J Grid Computing (2017) 15:501–526
https://doi.org/10.1007/s10723-017-9414-2

freeCycles - Efficient Multi-Cloud Computing Platform

Rodrigo Bruno · Fernando Costa ·
Paulo Ferreira

Received: 1 October 2016 / Accepted: 9 October 2017 / Published online: 30 October 2017

Abstract The growing adoption of the MapReduce programming model increases the appeal of using Internet-wide computing platforms to run MapReduce applications on the Internet. However, current data distribution techniques, used in such platforms to distribute the high volumes of information which are needed to run MapReduce jobs, are naive, and therefore fail to offer an efficient approach for running MapReduce over the Internet. Thus, we propose a computing platform called freeCycles that runs MapReduce jobs over the Internet and provides two new main contributions: i) it improves data distribution, and ii) it increases intermediate data availability by replicating tasks or data through nodes in order to avoid losing intermediate data and consequently avoiding significant delays on the overall MapReduce execution time. We present the design and implementation of freeCycles, in which we use the BitTorrent protocol to distribute all data, along with an extensive set of performance results, which confirm the usefulness of the above mentioned contributions. Our system's improved data distribution and availability makes it an ideal platform for large scale MapReduce jobs.

R. Bruno (✉) · F. Costa · P. Ferreira
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
e-mail: rodrigo.bruno@tecnico.ulisboa.pt

F. Costa
e-mail: fernando.costa@tecnico.ulisboa.pt

P. Ferreira
e-mail: paulo.ferreira@inesc-id.pt

Keywords Cloud computing · BitTorrent · BOINC · MapReduce · Volunteer computing

1 Introduction

Current trends show a growing demand for computational power; scientists and companies all over the world strive to harvest more computational resources in order to solve increasingly complex problems in less time, while spending the least amount of money possible [8, 23, 24]. With these two objectives in mind, we believe that aggregating computing resources all over the Internet (much like Volunteer Computing [4], VC, projects do) is a viable solution to access tremendous untapped computing power, namely CPU cycles, network bandwidth, and storage, at virtually no cost.

As Internet computing matures, more computing devices (e.g., PCs, gaming consoles, tablets, mobile phones, etc.) join the network. By gathering all these resources in a global computation pool, it is possible to obtain huge amounts of resources that would be impossible, or impractical, for most grids, supercomputers, and clusters. For example, recent data from

BOINC projects [3, 5] shows that, currently, there are 50 supported projects sustained by an average computational power of over seven PetaFLOPS.[1]

Large scale Internet-wide computing platforms enable large projects that could not be executed on grids, supercomputers, or clusters due to its size or cost, to be deployed using Internet-wide resources. In addition, recent developments of popular programming models, namely MapReduce[2] [19], raise the interest of using Internet-wide computing platforms to run MapReduce applications on large scale networks, such as the Internet. Despite increasing the attractiveness of such platforms, this also requires the research community to rethink and evolve the architecture and protocols (in particular, data distribution) used by existing systems.

Hence, we present freeCycles, an Internet-wide MapReduce-enabled computing platform which aims at aggregating as many computing resources as possible in order to run MapReduce jobs in a scalable, efficient, and fault-tolerant way, over the Internet. The output of this project is a middleware platform that enables upper software abstraction layers (programs developed by scientists, for example) to use available resources all over the Internet as they would use a supercomputer or a cluster.

freeCycles must fulfill several key requirements: i) improve, compared to other solutions, the overall throughput of MapReduce applications (results presented in Sections 6.2, 6.3, 6.5, and 6.8), ii) collect nodes' resources such as CPU cycles, network bandwidth, and storage in an efficient way (results presented in Section 6.4), iii) tolerate node failures (Section 4 provides further details on the fault model we use; Sections 6.7, 6.9, and 6.10 show such results), and iv) support MapReduce, a particularly interesting programming model, given its relevance for a large number of applications (Section 2 presents a description of our solution and how it supports MapReduce applications).

MapReduce applications typically have two key features: i) they require large amounts of input data to run, and ii) they may run for several iterations, where data can be processed and transformed several times (this is specially important for iterative algorithms such as the page rank [30]). Therefore, in order to take full advantage of the MapReduce model, freeCycles must be able to efficiently distribute large amounts of data while allowing applications to perform several MapReduce iterations without compromising its scalability (in a clustered environment this problem is solved using tools like Spark [52]). As shown in Section 6.5, solutions such as BOINC and SCOLARS do not scale with the number of iterations, as the application execution time increases linearly when multiple MapReduce iterations are required. On the contrary, freeCycles only adds a small cost when multiple MapReduce iterations are required.

To better understand the challenges inherent to building a solution like freeCycles, it is important to note the differences between the main three computing environments: clusters, grids, and volatile computing pools (available from the Internet).

Clusters are composed of dedicated computers, with fast inter-node connection speeds. Nodes have very low failure rates, are very similar in hardware and software and all their resources are focused on cluster jobs.

Grids may be created by aggregating desktop computers from universities, research labs or even companies. Computers have a 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.

Finally, we have volatile computing pools, i.e., Internet-connected nodes. This environment is made of arbitrary computers, owned by individuals, institutions, or cloud providers around the world. Nodes have variable Internet connection bandwidth, different computation costs (e.g., CPU per hour), node churn is very high (compared to clusters or grids), nodes are very asymmetrical in terms of hardware and software and their computing resources may be preempted by user tasks (if resources come from volunteer nodes). As opposed to grids and clusters, computers cannot be trusted since they may be managed by malicious users.

[1]Statistics from boincstats.com

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

It is important to note that in a real deployment, nodes can be managed by volunteers (as in Volunteer Computing projects), provided by Cloud Services (such as Amazon), or even both. In the scope of freeCycles, we do not distinguish between these scenarios.

By considering all the available solutions, note that systems based on clusters and/or grids do not fit our needs. Such solutions are designed for controlled environments in which node churn is expected to be low, nodes are typically well connected with each other, can be fully trusted, and are very similar in terms of software and hardware. Therefore, such solutions (HTCondor [48], Hadoop [50], XtremWeb [21]) and other similar computing platforms are of no use to reach our goals.

When considering platforms that harness resources all over the Internet (GridBot [43], Bayanihan [42], and others [2, 6, 14, 34]), we observe that most existing solutions are built and optimized to run Bag-of-Tasks applications. Therefore, such solutions do not support the execution of MapReduce jobs, which is one of our main requirements.

With respect to the few solutions that support MapReduce [18, 33, 35, 46], we detect some drawbacks (more details in Section 7): data distribution could be improved, intermediate data availability is overlooked, and there is a lack of support for iterative MapReduce applications.

To solve the aforementioned drawbacks, we present freeCycles[3], a BOINC-compatible computing platform that enables the deployment of MapReduce applications over the Internet. Besides supporting MapReduce jobs, freeCycles goes one step further by allowing nodes (mappers or reducers) to help distribute both the input, intermediate output, and final output data. freeCycles uses BitTorrent[4] to replace point-to-point protocols (such as HTTP and FTP). Therefore, freeCycles benefits from nodes' network bandwidth to distribute data. Additionally, freeCycles allows multiple MapReduce iterations to run without having to wait for a central server, i.e., data can flow directly from reducers to mappers (of the next iteration). Regarding the availability of data stored on volatile nodes, freeCycles proposes an enhanced scheduler that automatically replicates data or tasks to minimize the risk of stalling the MapReduce workflow (when waiting for the recomputation of some data that was lost due to some worker node failure).

By providing these functionalities, when compared to current solutions, freeCycles achieves: i) higher network scalability (reducing the burden on the data server network bandwidth), ii) improved dependability (by replicating tasks and data automatically), iii) reduced transfer time (improving the overall turn-around time), and iv) augmented fault tolerance since nodes (including the server) can fail during data transfers without compromising other nodes' transfers.

In short, the contributions of this work are the following:

- proposing and evaluating the idea of taking advantage of bandwidth available at remote computing nodes to help distribute data. To the best of our knowledge, freeCycles is the first system to take advantage of all the bandwidth available at remote working nodes in all phases of a MapReduce computation;
- a BOINC-compatible Internet-wide MapReduce computing platform which is able to run multiple MapReduce iterations efficiently (data does not have to go back to the server between iterations);
- enhanced data distribution (by using all available bandwidth at worker nodes through BitTorrent) and enhanced dependability/fault-tolerance (by using automatic data and task replication);
- set of performance tests which compares freeCycles with other approaches and highlights the performance and network scalability gains of this solution compared to previous ones. In addition, through some of our experiments, we draw some conclusions regarding the applicability of MapReduce to volatile pools and how to adapt MapReduce workloads for this specific environment.

The rest of this document is organized as follows. Section 2 describes the core architecture, Section 3 presents our data distribution algorithms and Section 4 presents our approach to handle fault tolerance and intermediate data availability. Section 5 describes some implementation details. We conclude the document

[3]This work is an extended version of a previous artcile published in a workshop[11].

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

with an extensive set of experiments (Section 6), and a section presenting the most relevant related work (Section 7), followed by some conclusions.

2 freeCycles Architecture

freeCycles is a MapReduce-enabled and BOINC-compatible computing platform. It provides the following main novel contributions: i) efficient MapReduce data distribution using the BitTorrent protocol, ii) efficient iterative MapReduce job support, and iii) automatic data and task replication (to guarantee the progress of MapReduce jobs). In this section, we describe freeCycles' architecture. As some of its basic components are already present in BOINC, we focus only on freeCycles' extensions.

2.1 Overview

freeCycles has two main entities: a central server (see Section 2.2), and many clients (see Section 2.3). The server schedules tasks to clients. Using the MapReduce model, each task is either a map or a reduce operation over some data. When a job is deployed, the scheduler starts by delivering map tasks. Reduce tasks are delivered once all map tasks are finished.

Using the MapReduce model, input data is first transformed (by mappers) into intermediate data. Intermediate data is then shuffled, and finally transformed (by reducers) into output data. The shuffle phase step is crucial for MapReduce workflow as it partitions intermediate data (according to some user-defined partitioning function) among reducers.

When running MapReduce applications in BOINC, all input, intermediate, and output data is stored in the data server (the data server is a component of the central server, as described in Section 2.2) which serves data to all mappers and reducers using, for example, HTTP or FTP. When mappers finish their tasks, intermediate data is uploaded to the data server. When all the intermediate data is available at the server, validation takes place, followed by the shuffle step. Then, in the reduce phase, reducers download intermediate data from the data server (using HTTP or FTP) and upload the output data back to the server, which validates it.

