that a seeder always sends different parts of a file, in a particular moment, to leechers (although the BitTorrent protocol converges to this situation). For these reasons, it is expected that our simulator performs better than freeCycles.

One interesting fact to notice is that our simulator obtains better results for grep than for word count. The same does not hold for freeCycles. As reported in Section 5.2.2, although in theory grep should be faster (since is has less intermediate data to transfer), both grep and word count have similar results. This is explained by the fact that the size of the intermediate data is so small (less than 2MB) that the time to complete the reduce phase is majored by the BitTorrent overhead. Since our simulator does not take into account such overheads, the simulated results for the grep application are better than for the word count application.

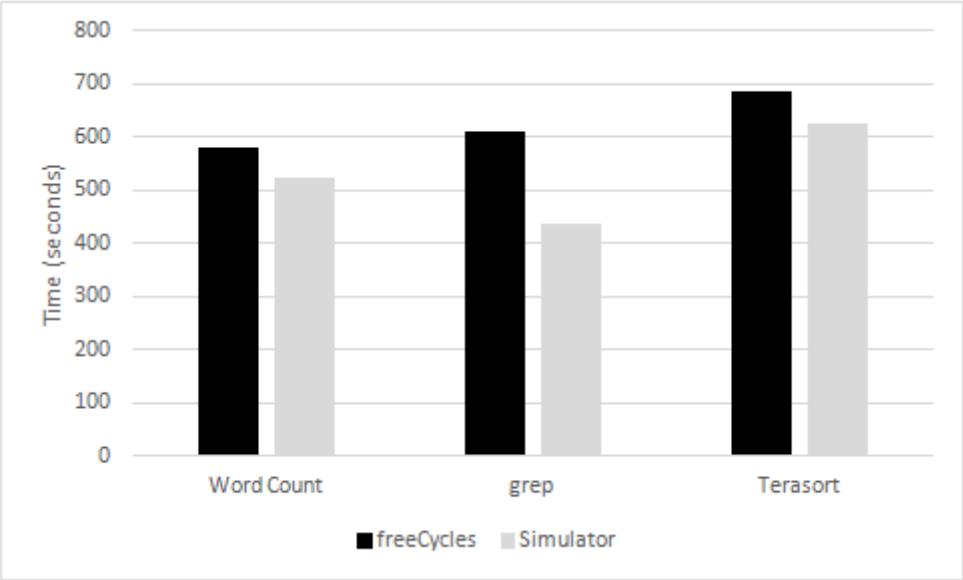


Figure 5.14: Performance comparison between freeCycles and the Simulator

To close this experiment, we conclude that, although the simulator provides a lower bound for our experiments (i.e., the simulator obtains better/lower results than the real system), conclusions will most probably hold on the real system.

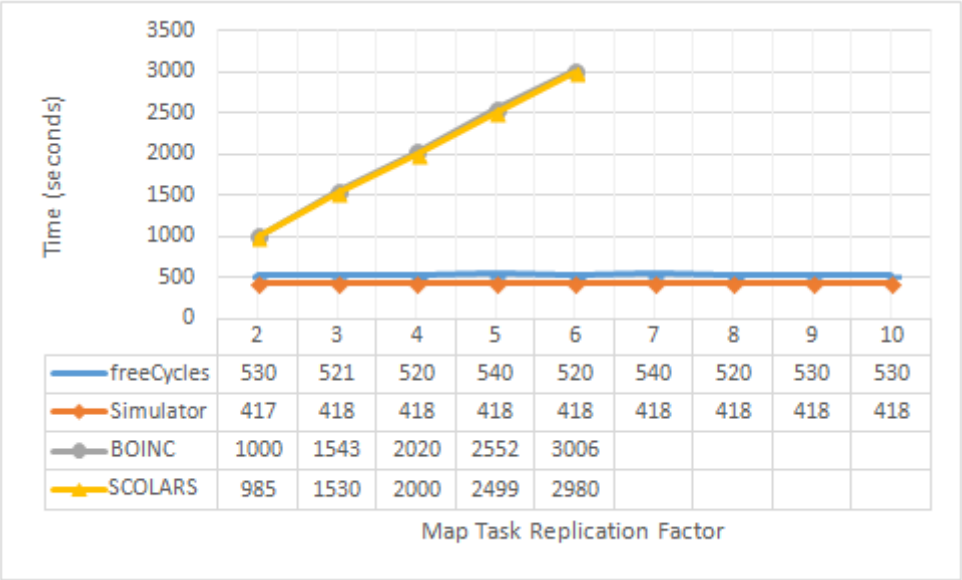
5.3.3 Varying the Task Replication Factor

In this second experiment, we show how BOINC, SCOLARS, freeCycles, and the simulator behave when increasing the map task replication factor. In previous experiments, this value was, by default, 3 (so that a quorum of answers was possible). To decrease the number of nodes needed for this experiment, we reduce the number of map and reduce tasks to 4 and 1 respectively. We measure the time needed to finish both the map and reduce phases, of a word count benchmark, on freeCycles, on our simulator, on BOINC, and on SCOLARS. The size of the input file is 512MB.

Figure 5.15 shows the amount of time needed, by each system, to complete the map phase as we increase the number of map task replicas. Both BOINC and SCOLARS show very similar results. In fact, with respect to the map phase, BOINC and SCOLARS have the same behaviour: use the central server to upload all input data to all mappers. BOINC has an extra (although smaller) step that every mapper has to follow: send the map output data back to the server. Since the map output data is much smaller than the map input data and given that we are not limiting the download bandwidth, this step is relatively fast (it is noticeable by a small overhead compared to SCOLARS).

freeCycles and our simulator also show very similar results. Both systems keep all map phase finish times almost constant as the number of map task replicas increases. This confirms the fact that BitTorrent scales with the number of nodes. In this scenario, as the number of replicas increases, the number of nodes downloading a particular file increases as well. However, since all nodes share all their file chunks, it is possible to keep the input data upload time almost constant.

The second plot (Figure 5.16) shows that amount of time that it took finish the reduce tasks when the



	2	3	4	5	6	7	8	9	10
freeCycles	530	521	520	540	520	540	520	530	530
Simulator	417	418	418	418	418	418	418	418	418
BOINC	1000	1543	2020	2552	3006				
SCOLARS	985	1530	2000	2499	2980				

Figure 5.15: Map Task Finish Time

map replication factor is increased. BOINC has the worst performance. As the reduce task replication factor is kept always constant (three), the reduce phase produces the same results for every map task replication factor. SCOLARS exhibits a similar behaviour, but with lower completion times. The fact that each reducer contacts the necessary mappers instead of receiving all the data through the central server brings some performance gains to SCOLARS, comparing to BOINC. Once again, freeCycles (and therefore the simulator also) presents the best results.

Figure 5.17 presents the overall MapReduce finish time (in fact, each point can be obtained by the sum of the correspondent map and reduce phases from the two previous plots). One important conclusion is that freeCycles' performance is not hindered by the increase of map replicas (as opposed to SCOLARS and BOINC). Since freeCycles uses the volunteer's upload bandwidth to distribute data, as more replicas come, more aggregated upload bandwidth there is to spread data. As a matter of fact, as the number of map replicas is increased, more volunteers will be available to upload intermediate data leading to a faster reduce phase (as we demonstrate next).

Increased Input File Size

Previous results (Figure 5.16) do not show clearly the increased performance when adding additional map replicas. This is so because freeCycles accumulates some overhead from using BitTorrent to distribute data. Therefore, for small files, the transfer time will be majored by this overhead, mainly waiting in priority queues, latency when communicating with other nodes or even with the tracker. In this experiment, we increased the input data to bring up the real transfer time.

We simulated a MapReduce workflow using our simulator. The MapReduce job used 16 map tasks and 4 reduce tasks, each one with a replication factor of 3. We used a bigger (than in previous experiments) input file, with 16GB.

Results are shown in Figures 5.18 and 5.19. We used two different benchmark applications since