Since each task is replicated at least three times to tolerate stragglers and node failures (see Section 4 for more details), at least three copies of each input will be transmitted by the data server. To eliminate this overhead, freeCycles uses BitTorrent. Therefore, all input, intermediate, and output data are distributed using BitTorrent.

Thus, in freeCycles (using BitTorrent) each mapper or reducer is able to send input or output data to other replicas of the same task. Moreover, intermediate data is not sent to the data server. Instead, it is directly transfered from mappers to reducers (using the BitTorrent protocol, as explained in Section 3.2). The shuffle and validation steps only manipulate hashes of the intermediate data (see Section 3.2 for more details). With this approach, the data server only holds input data, intermediate data hashes, and the output data. Thus, with freeCycles, the burden of distributing data is shared among the server and computing nodes.

To cope with high node heterogeneity and node failure, freeCycles employs an automatic data and task replication mechanism. High churn or faults are

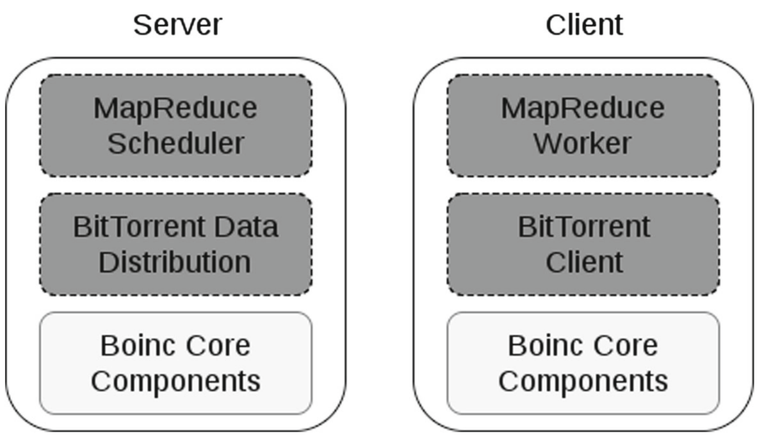


Fig. 1 freeCycles Server and Client Extensions Overview

specially harmful for MapReduce jobs because of the time dependency between the map and reduce phases (i.e., all intermediate data must be available before the reduce phase starts). Hence, replication of intermediate data or map tasks is necessary to keep intermediate data available while waiting for the reduce phase to start (more details in Section 4).

Figure 1 presents a high level overview of the components present in both server and client sides. Extensions regarding BOINC are presented in dark gray. Components that are reused from BOINC are presented in light gray. In practice, freeCycles adds (regarding BOINC) support for BitTorrent data distribution and for MapReduce workloads both at the server and client sides by adding components to support each of these features (BitTorrent data distribution and MapReduce workloads). Section 5 provides a more complete description over both the server and client side components. In the next sections, we give an overview over the architectural components of freeCycles.

2.2 Server Architecture

The server-side architecture (central server) is composed of several components (see the central server architecture in Fig. 2): i) a data server where input and output data are stored, and ii) a scheduler that has several responsibilities, such as creating, scheduling, and validating tasks. freeCycles adds two additional components: i) a BitTorrent tracker (to enable nodes to use the BitTorrent protocol to download and upload data to and from other nodes and the central server), and ii) a BitTorrent client (that is used to share the initial input and to receive the final output through the BitTorrent protocol).

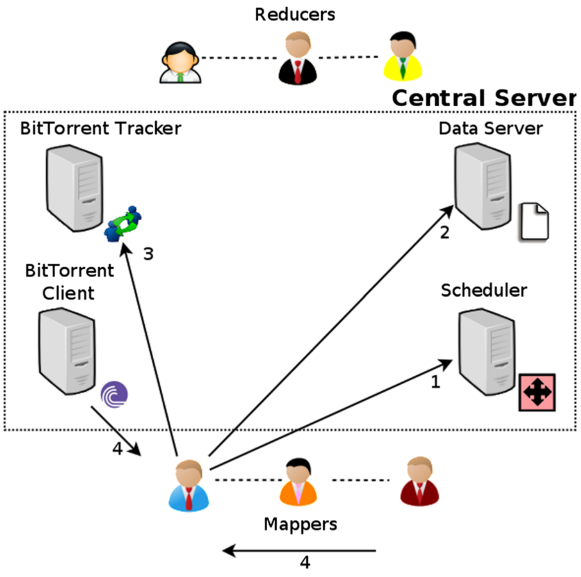


Fig. 2 Input Data Distribution

2.3 Client Architecture

For the client-side architecture, freeCycles reuses the BOINC client runtime (that manages all the issues related to server communication, process management, etc.). To support MapReduce applications and the BitTorrent protocol to distribute data, freeCycles augments the client software with a MapReduce framework and a BitTorrent client. Therefore, all nodes, either a map or a reduce node, can download and upload data to and from other nodes or the data server (this significantly decreases the used bandwidth and CPU load at the central server).

3 Data Distribution Algorithm

Having described the architectural components on both client-side and server-side, we now detail how freeCycles uses the BitTorrent file sharing protocol to coordinate input, intermediate and final output data transfers. By leveraging our data distribution algorithm, we show (in Section 3.4) how freeCycles is able to run multiple MapReduce iterations without compromising its scalability (i.e., avoiding high burden on the data server).

3.1 Input Distribution

Input distribution (see Fig. 2) is the very first step in every MapReduce application. Each worker node starts by asking the scheduler for a task (step 1); then, it downloads the hash of the input file (step 2), the `.torrent` file [5], and then uses it to search for other nodes that are also sharing that file (step 3). Afterwards, it starts downloading the input file from all nodes sharing the same input file (step 4).

[5] 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).

Note that the data server always shares all input files for all map tasks. Additionally, worker nodes that already have some file parts will also share them with other worker nodes interested in the same input file.

Thus, we 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 higher since a mapper may download input data from multiple sources (the data server and other mappers).

3.2 Intermediate Output Distribution

Once a map task is finished, the mapper has an intermediate output ready to be used. Figure 3 illustrates the steps for the intermediate data distribution. The first step is to create a hash of the intermediate file (step 1). From this point on, the mapper is able to share its intermediate data using the BitTorrent protocol: the BitTorrent client running at the computing node automatically informs the BitTorrent tracker (step 2), running at the central server, that some intermediate files can be accessed through this node. Then, the mapper notifies the server of the map task termination (step 3) by sending the intermediate file hash just created.

As more intermediate hash files arrive at the server, the server is able to decide (using a quorum of results) which mappers have the correct intermediate files by comparing the corresponding hashes (step 4). When all the intermediate outputs are available, the server shuffles all these files and prepares sets of inputs, one for each reduce task (step 5). When new nodes request work, the scheduler starts issuing reducer tasks (step 6). These reducer tasks contain references to the intermediate hash files that were successfully validated and that need to be downloaded (step 7). Once a reducer has access to these intermediate hash files, it starts transferring the intermediate files (using the BitTorrent protocol) from all the mappers that completed the map task with success (steps 8 and 9). Reduce tasks start as soon as all the needed intermediate values are successfully transfered.

3.3 Output Distribution

Given that reduce tasks are replicated at least on three nodes, it is possible to accelerate the upload of the final output files from reducers to the data server (see Fig. 4).

The procedure is similar to the one used for intermediate outputs. Once a reduce task finishes, the reducer computes a hash file for its fraction of the final output (step 1). Then, it informs the BitTorrent tracker that some output data is available at the reducer node (step 2). The next step is to send a message to the central scheduler containing the hash file and acknowledging the task termination (step 3). Once the scheduler has received enough results from reducers,

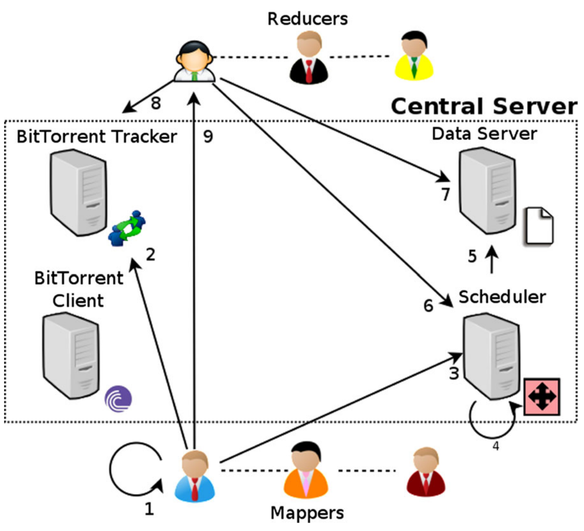


Fig. 3 Intermediate Data Distribution

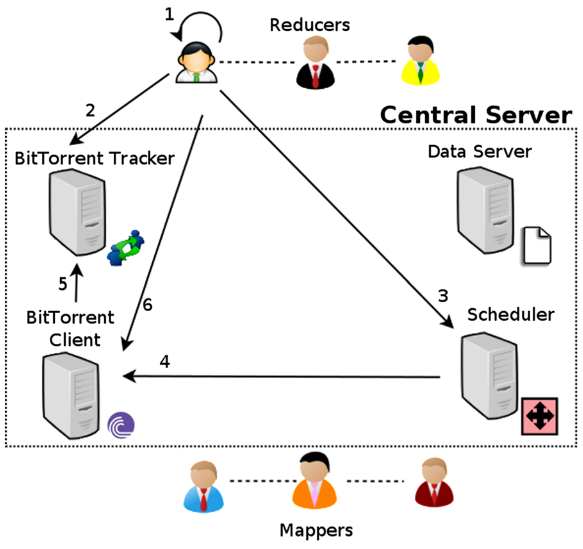


Fig. 4 Output Data Distribution

it can proceed with validation and decide which hash files will be used to download the final output. All the trustworthy hash files are then used by the BitTorrent client at the central server to download the final output (steps 4, 5, and 6).

Using BitTorrent to transmit the final outputs results in a faster transfer from reducers to the data server, a lower and shared bandwidth consumption from the nodes's perspective, and increased fault tolerance (since a reducer failure will not abort the file transfer as long as there is at least one reducer replica still alive).

3.4 Iterative MapReduce Workflows

Using the data distribution techniques just described, where the central server and all computing nodes have a BitTorrent client and use the BitTorrent Tracker to find peers with data, it is possible to use freeCycles to run applications that depend on multiple MapReduce iterations. The difference between our solution and previous ones (namely SCOLARS [18]) is that output data does not need 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 iteration to another).