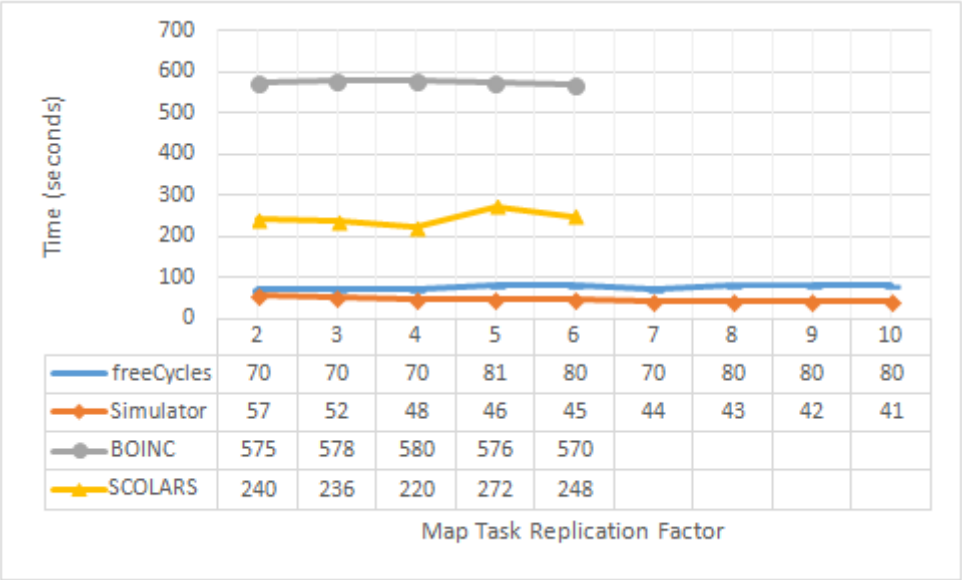


Figure 5.16: Reduce Task Finish Time

each one will produce a different amount of intermediate data and therefore, different performance gains. As the terasort benchmark produces much more intermediate data than the word count, it is possible to see that the terasort's reduce phase decreases much faster than word count.

It is then possible to conclude that freeCycles's performance is not harmed by extra replication of map tasks. However, this does not hold for BOINC or SCOLARS. Using this fact, the problem of deciding when to replicate data or tasks is almost solved. Since there is no performance degradation, data or tasks can be replicated as soon we some volunteer is suspected to be failing. This procedure, although applicable to freeCycles, might not be good for BOINC or SCOLARS, since it could bring a big performance degradation if data or tasks are replicated too soon (if the node is not really down).

5.3.4 Real World Simulation

Until now, all described experiments used a set of machines that were always ready to receive work. Since this is not much representative of a real world volunteer pool, we now use our simulator to create a virtual scenario were we try to approximate a real world environment.

To simulate a real world environment, we manipulate two arguments used in our simulator: the volunteer session timeout (it is important to note that each volunteer will have a different lifetime, as we use a normal distribution (which is one of the most used [8, 38] distributions to model node churn) to add or subtract some variation on this parameter), and the new volunteer rate (how many volunteers join the network in a fixed amount of time). Using these two arguments we can simulate the node churn rate. The higher the session timeout, the lower the churn rate is. On the other hand, the higher the new volunteer rate, the higher the churn rate is.

Using published results [8, 38] on peer-to-peer node churn, in particular, node churn in BitTorrent networks, we decided to use a volunteer session time of 2 hours and a new volunteer rate of 1 volunteer per minute. As for the other simulation values, we performed a MapReduce workflow with 16 map tasks

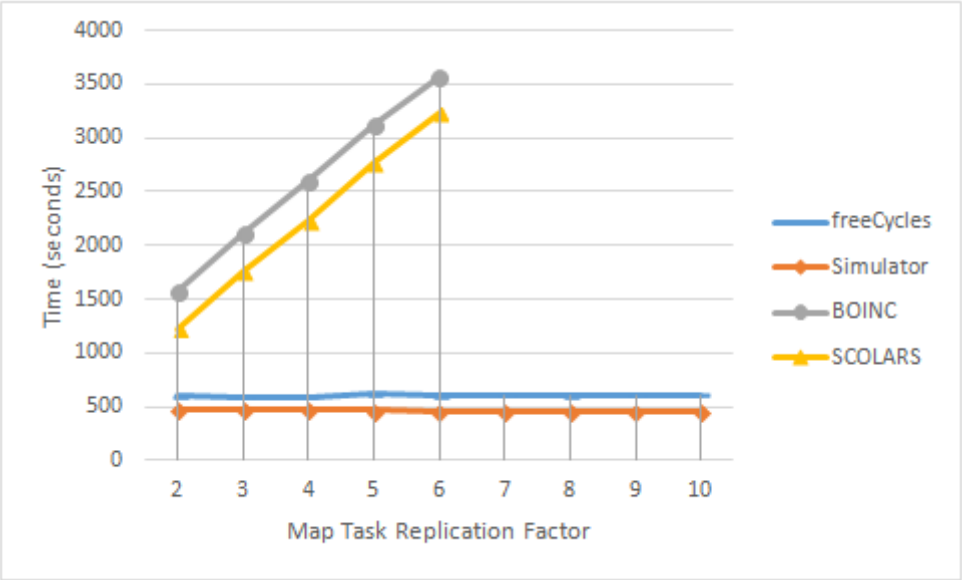


Figure 5.17: MapReduce Total Finish Time

Session Time / New Volunteer Rate	1 vol/min	10 vol/min	100 vol/min
1 hour	INF	2710 sec	2282 sec
2 hour	3922 sec	2527 sec	2114 sec
3 hour	3880 sec	2398 sec	2104 sec
4 hour	3754 sec	2354 sec	2104 sec
5 hour	3754 sec	2354 sec	2104 sec

Table 5.2: Real World Simulations

and 4 reduce tasks, all with a replication factor of 3. The input data has 2GB of size. The upload bandwidth is always kept as 10Mbps (1,25MBps).

We can conclude from Figure 5.20, that the results for the benchmark application are much worse in a real environment, where there are nodes constantly failing and new nodes coming.

One important conclusion to take from this experiment it that, each MapReduce job should be carefully designed to keep the overall needed time below the average node session time. This is a very simple way to avoid extra replication costs that would increase the turnaround time. In a previous experiment we showed that having extra replication of map tasks and/or intermediate data can decrease the reduce phase time. However, for example, failed reduce tasks will always delay the finish time.

We further demonstrate the effects of higher and lower churn rate on Table 5.2. Results were produced with the job parameters used in the previous experiment (the one in Figure 5.20). The difference is that we now manipulate the volunteer session time and new volunteer rate to change the network churn rate.

Results from Table 5.2 show that the time needed to complete a MapReduce job is reduced by: i) increasing the volunteer session time and ii) increasing the new volunteer rate (number of new volunteers per minute). The interesting result is the one obtained for one hour of session time and one new volunteer per minute. Our simulator shows that, for this specific job characteristics: i) there are not enough volunteers and ii) volunteers' session time is too low to complete the job. For these two factors, most of