From a mappers's perspective, when a work unit is received, the BitTorrent tracker is asked for nodes (which can be reducers or the central server) with the required data. It does not differentiate between the single iteration scenario (where the node downloads from the central server) or the multiple iterations scenario (where the node downloads from reducers of the previous iteration). 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 Section 5).

Figure 5 shows how computing nodes and the server interact to feed the new iteration with the output of the previous one (assuming that all steps in Fig. 4 are finished): 1) when new nodes ask for work, the scheduler delivers new map tasks with references to the hash files sent by the reducers (of the previous iteration), 2) the new mappers download the hash files from the data server, 3) after retrieving the hash files, each mapper asks the BitTorrent tracker for nodes that are sharing the reducer output data, and 4) mappers from the next iteration can now download output data from multiple reducers (from the previous iteration).

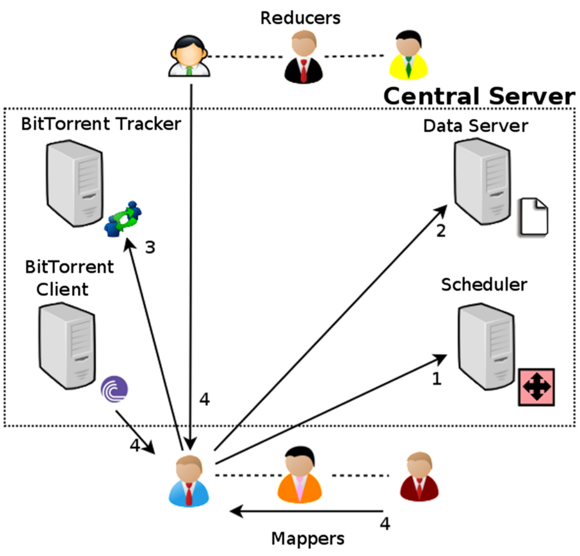


Fig. 5 Data Distribution Between MapReduce Cycles

4 Fault Tolerance and Intermediate Data Availability

freeCycles employs task replication to cope with stragglers and to tolerate node failures. The default replication value is three which means that, for each map and reduce task, three identical replicas are created and distributed to three different nodes.

For each task, the central server validates the output using a quorum of identical results. In other words, if two outputs are identical, the server does not wait for a third output and accepts the agreed upon outcome. If, on the other hand, all the received outputs are different, the validator does not accept them and the task is re-deployed.

Using this model, the MapReduce workflow advances as soon as at least two identical results are received. This is particularly helpful in the following scenarios:

- one of the three replicas is significantly slower than the other two;
- one of the three replicas fails (fail stop failure);
- one of the three replicas outputs wrong values (byzantine failure [13]).

Therefore, freeCycles supports at most one node failure (fail stop or byzantine) per task. If more node failures are to be supported, the number of task replicas would have to be updated. We do not cope with collusion attacks in which multiple malicious nodes try to subvert the system.

Previous studies [28, 47] show that the availability of intermediate data is a very sensitive issue for programming models 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 MapReduce is applied to Internet wide computing, where node churn is very high, such lack of replication leads to a loss of intermediate data. It has been shown that losing a single chunk of intermediate data incurs into a 30% delay of the overall MapReduce execution time [28]. To cope with this problem, and achieve better fault tolerance, freeCycles supports two methods:

1. Replicate map tasks aggressively when nodes designated to execute a particular map task take too long to answer the central server probes (periodic messages to assess the liveliness of the client). 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. As soon as a mapper is suspected to be failing, a new map task will be delivered to replicate the failed one.
2. Replicate intermediate data when there are intermediate outputs that have already been validated (by the central server) and some of the mappers that reported these results take too long to answer to the central server probes. Therefore, nodes 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 hash files and use them to start downloading intermediate data. Once the reduce phase starts, these new nodes can also participate in the intermediate data distribution phase, just like the mappers 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, and ii) intermediate data is normally smaller than input data. These two methods are applied before the reduce stage starts.

It is important to note that if a map task is not validated, it is not 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 be wrongly validated.

Another relevant aspect is that freeCycles does not discard results from nodes that were thought to be failing. In other words, if we receive results from a node that took too much time to report to the central server (but did not fail), we will consider the results and include them in the initial quorum of results.

Regarding the reduce stage, reducer nodes that take too long to answer the central server probes will also be replaced by other nodes until the computation is finished.

5 Implementation

This section describes how freeCycles is implemented and how to use our solution to create and distribute MapReduce applications over a computation pool. We further present a detailed description of all its components and how they cooperate with each other.

freeCycles is implemented directly on top of BOINC. It does not, however, change BOINC's core implementation since: i) it would create a dependency between our project and a specific version of BOINC, and ii) it would be impossible to have a single BOINC server hosting MapReduce and non MapReduce projects at the same time.

5.1 Server

Figure 6 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 (BitTorrent client and BitTorrent tracker) or modified (work generator and assimilator) by our solution.

As a normal BOINC server, freeCycles' server has two global components (a database and a data server) and several project specific daemons (work scheduler, work generator, validator, assimilator, etc).

In order to support the BitTorrent protocol, freeCycles adds two new global components: a BitTorrent client (to share the initial input and to retrieve the final output), and a BitTorrent tracker that is used by all BitTorrent clients, both on the server and client sides to know the locations of other BitTorrent clients sharing a desired file.

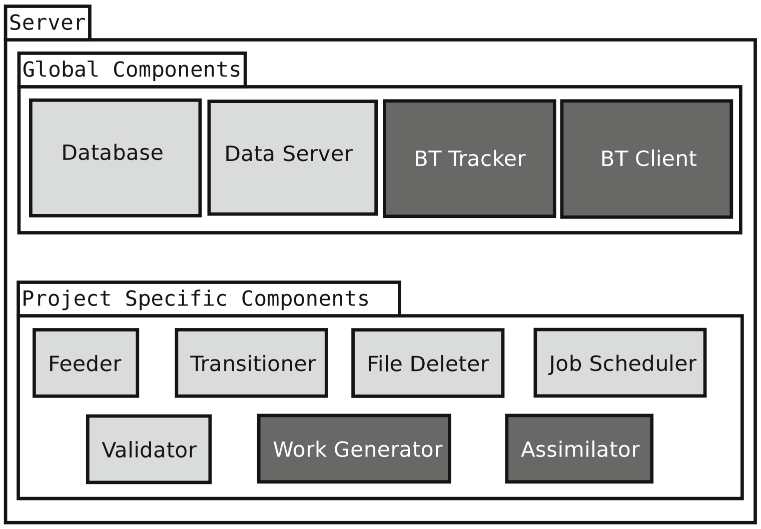


Fig. 6 freeCycles Server-side Implementation

Regarding the handling of MapReduce applications, freeCycles provides modified implementations for two of the project specific daemons, namely: the work generator (daemon that creates tasks), and the work assimilator (daemon that closes tasks when finished).

These two specific modifications are needed to introduce the dependency between reduce and map tasks (since the reduce phase can only start when all map tasks are finished). Therefore, when a task is finished, the assimilator moves the results to some expected location that is periodically verified by the work generator. The job configuration (input and output paths, number of mappers and number of reducers) is stored in a well known configuration file (that is automatically generated).

When all intermediate results (`.torrent` files) are present and validated, the work generator triggers the shuffle operation. This operation is responsible for assembling and assigning sets of intermediate results for each reduce task (since every map task typically produces data for every reduce task).

For example, consider the scenario of a MapReduce job with sixteen map tasks and four reduce tasks; each mapper splits its map output using the user-defined map function. Each one of the map output splits is used to create a `.torrent` file which is sent to the central server. For the sake of simplicity (but without loss of generality), assume that each map task produces four intermediate files (map output splits), one for each reduce task. After validating (see Section 4) all map tasks, the server has access to 64 `.torrent` files (four files from each map task). The shuffle operation consists in organizing these 64 files per target reducer. This results in sixteen groups of files, one for each reducer task. Each reducer is then responsible for downloading each one of the sixteen intermediate files and group all keys before calling the user-defined reduce function.

5.2 Client

Using freeCycles, all computing 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 nodes and possibly the central server.

To remain compatible with current BOINC clients, the freeCycles project is implemented as a regular BOINC application. Therefore, nodes that already use BOINC will be able to join a MapReduce computation without upgrading their client software. If we did not follow this approach, nodes 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, they cannot be used without forcing users to upgrade their client software.

freeCycles's client side application is meant to be used as a framework, i.e., developers would simply call freeCycles's code to register the map and reduce functions. All other issues related to managing map and reduce task execution, downloading and uploading data, is handled by our system.

Notwithstanding, 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 may be easily implemented as well.

Figure 7 shows a graphical representation of the freeCycles client implementation. The light gray box, BOINC Client Runtime, is the core component in the client software. All nodes 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:

– freeCycles Application: 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).
– Data Handler: 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 data distribution protocol (BitTorrent). This API, however, does not depend on the protocol (so it can be used for multiple protocols).
– BitTorrent Client: low level and protocol specific component. It is implemented using an open source BitTorrent library (libtorrent[6]).
– 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.

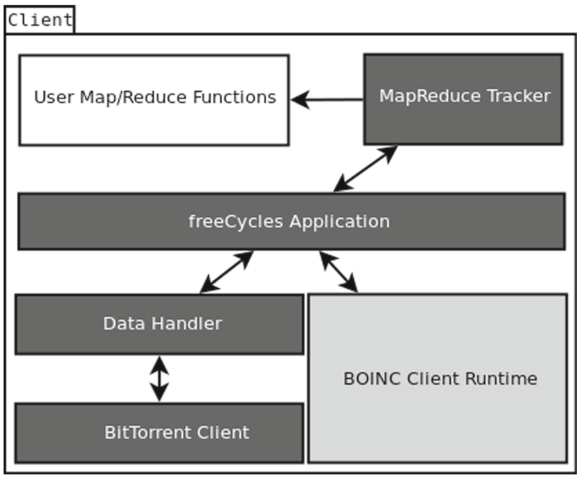


Fig. 7 freeCycles Client-side Implementation

To create and use an application, one would only need to provide a map and a reduce function implementation (white box from Fig. 7). These functions are then registered in our MapReduce Tracker module, and thereafter are called for all keys and value pairs.

6 Evaluation

In this section, we go through an extensive set of evaluation experiments. We compare freeCycles with SCOLARS (a BOINC-compatible MapReduce computing system), and BOINC, one of the most successful Volunteer Computing platforms. This makes it a good reference for performance and overall scalability comparison for an Internet-wide computing system. We use several representative benchmark applications, and different environment setups that verify the performance and network scalability of freeCycles.

6.1 Evaluation Setup

To conduct our experiments, we use a set of university laboratory computers equipped with Intel(R) Core(TM) i5-3570 CPUs and 8 GBs of RAM. Each node is connected in a 1Gbps network. In order to be more realistic, most of our evaluation is performed with throttled upload bandwidth, i.e., to simulate Internet connection bandwidths. This is an important restriction since Internet Service Providers tend to limit users' upload bandwidth (the ratio between download and upload bandwidths usually goes from five to ten, or even more).

[6]libtorrent is an open source implementation of the BitTorrent protocol. It is available at libtorrent.org

Table 1 Evaluation experiments summary

Section	Benchmark	Mappers	Reducers	Repl.	Input	Bandwidth
6.2.1	grep	16	4	3	512MB	10Mbps
6.2.2	word count	16	4	3	512MB	10Mbps
6.2.3	terasort	16	4	3	512MB	10Mbps
6.3	word count	16	4	3	256-2048MB	10Mbps
6.4	word count	16	4	3	512MB	5-100Mbps
6.5	page rank	16	4	3	512MB	10Mbps
6.6	word count	16	4	3	512MB	10-1000Mbps
6.7	word count	16	4	2-7	512MB	10Mbps
6.8	word count	100	15	3	1024MB	unlimited
6.9,6.10	word count	16	4	3	2048MB	10Mbps

During our experimental evaluation, all MapReduce workflows use sixteen map tasks and four reduce tasks both with a replication factor of 3 (except when explicitly said in contrary). All map and reduce tasks run on different nodes (to simulate what would probably happen in a regular deployment). Thus, we used a total of 60 physical nodes (48 map nodes and twelve reduce nodes).

For most tests we use 10 Mbps for upload bandwidth with unrestricted download bandwidth (the maximum link capacity is 1 Gbps). Such upload connection bandwidth is common in typical Internet connections. The download bandwidth is usually not a performance bottleneck, so we do not limit it.

By default, the MapReduce application running for most of this performance evaluation is the Word Count benchmark. Word Count (described in more detail in the following section) is a popular MapReduce benchmark that produces medium sized intermediate data.

Most experiments are executed with a 512 MB input file (we explicitly alert when this is not true). This means that, using a replication factor of three (which is the default value for our experiments), the central server sends approximately 1.5 GB of data through the network before the computation starts. Although this is a very small input size for a clustered environment, this is representative of the normal file sizes used in real deployments of Internet-wide computation (such as BOINC projects for example). This comes from the fact that Internet-wide computation normally has high compute ratio meaning that a small amount of data takes a lot of computation effort to process. However, since we are focused on evaluating the data management system, we use benchmarks that have a low compute ratio (i.e., the application runtime is dominated by the data transfer time). Increasing the size of the input file or using other benchmarks (with higher compute ratio) would not bring any novelty (we have done such experiments in the past). When increasing the size of the input file, the performance gap between each solution will naturally increase as well. When using experiments which take longer to process the input file, all solutions will have their computation times increased. For a specific experiment (varying the task replication factor), we also use nodes from PlanetLab [17] as described in Section 6.8.[7]

Table 1 presents a summary of the evaluation experiments. The table presents the section where the experiment is presented, the benchmark used for the experiment, the number of map and reduce tasks, the replication factor for tasks, the size of the input data and the available upload bandwidth.

6.2 Application Benchmarking

For benchmark testing, we selected a set of MapReduce representative benchmark applications to compare our solution with SCOLARS and BOINC.

From the data handling perspective (and according to some previous work [1]), each one of the three selected benchmarks belongs to a different MapReduce application class: small intermediate output

[7]PlanetLab is a global research network that supports the development of new network services. It is available at www.planet-lab.org

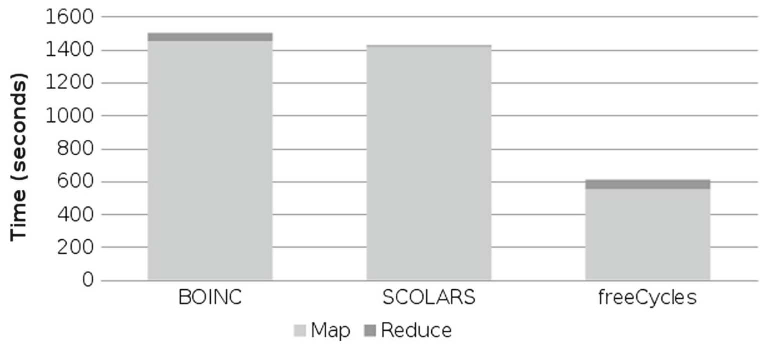


Fig. 8 Grep Benchmark Application

(Grep), medium intermediate output (Word Count), and large intermediate output (Terasort).

For each benchmark we present the results for the map and reduce phases individually. Note that the map phase starts when the central server starts delivering map tasks, goes through the distribution of input data, and ends when all map tasks are finished. The reduce phase starts right after the map phase ends; the central server starts issuing reduce tasks and reduce nodes start downloading intermediate data from map nodes (shuffle step). The reduce phase ends when all reduce tasks are finished and the final output is uploaded to the data server.

6.2.1 Grep

Our first benchmark application is Grep. Much like the application with the same name in the 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 Fig. 8, 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 computing nodes to help the server distributing the input.

A slight overhead can be noticed in the reduce phase for our solution (w.r.t. SCOLARS). This stems from the fact that the intermediate output is so small that the time needed to distribute all the intermediate and final output is dominated by the BitTorrent protocol overhead (contact the central tracker, contact several nodes, wait in queues, etc).

6.2.2 Word Count

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

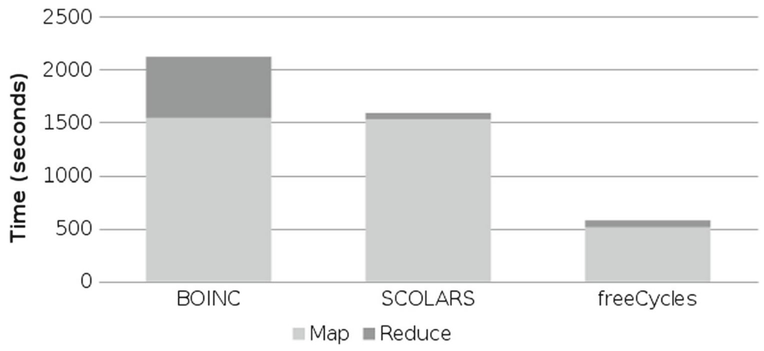


Fig. 9 Word Count Benchmark Application

Figure 9 shows the results of running Word Count. As intermediate data is larger than in the previous application (186 MB of generated intermediate data versus 2 MB, in Grep), BOINC performance is worse when 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.

6.2.3 Terasort

The last benchmark application is Terasort. Terasort is yet another famous benchmark application for MapReduce platforms [50]. At a very high level, it is a distributed sorting algorithm that: i) divides numbers in smaller chunks (with certain ranges), ii) sorts all chunks, and iii) combines all sorted chunks. We developed a simple implementation of this algorithm to be able to compare the performance of freeCycles with other systems.

In addition to being a very popular benchmark, Terasort is also important because it generates large volumes of intermediate and output data.

Looking at Fig. 10, it is possible to see a long reduce phase in BOINC. This results from 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, freeCycles suffered a slight increase in the reduce phase duration time.

Despite the size of intermediate data (512 MB), freeCycles is able to distribute intermediate data faster than previous solutions. It is almost five times faster than BOINC, and three times faster than SCOLARS.

6.2.4 Results Analysis

We now analyze the experiments presented in the previous sections to identify the factors that justify the benchmark performance results regarding freeCycles.

Starting with the map phase, it is important to note that all three benchmarks achieve similar performance results for this phase. This is explained by the fact that all benchmarks have the same input file size (512 MB). The initial map input transfer is very costly, and takes almost all the time associated to the map phase. Transferring 512 MB using a 1.25 MB/s connection takes at least 410 seconds. Then, we have the fact that these 512 MBs are replicated three times; even though BitTorrent is able to efficiently distribute the file, it takes a little more time to distribute all data, with replication, than without it. Other sources of overhead are the following: a) time to prepare and run the map task at the worker node; b) time to compute the hash of the intermediate output and send it to the data server; c) BitTorrent peer discovery and exchange time. We measured the time it takes for the application to run at the worker node and it takes around 5 seconds for each benchmark (including preparing the environment to run, running the benchmark, and hashing the intermediate data). In total, the map phase (comprehending the input file distribution, map task runtime, hashing the intermediate data and sending the hash back to the data server) takes on average 520 seconds.

Regarding the reduce phase (comprehending intermediate date distribution, reduce task runtime, validating the final output and sending it to the data server), each benchmark leads to different performance results. This results from the fact that each

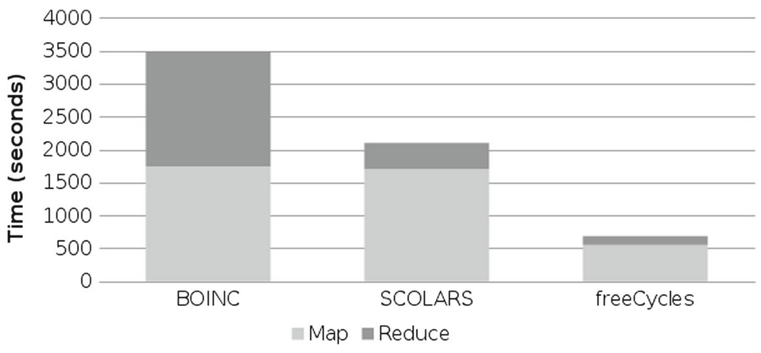


Fig. 10 Terasort Benchmark Application

benchmark produces different amounts of data in the map phase. Grep has the lowest time for the reduce phase since it has the lowest amount of intermediate data to distribute and process (2 MB). Word Count follows with 186 MB of intermediate data, and Terasort has the longest reduce phase that results from the 512 MB of intermediate data that needs to be distributed and processed.

In conclusion, data transfers associated with the MapReduce workflow dominate the job runtime; this is expected since we chose low compute ratio benchmarks to emphasize the data transfer overheads inherent to these Internet-wide computation systems. In particular, the data transfers associated with the map phase are particularly costly because all data originally comes from one single data server.

6.3 Varying Input File Size

For the next experiment, we change the input file size. In the previous scenarios 512 MB files were used. For this experiment, we start with a smaller input file (256 MB) and increase the file size until 2048 MB. The reader might notice that, after replication, the amount of data in the network is tripled.