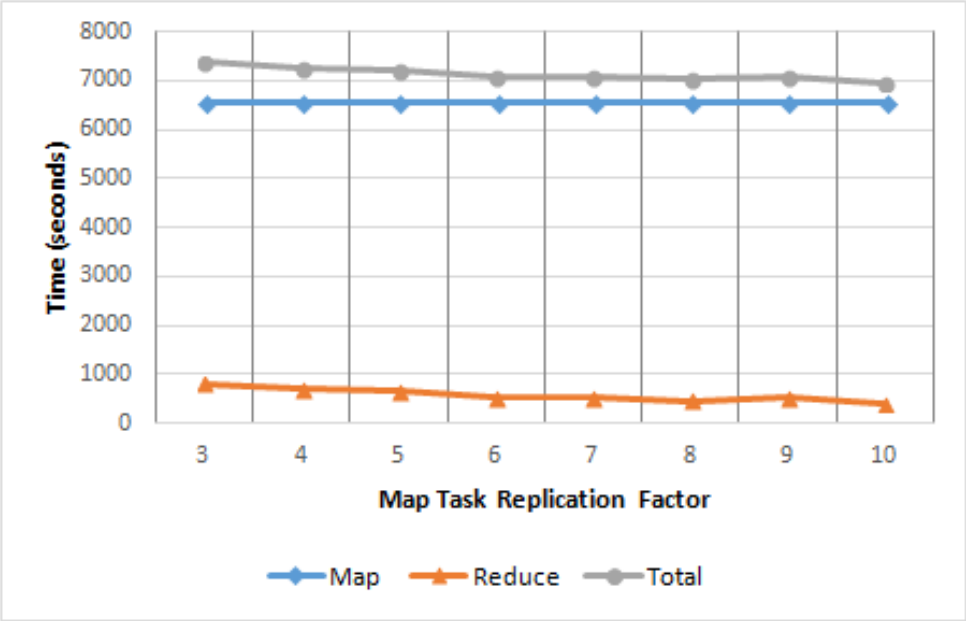


Figure 5.18: Word Count Benchmark

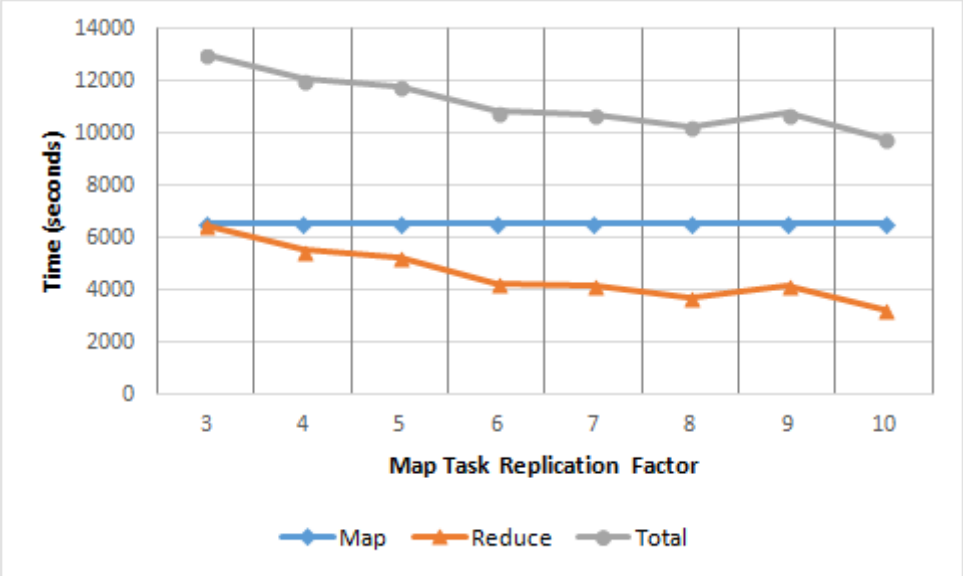


Figure 5.19: Terasort Benchmark

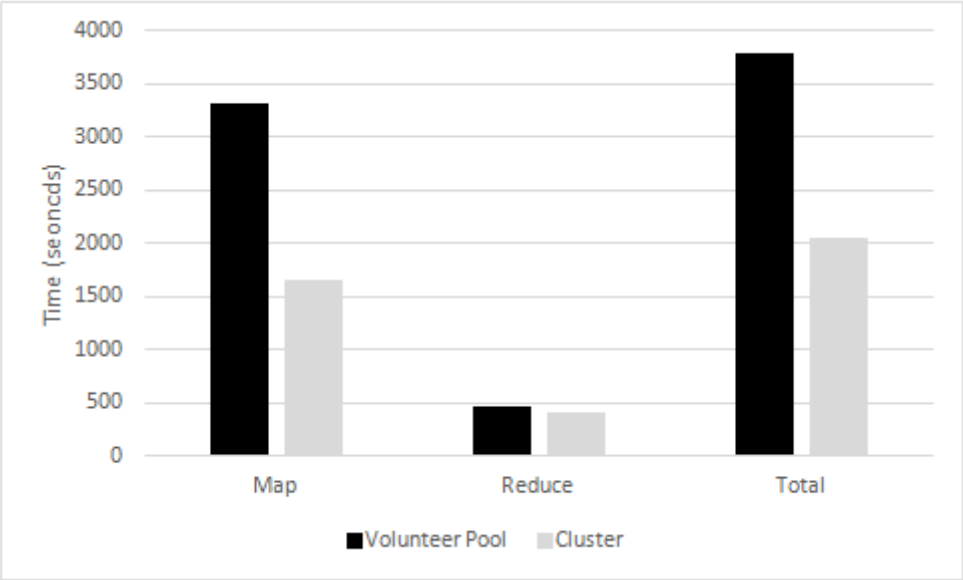


Figure 5.20: Environment Comparison using Word Count Benchmark

our runs never end because the few nodes that exist fail too often.

5.3.5 Varying the Replication Timeout

We now close our evaluation with an experiment that shows how much time one should wait until start replicating map tasks or intermediate data (if already validated). To this end, we repeated the experiment of the previous section (using the environment configuration for a volunteer pool) and used an increasing number of seconds for the time we wait until we declare a node failing. As soon as a node is detected to be failing, some kind of replication process is started: intermediate data replication (if the failed node is a mapper and the map output is already validated) or task replication.

Analysing the results on Figure 5.21, we are able to confirm our expectations: the sooner we start the replication process, the faster the MapReduce job finishes. This is explained by the fact that our data distribution protocol (that uses BitTorrent) is not harmed by the addition of replicas (see a detailed explanation on Section 5.3.3).

5.4 Summary

This chapter provided an extensive set of evaluation experiments and results. We started by testing the selected protocol, BitTorrent, against the ones that are usually used (FTP and HTTP). Then we compared our platform against BOINC and SCOLARS using several benchmark applications in different settings. The final step in our evaluation was to introduce failures and observe how the systems behaved. In all the three stages of the evaluation, freeCycles, outperformed the other systems both in performance and scalability.