The reason for this experiment is to show how computing platforms behave when larger input files are used. Typical Internet-scale projects use small input files that require lots of computation (high compute to communication ratio). However, as previously mentioned, typical MapReduce applications need large input files that, most of the time, will be consumed quickly.

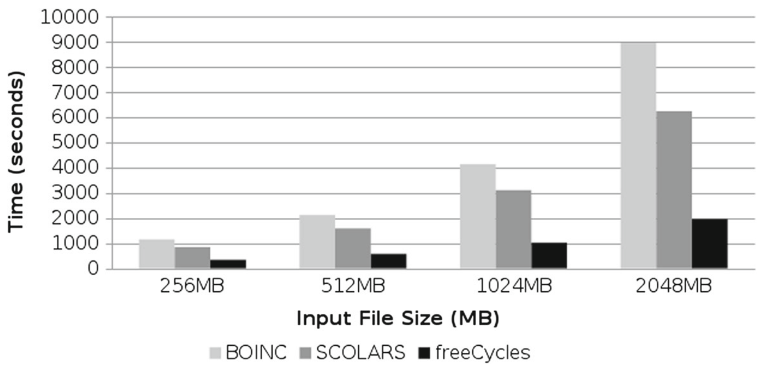
In our implementation, we divide input data equally among all map tasks. Therefore, all mappers will have a very similar input file size to process (and consequently, will have similar execution times). We are also cautious to make sure that intermediate keys are evenly distributed. The average standard deviation in the size (MBs) of the intermediate output produced by each mapper is of 0.12 (average of 0.471 MBs) for Grep, 5.51 (average of 51.5 MBs) for Word Count, and 12.88 (average of 136 MBs) for Terasort.

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 10 Mbps and we maintain the MapReduce configuration of sixteen mappers and four reducers (all replicated three times).

Figure 11 shows the results for this experiment. BOINC is the system with the worst execution times. SCOLARS is able to lower the time by using inter-client transfers, which effectively shortens 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 three times faster than SCOLARS), showing improved scalability regarding input file size.

Since the job runtime is dominated by the transfer times involved in the MapReduce workflow, doubling the amount of data fed to the MapReduce job will duplicate the amount of input, intermediate, and output data. This has a direct impact in the MapReduce job runtime. In practice, the MapReduce job runtime depends linearly on the size of the MapReduce input file. The difference between all the three systems is the rate at which the job runtime grows. freeCycles has the lowest rate, followed by SCOLARS. BOINC is the system that presents the highest (worst) runtime growing rate.

Fig. 11 Performance varying the Input File Size



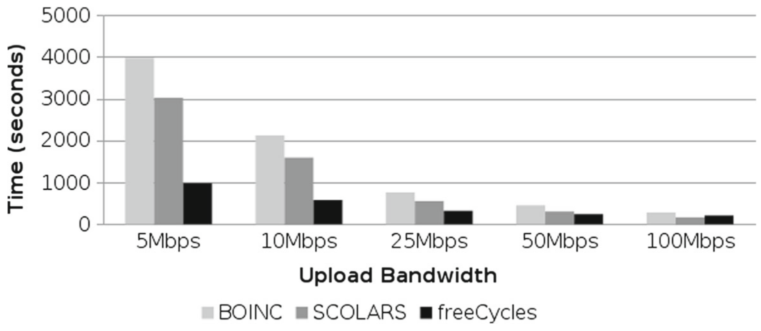


Fig. 12 Performance varying the Upload Bandwidth

6.4 Varying Upload Bandwidth

Another important restriction on real-life Internet-wide 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 tasks that can be delivered at any moment.

In order to evaluate the behaviour of different systems with varying upload bandwidth, we present this evaluation scenario in which we executed the Word Count application with a 512 MB input, while changing the upload bandwidth. We keep the sixteen mappers and four reducers configuration for all runs.

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

In Fig. 12, 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 512 MB). Hence, freeCycles, for example, always has a small overhead from using BitTorrent (delay to contact the central tracker, contact several nodes, delay in priority queues, etc). This overhead is particularly noticeable in the last scenario (with 100 Mbps) where SCOLARS performs better (since it has far less overhead) and the overall time for freeCycles' execution is dominated by BitTorrent overheads.

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

With the same data from this experiment, we measure the aggregate usable upload bandwidth in: i) distribution of input data to mappers (Fig. 13), and ii) distribution of intermediate data to reducers and distribution of output data to the data server (Fig. 14). Each figure presents values for different limits of upload bandwidth (remember that this limit is placed on each individual node used in this experiment). The values

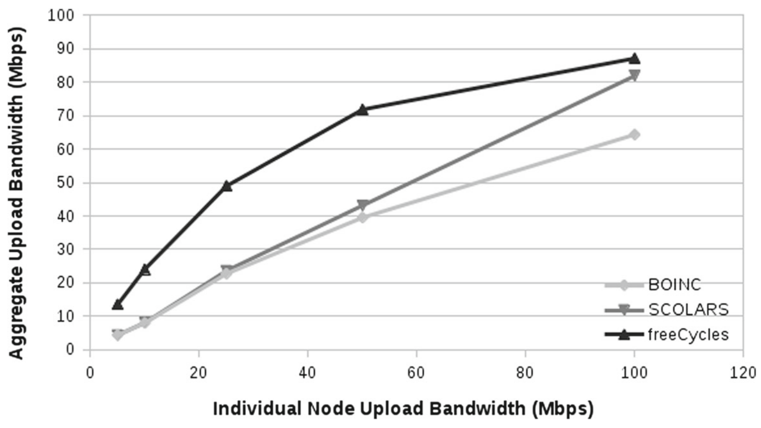


Fig. 13 Aggregate Map Upload Bandwidth

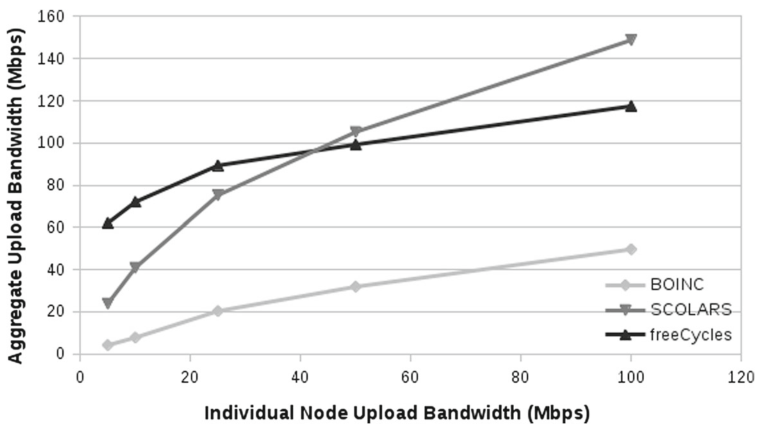


Fig. 14 Aggregate Reduce Upload Bandwidth

of the y-axis result from dividing the total size of input (512 MB), in Fig. 13, or intermediate and final data (189 MB), in Fig. 14, by total time to distribute data (which depends on how each system distributes data). The result is the aggregate usable upload bandwidth. Note that for some scenarios (for example, freeCycles with 10 Mbps), the usable upload bandwidth is superior to the individual node upload bandwidth (three times for freeCycles with 10 Mbps). This experiment basically measures how much available upload bandwidth from worker nodes can freeCycles use to distribute data. This experiment also shows that for upload bandwidths greater than 70 Mbps, SCOLARS is able to generate more aggregate upload bandwidth. We investigated the source of this potential scalability limitation and we found out to be a combination of two factors: i) the actual time each transfer takes is small (as we increase the upload bandwidth, we maintain the input file size, 512 MB), leading to ii) large percentage of the transfer time is lost in BitTorrent peer discovery and exchange overheads.

It is possible to conclude that BitTorrent presents network scalability benefits when the the upload bandwidth at the server is a bottleneck. In this situation, BitTorrent enables freeCycles to take advantage of available bandwidth at worker nodes to distribute data.

6.5 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 iterations (such as the original Page Ranking algorithm).

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

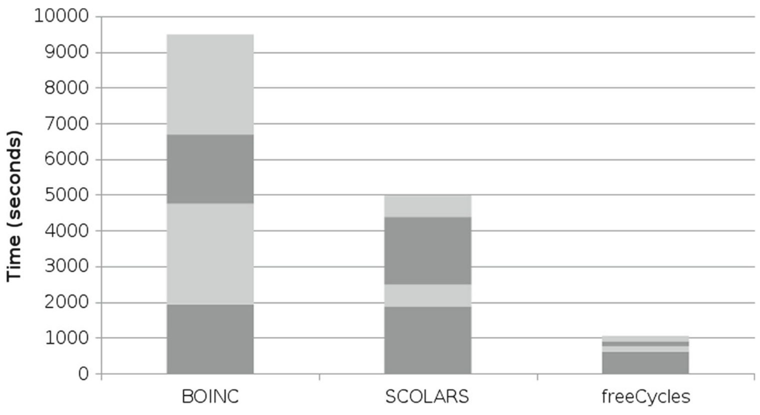


Fig. 15 Two Page Ranking Cycles

We keep the setup used in previous experiments: upload bandwidth throttled to 10 Mbps, input file size of 512 MB, sixteen map tasks, four reduce tasks, and three 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 Fig. 15. 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 iterations, 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 (the same applies to BOINC). In other words, the execution time increases linearly with the number of iterations.

freeCycles presents the most interesting results, i.e., it is much more scalable with regards to the number of MapReduce iterations (the execution time does not increase linearly with the number of iterations). 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 does not need 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.

6.6 Comparison with Hadoop Cluster

Another interesting experiment to perform is to compare the execution times for the three benchmark applications on: i) computation pool (freeCycles), and ii) Hadoop cluster (Hadoop [50] is an open source MapReduce implementation).

To that end, we use a cluster of 60 machines (the same number of machines as in our computation pool) in which ten machines also play as datanodes (datanodes are nodes that are hosting the distributed file system, HDFS [10]). All nodes have four cores, a total 8 GB of RAM, and are interconnected with 1 Gigabit Ethernet.

For this experiment, we run the three application benchmarks: Grep, Word Count, and Terasort. We did not build these applications for Hadoop, but instead, we use the implementation provided with the platform. Regarding the freeCycles platform (computation pool), we use the same implementation as in previous experiments. All applications run with sixteen mappers and four reducers (replicated three times when using freeCycles).

Results are shown in Table 2. From these results, it is possible to conclude that a MapReduce application deployed on a cluster runs approximately six times faster than the same application deployed on a computation pool. This is the performance price that it takes to port an application to Internet-wide computing systems.