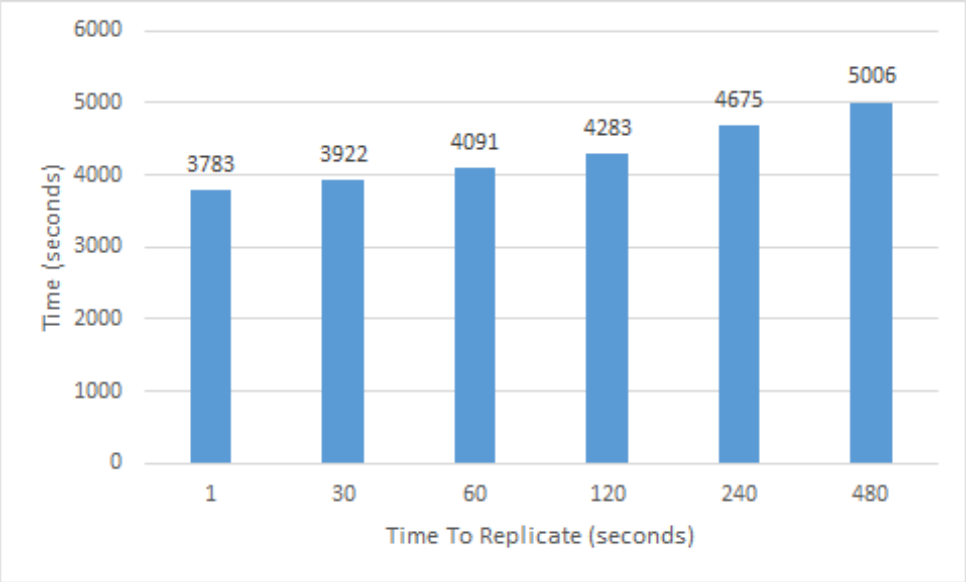


Figure 5.21: Performance Evaluation Varying the Replication Timeout

Chapter 6

Conclusions

In this final chapter, we conclude by presenting several achievements and contributions to the scientific community, Volunteer Computing area in particular. We further extend this chapter by suggesting some ideas for future work, some related or motivated by this work, others concerning VC in general.

6.1 Achievements

Back in the beginning of this document (see Chapter 1), we propose and present freeCycles as a VC platform that supports MapReduce and uses BitTorrent to provide efficient and scalable data transfers between all MapReduce participants (server and all volunteers). To deal with the volatility of intermediate data (that might lead to large overheads in the MapReduce execution time) freeCycles proposes aggressive replication of computation (map tasks) or data (intermediate data).

Through our extensive performance evaluation (see previous chapter) we demonstrate that our goals were successfully achieved. freeCycles achieves better performance and scalability compared than other equivalent solutions. With freeCycles, MapReduce applications can now benefit from larger volunteer pools since scalability is largely increased (in the number of nodes, in the size of input data and in the amount of usable upload bandwidth).

In short, freeCycles' contributions are: i) a BOINC compatible platform that introduces the BitTorrent protocol in the context of MapReduce jobs; ii) conclusions on the performance and scalability impact of data and task replication; iii) extensive performance evaluation, comparing freeCycles with other similar platforms and proving its benefits (with regards to previous solutions); iv) a research paper entitled *freeCycles: Efficient Data Distribution for Volunteer Computing* published in CloudDP'14 Proceeding of the Fourth International Workshop on Cloud Data and Platforms (within the EuroSys'14 Conference, a rank A conference according to CORE 2013).

We hope that our contributions help to further develop current VC platforms, namely BOINC, and future ones, still to be created.

6.2 Future Work

During the time that we worked on this topic, Volunteer Computing, we were able to identify several possible future contributions to this area. In other words, there are several aspects of VC that could be improved to better facilitate the harnessing of volunteer pools, for example.

Some of these contributions are directly related to this work, freeCyles, but others concern VC in general. We conclude this document with the following ideas for future work:

- Study ways to use some kind of (system or process) Virtual Machines (VMs) in the context of VC. Several VC platforms (like BOINC itself) use native code to run. Using native code brings better performance guarantees but forces developers to write the same application for multiple platforms. The typical scenario is to have scientists as Linux (or some other similar platform) users, and the vast majority of volunteers as Windows users. In this scenario, scientists are forced to learn and program for Windows.