Nevertheless, it is important to note two factors that motivate this big performance discrepancy (between the cluster and the computation pool): i) we limited the bandwidth to 10 Mbps on the computation pool while the cluster nodes where connected via 1 Gbps links, and ii) input data was previously distributed and replicated amongst ten datanodes (in the Hadoop cluster) while our freeCycles deployment only had one data server.

Another important difference between Hadoop and freeCycles is the moment when the shuffle step takes place. Hadoop starts shuffling intermediate data before all map tasks are complete. In freeCycles this is difficult because we follow a pull model, common in Internet-wide computation systems. In these systems, worker nodes contact a server asking for work, not the other way around. Therefore, imagine that we already have some map tasks finished; we cannot deliver a reduce task with partial intermediate data and send the rest of the intermediate data later. Thus, we must wait until all map tasks are finished to deliver reduce tasks which contain the complete portion of intermediate data for that particular reduce tasks. Therefore, we only start the distribution of the intermediate data

Table 2 Benchmark execution times

Benchmark	Hadoop cluster	Computation pool
Grep	102 sec	610 sec
Word Count	106 sec	578 sec
Terasort	92 sec	686 sec

(shuffle step) after all map tasks are finished. Nevertheless, this is an interesting topic for future work as it would lead to interesting performance improvements therefore increasing the attractiveness of our solution.

In conclusion, there is a trade-off between an environment with limited size and high cost but highly clustered (all nodes are very close to each other, i.e, data transfers between nodes are very fast) or a very large environment, with low cost, using Internet-connected working nodes (which result in slower data transfers between nodes).

Please note that we do not evaluate neither freeCycles in a cluster environment nor Hadoop in a volatile environment. Both systems (freeCycles and Hadoop) are designed and optimized to work in specific environments. For example, the data distribution protocol that freeCycles uses (BitTorrent) is very inefficient for short data transfers (transfers that take only a couple of seconds) as it takes potentially more time exchanging control messages than real data. On the other hand, testing Hadoop in a volatile environment would be unrealistic for a number of factors: i) it does not replicate tasks by default, only when some task fails (and it is known that Hadoop has terrible performance in these situations [20, 36]); ii) Hadoop is designed using a push model (which is difficult to use in a volatile environment since we might not know all resources before hand) instead of a pool model (tasks are pushed into workers); iii) HDFS uses Point-to-Point data transfers.

6.7 Varying the Task Replication Factor

In this experiment, we show how BOINC, SCOLARS, and freeCycles behave when increasing the map task replication factor. In previous experiments this value was, by default, three (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 four and one, respectively.

This is a particularly important experiment to analyze how current platforms, including freeCycles, support a larger replication factor. A high replication factor is fundamental to improve the resilience of the MapReduce computation in environments with high churn. For this experiment we are specially interested in increasing the map task replication factor to provide good availability for the intermediate data.

Figure 16 presents the overall MapReduce turnaround time. 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 nodes' 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 nodes will be available to upload intermediate data leading to a faster reduce phase. In other words, while BOINC and SCOLARS experience a linear increase in time when we increase the replication factor, freeCycles does not (meaning that the execution time does not depend on the replication factor).

6.8 PlanetLab Deployment

In this specific experiment, we use a different execution environment. Instead of using a set of nodes in our local cluster, we use nodes from PlanetLab [17].

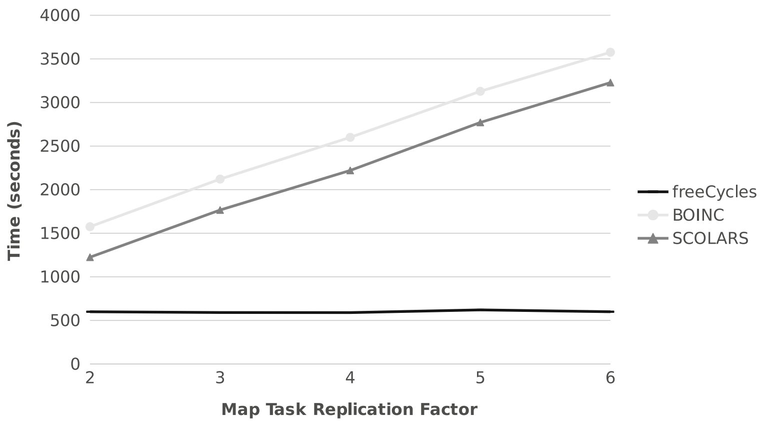


Fig. 16 MapReduce Total Finish Time for Different Map Task Replication Factors

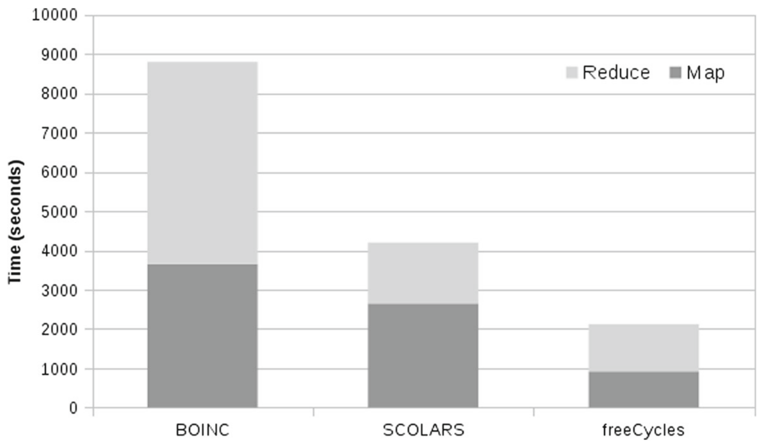


Fig. 17 Word Count in PlatnetLab

The main difference between running MapReduce on PlanetLab and in our cluster is that we do not have to artificially limit upload bandwidth; this means that with PlanetLab, we may obtain more realistic results. Besides that, in PlanetLab, nodes capacity is not uniform, upload bandwidth available at each node is also not uniform, and the latency is much higher than in our previous environment. These factors contribute for a longer job runtime for all solutions when compared to the runtimes achieved in our previous environment.

For this experiment, we run the Word Count benchmark with an input file of 1 GB. We use 50 worker nodes (all from PlanetLab), and we deploy a MapReduce computation with 100 map tasks and 15 reduce tasks. The dataset we used for this experiment produces approximately 1 GB of intermediate data. Figure 17 presents the results for this experiment.

As with the previous experiments, BOINC has the worst performance, followed by SCOLARS. freeCycles achieves the best performance in both phases, map and reduce. Compared to previous Word Count experiments, the reduce phase in this experiment is significantly longer. This results from the fact that the dataset used in this experiment produces much more intermediate data then the previous one.

6.9 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 computation pool, we now use a virtual scenario where we try to approximate a real world environment. To do this, we developed a simulated environment (a Java application) that simulates all the elements present in a real freeCycles deployment and all the interactions between them. This tool is used to estimate the performance of freeCycles in a real world deployment (in which new nodes can join and leave the computation pool at any time).

The main goal of this experiment is to assess how the node churn rate affects the performance of our solution. Remember that the churn rate is the number of nodes that join or leave the network in a fixed amount of time. Therefore, to emulate it we manipulate two settings in our simulator: the average node session time (it is important to note that each node will have a different session time in a real world deployment), and the new node rate (how many nodes join the network in a fixed amount of time). To introduce some indeterminism, we use a normal distribution (which is one of the most used distributions to model node churn [9, 45]) to calculate the node session time for each node.

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 [9, 45] on peer-to-peer node churn (in BitTorrent networks in particular), we decided to use a session time of two hours (average), and a new node rate of one per minute. As with the previous experiments, we performed a MapReduce workflow with sixteen map tasks and four reduce tasks, all with a replication factor of three. The input data has 2 GB of size. The upload bandwidth is always limited to 10 Mbps (1,25 MBps).

Figure 18 presents the results of freeCycles for two different environments: a computation pool, and a cluster. In a computation pool, nodes can leave and

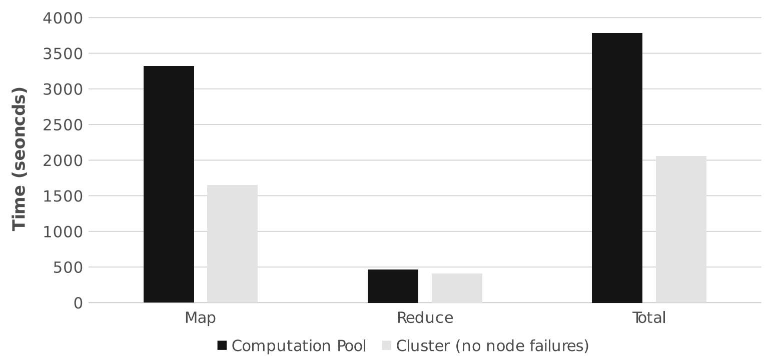


Fig. 18 Environment Comparison using Word Count Benchmark

join the workflow (node churn) while in a cluster, all nodes participate in the workflow (and do not fail).

From Fig. 18, we can conclude that the performance difference between the two environments is still significant. The total time almost doubles when it is moved to a computation pool, where node churn is a real problem.

We further demonstrate the effects of higher and lower churn rate in Table 3. Results were obtained with the same job parameters used in the previous experiment (Fig. 18). The difference is that we now manipulate the node session time and new node rate to change the network churn rate (in the previous experiment, these variables are fixed).

Results from Table 3 show that the time needed to complete a MapReduce job is reduced by: i) increasing the node session time, and ii) increasing the new node rate (number of new volunteers per minute). The interesting result is the one obtained for one hour of session time, and one new node per minute. Our simulation shows that, for this specific job characteristics: i) there are not enough connected nodes, and ii) nodes' session time is too short to complete the job. Caused by these two factors, most of our runs never end because the few nodes that exist fail too often, and the MapReduce workflow never completes.

One important conclusion to take from this experiment it that each MapReduce job should be carefully designed to keep the overall job execution time below the average node session time. The best way to achieve this, and still be able to process large amounts of data, is to partition jobs in many relatively small tasks (which will take less time to complete than large tasks). This is a very simple way (which has also been proposed in other works [7, 26]) 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.

6.10 Varying the Replication Timeout

The last experiment shows how much time one should wait to 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 computation pool), and used an increasing number of seconds for the time we wait before we consider that a node has failed. As soon as a node is detected to be failing,

Table 3 Real world simulations

Session time / New node rate	1 vol/min	10 vol/min	100 vol/min
1 hour	Infinite symbol	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

the corresponding 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 shown in Fig. 19, we are able to confirm our prediction: 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 hampered by the addition of replicas (see a detailed explanation in Section 6.7). This is not true for other platforms, such as BOINC and SCOLARS. Note that these results were obtained using our simulator. In a real world setting, using a replication timeout too low (one second for example) would be unwise since Internet connections can have latency spikes that can go up to a few seconds.