  By using some kind of VMs, scientists would only need to create one version of the distributed application and use a VC platform to ship the computation. There are, however, some problems: 1) instead of shipping data plus an executable, a VC platform using VMs would have to ship a VM image (although this could be highly optimized by using peer-to-peer protocols and differential VM images); 2) code execution would be slower if the hosting Operating System (OS) and hosting CPU are not ready for virtualization; 3) volunteers would be forced to have some virtualization software (although this could be included in the VC platform software; 4) the access to GPUs or other dedicated computing cards could be limited (this depends on the support of the virtualization technology).

  Using Java or other language with an intermediate representation might not be the solution since: 1) scientists are used to program in C/C++; 2) C/C++ is more appropriate and efficient for the type of computation (CPU intensive) usually performed in VC.

  Despite all the problems, having a facilitated development of distributed applications would greatly alleviate an important problem of VC;

- Study the energy efficiency of using a cluster versus using VC. It is clear that VC needs replication to overcome some problems (slow volunteers and malicious/faulty volunteers). However, using two or three volunteers to export some computation is worth shutting a cluster machine down?

  From the energetic point of view, the CPU is, usually, the most expensive component. Yet, the CPU energy consumption depends on its utilization, i.e., harnessing the CPU power from a volunteer will produce a slight increase in the energy consumption (assuming that the computer is being utilized). Another interesting fact is that clusters spend a lot of energy in cooling systems (recent data from Google and Facebook shows that 20-30% [1] of the energy is used to feed cooling systems). In a VC environment, there is no need for such cooling systems.

[1] several links discussing this issue can be found at: http://www.google.com/about/datacenters/efficiency/internal/#water-and-cooling a nd https://www.facebook.com/notes/facebook-engineering/new-cooling-strategies-for-greater-data-center-energy-efficiency/448717123919.

The question is if it is worth exporting some (non critical) computation from cluster computing nodes (like the ones in big data centers) to volunteer computers (even if receiving small incentives).

- Study and develop a system that decides in which workload, a new volunteer asking for work, should be placed. Previous studies [2] show that mixing different nodes (i.e., nodes with different computing capabilities) produce bad results, even worse than using only slower nodes.

  As volunteer pools are very heterogeneous, it would be very important to study even further this issue and understand the relation between the difference of computing power and the resultant execution time overhead.

  A VC platform would benefit from a system like this by being able to place nodes in different jobs, according to the job's properties (priority, computation to data ratio, etc) and some node benchmark results (for example: CPU power, available upload bandwidth, available storage).

Bibliography

[1] N. Ahituv, Y. Lapid, and S. Neumann. Processing encrypted data. *Commun. ACM*, 30(9):777–780, Sept. 1987. ISSN 0001-0782. doi: 10.1145/30401.30404. URL http://doi.acm.org/10.1145/30401.30404.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. *SIGARCH Comput. Archit. News*, 40(1):61–74, Mar. 2012. ISSN 0163-5964. doi: 10.1145/2189750.2150984. URL http://doi.acm.org/10.1145/2189750.2150984.

[3] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Superweb: towards a global web-based parallel computing infrastructure. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 100–106, 1997. doi: 10.1109/IPPS.1997.580858.

[4] D. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004. doi: 10.1109/GRID.2004.14.

[5] D. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80, 2006. doi: 10.1109/CCGRID.2006.101.

[6] D. Austin. How google finds your needle in the web's haystack. *American Mathematical Society Feature Column*, 10:12, 2006.

[7] A. Baratloo, M. Karaul, Z. Kedem, and P. Wijckoff. Charlotte: Metacomputing on the web. *Future Generation Computer Systems*, 15(5–6):559 – 570, 1999. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/S0167-739X(99)00009-6. URL http://www.sciencedirect.com/science/article/pii/S0167739X99000096.

[8] A. Binzenhöfer and K. Leibnitz. Estimating churn in structured p2p networks. In *Managing Traffic Performance in Converged Networks*, pages 630–641. Springer, 2007.

[9] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[10] A. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: self-organizing computation

on a peer-to-peer network. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(3):373–384, 2005. ISSN 1083-4427. doi: 10.1109/TSMCA.2005.846396.

[11] L. Cherkasova and J. Lee. Fastreplica: Efficient large file distribution within content delivery networks. In *USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, USA, 2003.

[12] F. Costa, L. Silva, G. Fedak, and I. Kelley. Optimizing the data distribution layer of boinc with bittorrent. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008. doi: 10.1109/IPDPS.2008.4536446.

[13] F. Costa, L. Veiga, and P. Ferreira. Vmr: volunteer mapreduce over the large scale internet. In *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1608-8. doi: 10.1145/2405136.2405137. URL http://doi.acm.org/10.1145/2405136.2405137.

[14] F. Costa, L. Veiga, and P. Ferreira. Internet-scale support for map-reduce processing. *Journal of Internet Services and Applications*, 4(1):1–17, 2013. ISSN 1867-4828. doi: 10.1186/1869-0238-4-18. URL http://dx.doi.org/10.1186/1869-0238-4-18.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[16] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: a generic global computing system. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 582–587, 2001. doi: 10.1109/CCGRID.2001.923246.

[17] G. Fedak, H. He, and F. Cappello. Bitdew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. *Journal of Network and Computer Applications*, 32(5):961 – 975, 2009. ISSN 1084-8045. doi: http://dx.doi.org/10.1016/j.jnca.2009.04.002. URL http://www.sciencedirect.com/science/article/pii/S1084804509000587. ¡ce:title¿Next Generation Content Networks¡/ce:title¿.

[18] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[19] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[20] O. Heckmann and A. Bock. The edonkey 2000 protocol. *Multimedia Communications Lab, Darmstadt University of Technology, Tech. Rep. KOM-TR-08-2002*, 2002.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[22] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 181–192. ACM, 2010.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[24] J. Liang, R. Kumar, and K. W. Ross. The fasttrack overlay: A measurement study. *Computer Networks*, 50(6):842–858, 2006.

[25] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851489. URL http://doi.acm.org/10.1145/1851476.1851489.

[26] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In *Peer-to-Peer Systems III*, pages 227–236. Springer, 2005.

[27] F. Marozzo, D. Talia, and P. Trunfio. Adapting mapreduce for dynamic environments using a peer-to-peer model. In *Proceedings of the 1st Workshop on Cloud Computing and its Applications*, 2008.

[28] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Euro-Par 2000 Parallel Processing*, pages 1231–1238. Springer, 2000.

[29] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet-the popcorn project. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 592–601, 1998. doi: 10.1109/ICDCS.1998.679836.

[30] H. Pedroso, L. M. Silva, and J. G. Silva. Web-based metacomputing with jet. *Concurrency: Practice and Experience*, 9(11):1169–1173, 1997. ISSN 1096-9128. doi: 10.1002/(SICI)1096-9128(199711)9:11⟨1169::AID-CPE350⟩3.0.CO;2-6. URL http://dx.doi.org/10.1002/(SICI)1096-9128(199711)9:11<1169::AID-CPE350>3.0.CO;2-6.

[31] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[32] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[33] L. F. Sarmenta. *Volunteer computing*. PhD thesis, Massachusetts Institute of Technology, 2001.

[34] L. F. Sarmenta and S. Hirano. Bayanihan: building and studying web-based volunteer computing systems using java. *Future Generation Computer Systems*, 15(5–6):675 – 686, 1999. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/S0167-739X(99)00018-7. URL http://www.sciencedirect.com/science/article/pii/S0167739X99000187.

[35] X. Shen, H. Yu, J. Buford, and M. Akon. *Handbook of peer-to-peer networking*, volume 1. Springer Heidelberg, 2010.

[36] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2p-based middleware enabling transfer and aggregation of computational resources. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 1, pages 259–266 Vol. 1, 2005. doi: 10.1109/CCGRID.2005.1558563.

[37] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 11:1–11:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654071. URL `http://doi.acm.org/10.1145/1654059.1654071`.

[38] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[39] B. Tang, M. Moca, S. Chevalier, H. He, and G. Fedak. Towards mapreduce for desktop grid computing. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on*, pages 193–200, 2010. doi: 10.1109/3PGCIC.2010.33.

[40] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005. ISSN 1532-0634. doi: 10.1002/cpe.938. URL `http://dx.doi.org/10.1002/cpe.938`.

[41] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: A mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 36(SI):329–343, 2002.

[42] B. Wei, G. Fedak, and F. Cappello. Scheduling independent tasks sharing large data distributed with bittorrent. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 219–226. IEEE Computer Society, 2005.

[43] B. Wei, G. Fedak, and F. Cappello. Towards efficient data distribution on computational desktop grids with bittorrent. *Future Generation Computer Systems*, 23(8):983–989, 2007.

[44] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[45] L. Zhong, D. Wen, Z. W. Ming, and Z. Peng. Paradropper: a general-purpose global computing environment built on peer-to-peer overlay network. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 954–957, 2003. doi: 10.1109/ICDCSW.2003.1203674.