6.11 Final Analysis

Throughout the evaluation section, we showed that freeCycles is able to perform better than current systems (BOINC and SCOLARS) mainly because it leverages bandwidth available at volatile nodes, thus reducing the time spent on distributing data.

We also show that running a MapReduce job in a cluster is faster than in a volatile pool mainly because we have to pay the price of sending data through slow links (Internet) and this confirms that compute bound workloads are a better fit for Internet-wide scenarios than data bound workloads.

Regarding the applicability of MapReduce to wide-area scenarios (studied in Section 6.9) we are able to conclude that MapReduce jobs should be carefully designed to keep the overall execution time below the average node session time. Otherwise the job might fail to complete because nodes fail (for example, the map-reduce barrier is difficult to pass if map workers start to fail).

7 Related Work

In this section we analyse and discuss systems that are similar to freeCycles. In addition, we analyse and discuss some data distribution protocols that we considered when designing freeCycles.

7.1 Computing Platforms

Most of the existent Internet-wide computing platforms are focused and optimized to run Bag-of-Tasks A2HA - automatic and adaptive host allocation utility computing for bag-of-tasks' JISA 2011 (i.e., embarrassingly parallel applications), and thus cannot support MapReduce applications. Nevertheless, as MapReduce's popularity increased, some platforms decided to use available resources over the Internet to run MapReduce jobs.

Several of these platforms (that already support MapReduce) are specialized in deploying MapReduce jobs on multiple clusters/clouds connected through the Internet ([12, 16, 27, 31]). For example, in Kailasam [27], the authors use streaming to hide high latency derived from using Internet connections to transfer MapReduce data between two separate clusters. In Li [31], the authors propose a scheduler that decides where to place tasks according to the amount of necessary input and the amount of produced output (the goal is to minimize data transfers between separate clusters).

Although the previously mentioned systems are able to deploy Internet-wide MapReduce computations, they do not cope with all the goals we set for

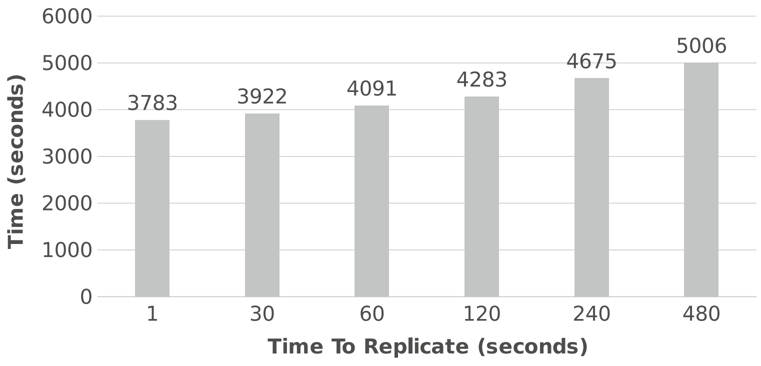


Fig. 19 Performance Evaluation Varying the Replication Timeout

freeCycles. In particular, these systems assume that nodes and the connection between them are reliable. Another important difference between freeCycles and these systems is that we can not assume that only one connection will go through the Internet, and all other connections are reliable, have low latency and high bandwidth. In conclusion, we aim at allowing any Internet-connected device to join the computation (and not only specific nodes from clusters).

Solutions such as SCOLARS [18], MOON [33], Tang [46], and Marozzo [35] share some goals with freeCycles as they support MapReduce applications and allow arbitrary nodes to join the computation.

MOON (MapReduce On Opportunistic Environments) is an extension of Hadoop (an open source implementation of MapReduce). MOON ports MapReduce to opportunistic environments mainly by: i) modifying both data and task scheduling (to support two types of nodes, stable and volatile nodes), and ii) performing intermediate data replication. However, although MOON was designed to run MapReduce tasks on volatile nodes, it still relies on a large set of dedicated nodes (mainly for hosting dedicated data servers). This assumption does not hold in a volatile computing setting as the Internet (where the availability of such dedicated resources can not be garanteed).

The solution presented by Tang [46] is a MapReduce implementation focused on desktop grids. It was built on top of a data management framework, Bitdew [22]. Even though BitDew supports BitTorrent, the authors do not mention it or even evaluate it. Nevertheless, this solution presents the same problem as MOON: it relies on the high availability of several central services, which is prohibitive in large pools connected through the Internet.

Marozzo [35] presents a solution to exploit the MapReduce model in dynamic environments. The major drawbacks of this solution are: i) data is distributed point-to-point (which fails to fully utilize the node's bandwidth), and ii) there is no intermediate output replication.

SCOLARS (Scalable Complex Large Scale Volunteer Computing)[18] is a modified version of BOINC that supports MapReduce applications, and presents two contributions: i) inter-client transfers (for intermediate data only), and ii) hash based task validation (where only a hash of the intermediate output is validated on the central server). However, it presents the same drawbacks as the previous solution: only point-to-point transfers, and no intermediate data replication.

In conclusion, all the analysed solutions present problems that invalidate them as candidate solutions for the problem we are addressing. Table 4 provides a taxonomy of the discussed systems. For each system, we show the target environment, the data distribution strategy, and if the system supports MapReduce jobs or not. It is therefore possible to conclude that no system is able to provide Peer-to-Peer data distribution (i.e., have multiple nodes helping to distribute data) in a volatile pool and also provide MapReduce support.

In addition, Internet-wide solutions such as BOINC and SCOLARS do not handle faults adequately in the sense that they do not provide means to keep intermediate data available. Additionally, no solution showed that it was able to fully utilize the node's upload bandwidth, or that it was possible to run multiple MapReduce iterations, while avoiding a bottleneck on the data server.

7.2 Data Distribution Protocols

In this section we discuss possible existing protocols that could be used in an Internet-wide system to be able to use distribute data. Since we are interested in supporting MapReduce workflow, protocols should be able to efficiently distribute data in all stages of the MapReduce workflows. More specifically, protocols must support efficient data distribution in: one to all,

Table 4 Internet-wide computing platforms taxonomy

System	Environment	Data distr.	MR Support
BOINC	Volatile Pool	Point-to-Point	No
SCOLARS	Volatile Pool	Point-to-Point	Yes
MOON	Desktop Grid	Point-to-Point	Yes
work by Tang	Desktop Grid	Peer-to-Peer	Yes
work by Marozzo	Desktop Grid	Point-to-Point	Yes
XtremWeb	Desktop Grid	Peer-to-Peer	No

all to all, and all to one data transfers. Please also note that, since freeCycles is targeted to volatile environments (where nodes fail frequently), we assume that all MapReduce tasks are replicated. We begin our discussion by considering the difference between Point-to-Point protocols and Peer-to-Peer protocols and why we chose Peer-to-Peer protocols. Then we discuss some existing Peer-to-Peer protocols.

7.2.1 Point-to-Point VS Peer-to-Peer

Point-to-Point protocols (for example FTP) are typically used to transfer data from one node to another. On the other hand, Peer-to-Peer protocols are typically used to transfer data from one (or possibly many) nodes to one (or possibly many) nodes. For example, if some node has some data that other nodes want, data can be transferred to one or more nodes. These nodes (that received data from the first node) can then start also sharing the data with other nodes that still want the data. In other words, while in Point-to-Point protocols there is only one source node which is responsible for sending data to all nodes, in Peer-to-Peer protocols, all nodes can work as source node and send all the data they already have locally.

If we think about this, we realize that Point-to-point algorithms take O(D*N) seconds to distribute all data, where D represents the size of the data to distribute and N represents the number of times D needs to be sent to different nodes. On the other hand, Peer-to-Peer protocols take O(D) seconds to distribute the same amount of data. This is because the protocols harness available bandwidth at remote nodes to distribute data.

In conclusion, we find Peer-to-Peer protocols more suitable for MapReduce data operations, namely: i) one source to many nodes (map input distribution), ii) many sources to many nodes (intermediate data distribution), and iii) many sources to one node (output distribution). Also note that Peer-to-Peer only makes sense when we consider replication. If freeCycles did not need to replicate worker tasks, every worker node would receive a different piece of data and therefore there would be no opportunity for sharing with other worker nodes.

7.2.2 Peer-to-Peer Protocols

Regarding Peer-to-Peer protocols, we only analyze those that we found to be more relevant and closer to our objectives. Thus, other systems such as Gnutella [40], FastTrack [32], eDonkey2000 [25], and OceanStore [29] are not addressed. Therefore, in this section, we only discuss two protocols that are relevant to our objective (distribute large amounts of data efficiently within a large set of volatile nodes): FastReplica [15], and BitTorrent [37].

FastReplica is a replication algorithm focused on replicating files in Content Distribution Networks (CDNs). It was designed for use in large-scale distributed networks of servers, and it uses a push model (where the sender triggers the data transfer). FastReplica works in two steps: i) distribute equally sized parts of the original data among the destination servers, and ii) all destination servers send to all other destination servers their part (at the end of this step, all destination servers can merge all parts to obtain the original file).

Despite being very efficient, we point out two main issues: i) FastReplica relies on a push model, which can be very difficult to use when there is a variable set of destination nodes, and ii) it does not cope with node failures (since it was designed for servers, which are supposed to be up almost all the time).

BitTorrent is a peer-to-peer data distribution protocol widely used to distribute large amounts of data over the Internet. Its key idea is to use available upload bandwidth at peer nodes to distribute files that other peers want. This is specially useful to avoid the bottleneck of file servers (such as FTP servers). In order to find peers sharing the same file, a BitTorrent Tracker is used.

This protocol has proven to be capable of scaling up to millions of users and providing high efficiency in the distribution of large amounts of data [37]. Thus, we decided to integrate this protocol in our solution given that: i) it enables a pull model where data can flow as nodes need it (as opposed to FastReplica), and ii) it is more tolerant to faults regarding point-to-point protocols (if one client fails, other clients can still download the file from other clients), and iii) it scales up to millions of users.

8 Conclusions

freeCycles is a new, Internet wide, MapReduce-enabled computing platform. It presents a new data distribution technique for input, intermediate and final output data using the BitTorrent protocol. freeCycles is able to harness node's upload bandwidth to help distributing data. Moreover, it presents an improved map task and data replication scheduler, and is able to efficiently run iterative MapReduce applications.

From our experiments, we conclude that freeCycles is able to perform much better (performance and network scalability wise) than current platforms, namely BOINC and SCOLARS. Hence, with our work, it is possible to improve MapReduce applications' execution time on large computation pools such as the Internet. The current version of our prototype (freeCycles) is is publicly available.[8]

We describe (and built) freeCycles in the most generic way, i.e., making as few assumptions as possible regarding computing nodes. However, we believe this project could be of great use in two specific scenarios (or a combination of both). First, freeCycles can be used as a VC platform, harnessing volunteer node's resources in order to process MapReduce tasks. This is the most simple and generic scenario. Second, freeCycles can be used to deploy MapReduce jobs over multi-cloud environments. In this scenario, the user can utilize VMs (Virtual Machines) from different cloud providers and volunteer nodes at the same time. Using multi-cloud environments, instead of only one, can be of great use for several reasons: i) minimize the cost of all VMs (resource selection is out of the scope of this work, several existing solutions [38, 39, 41, 44, 49, 51] can be used to take into account several parameters such as network quality, CPU power and hourly cost), ii) fault tolerance (tolerate cloud provider faults), and iii) securing computation privacy by distributing it over multiple independent clouds.

Regarding future work, we plan to support the scheduling ability to decide which tasks to give each node according to its capacity (amount of CPU, bandwidth, storage, RAM, etc.), and its reliability. For reliability we assume that volunteer nodes are less reliable (nodes can go down at any time) while VMs from cloud providers are more reliable. Taking into consideration the capacity of each node we hope to be able to improve task distribution and therefore reduce the amount of stragglers (caused by bad scheduling decisions). Reliability will be used to tune the replication factor of each task. For example, tasks given to more reliable nodes can have a reduced replication factor.

[8]The source code can be found at https://github.com/rodrigo-bruno/freeCycles.

References

1. Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar, T.N.: Tarazu: Optimizing mapreduce on heterogeneous clusters. SIGARCH Comput. Archit. News **40**(1), 61–74 (2012)
2. Alexandrov, A.D., Ibel, M., Schauser, K.E., Scheiman, C.J.: Superweb: towards a global web-based parallel computing infrastructure. In: Parallel Processing Symposium, 1997. Proceedings., 11th International, pp. 100–106 (1997)
3. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: 2004. Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10 (2004)
4. Anderson, D.P., Christensen, C., Allen, B.: Designing a runtime system for volunteer computing. In: SC 2006 Conference, Proceedings of the ACM/IEEE, pp. 33–33 (2006)
5. Anderson, D.P., Fedak, G.: The computational and storage potential of volunteer computing. In: 2006. CCGRID 06. Sixth IEEE International Symposium on Cluster Computing and the Grid, vol. 1, pp. 73–80 (2006)
6. Baratloo, A., Karaul, M., Kedem, Z.M., Wijckoff, P.: Charlotte: Metacomputing on the web. Futur. Gener. Comput. Syst. **15**(5–6), 559–570 (1999)
7. Bazinet, A.L., Cummings, M.P.: Subdividing long-running, variable-length analyses into short, fixed-length boinc workunits. J. Grid Comput. **14**(3), 429–441 (2016)
8. Bertis, V., Bolze, R., Desprez, F., Reed, K.: From dedicated grid to volunteer grid: Large scale execution of a bioinformatics application. J. Grid Comput. **7**(4), 463 (2009)
9. Binzenhöfer, A., Leibnitz, K.: Estimating churn in structured p2p networks. In: Managing Traffic Performance in Converged Networks, pp. 630–641. Springer, Berlin (2007)
10. Borthakur, D.: The hadoop distributed file system: Architecture and design. Hadoop Proj. Website **11**, 21 (2007)
11. Bruno, R., Ferreira, P.: Scadamar: Scalable and data-efficient internet mapreduce. In: Proceedings of the 2Nd International Workshop on CrossCloud Systems, CCB'14, pp. 2:1–2:6. ACM, New York (2014)
12. Cardosa, M., Wang, C., Nangia, A., Chandra, A., Weissman, J.: Exploring mapreduce efficiency with highly-distributed data, In Proceedings of the Second International Workshop on MapReduce and its Applications, 27–34, ACM, New York (2011)
13. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI, vol. 99, pp. 173–186 (1999)
14. Chakravarti, A.J., Baumgartner, G., Lauria, M.: The organic grid: self-organizing computation on a peer-to-peer network. IEEE Trans. Syst. Man Cybern. Part A: Syst. Humans **35**(3), 373–384 (2005)
15. Cherkasova, L., Lee, J.: Fastreplica: Efficient large file distribution within content delivery networks. In: USENIX Symposium on Internet Technologies and Systems, Seattle (2003)

16. Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I.: Managing data transfers in computer clusters with orchestra. ACM SIGCOMM Comput. Commun. Rev. **41**(4), 98–109 (2011)
17. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: Planetlab: an overlay testbed for broad-coverage services. ACM SIGCOMM Comput. Commun. Rev. **33**(3), 3–12 (2003)
18. Costa, F., Veiga, L., Ferreira, P.: Internet-scale support for map-reduce processing. J. Internet Serv. Appl. **4**(1), 1–17 (2013)
19. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
20. Dinu, F., Ng, T.S.: Understanding the effects and implications of compute node related failures in hadoop. In: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, pp. 187–198. ACM, New York (2012)
21. Fedak, G., Germain, C., Neri, V., Cappello, F.: Xtremweb: a generic global computing system. In: 2001. Proceedings. First IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 582–587 (2001)
22. Fedak, G., He, H., Cappello, F.: Bitdew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. J. Netw. Comput. Appl. **32**(5), 961–975 (2009). Next Generation Content Networks
23. Gentzsch, W., Girou, D., Kennedy, A., Lederer, H., Reetz, J., Riedel, M., Schott, A., Vanni, A., Vazquez, M., Wolfrat, J.: Deisa—distributed european infrastructure for supercomputing applications. J. Grid Comput. **9**(2), 259–277 (2011)
24. Georgatos, F., Gkamas, V., Ilias, A., Kouretis, G., Varvarigos, E.: A grid-enabled cpu scavenging architecture and a case study of its use in the greek school network. J. Grid Comput. **8**(1), 61–75 (2010)
25. Heckmann, O., Bock, A.: The edonkey 2000 protocol. Rapport technique, Multimedia Communications Lab, Darmstadt University of Technology, 13 (2002)
26. Heien, E.M., Anderson, D.P., Hagihara, K.: Computing low latency batches with unreliable workers in volunteer computing environments. J. Grid Comput. **7**(4), 501 (2009)
27. Kailasam, S., Dhawalia, P., Balaji, S.J., Iyer, G., Dharanipragada, J.: Extending mapreduce across clouds with bstream. IEEE Trans. Cloud Comput. **2**(3), 362–376 (2014)
28. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: Making cloud intermediate data fault-tolerant. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 181–192. ACM, Berlin (2010)
29. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., et al.: Oceanstore: An architecture for global-scale persistent storage. ACM Sigplan Not. **35**(11), 190–201 (2000)
30. Langville, A.N., Meyer, C.D.: Google's PageRank and beyond: the science of search engine rankings. Princeton University Press, Princeton (2011)
31. Li, P., Guo, S., Yu, S., Zhuang, W.: Cross-cloud mapreduce for big data. IEEE Trans. Cloud Comput. **PP**(99), 1–1 (2015)
32. Liang, J., Kumar, R., Ross, K.W.: The fasttrack overlay: A measurement study. Comput. Netw. **50**(6), 842–858 (2006)
33. Lin, H., Ma, X., Archuleta, J., Feng, W.-c., Gardner, M., Zhang, Z.: Moon: Mapreduce on opportunistic environments. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pp. 95–106. ACM, New York (2010)
34. Lo, V., Zappala, D., Zhou, D., Liu, Y., Zhao, S.: Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In: Peer-to-Peer Systems III, pp. 227–236. Springer, Berlin (2005)
35. Marozzo, F., Talia, D., Trunfio, P.: Adapting mapreduce for dynamic environments using a peer-to-peer model. In: Proceedings of the 1st Workshop on Cloud Computing and its Applications (2008)
36. Nguyen, T., Shi, W.: Improving resource efficiency in data centers using reputation-based resource selection. In: Green Computing Conference, 2010 International, pp. 389–396, USA (2010)
37. Pouwelse, J., Garbacki, P., Epema, D., Sips, H.: The bittorrent p2p file-sharing system: Measurements and analysis. In: Peer-to-Peer Systems IV, pp. 205–216. Springer, Berlink (2005)
38. Qureshi, M.B., Dehnavi, M.M., Min-Allah, N., Qureshi, M.S., Hussain, H., Rentifis, I., Tziritas, N., Loukopoulos, T., Khan, S.amee.U., Xu, C.-Z., Zomaya, A.Y.: Survey on grid resource allocation mechanisms. J. Grid Comput. **12**(2), 399–441 (2014)
39. Rasooli, A., Down, D.G.: Guidelines for selecting hadoop schedulers based on system heterogeneity. J. Grid Comput. **12**(3), 499–519 (2014)
40. Ripeanu, M.: Peer-to-peer architecture case study: Gnutella network. In: 2001. Proceedings. First International Conference on Peer-to-Peer Computing, pp. 99–100. IEEE, USA (2001)
41. Rood, B., Lewis, M.J.: Grid resource availability prediction-based scheduling and task replication. J. Grid Comput. **7**(4), 479 (2009)
42. Sarmenta, L.F.G., Hirano, S.: Bayanihan: building and studying web-based volunteer computing systems using java. Futur. Gener. Comput. Syst. **15**(5–6), 675–686 (1999)
43. Silberstein, M., Sharov, A., Geiger, D., Schuster, A.: Gridbot: execution of bags of tasks in multiple grids. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09, pp. 11:1–11:12. ACM, New York (2009)
44. Singh, S., Chana, I.: A survey on resource scheduling in cloud computing Issues and challenges. J. Grid Comput. **14**(2), 217–264 (2016)
45. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks, In Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement, 189–202, ACM, New York (2006)
46. Tang, B., Moca, M., Chevalier, S., He, H., Fedak, G.: Towards mapreduce for desktop grid computing. In: 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 193–200 (2010)
47. Tang, B., Tang, M., Fedak, G., He, H.: Availability/ network-aware mapreduce over the internet. Inf. Sci. **379**, 94–111 (2017)

48. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurr. Comput. Pract. Exper. **17**(2-4), 323–356 (2005)
49. Toth, D., Finkel, D.: Improving the productivity of volunteer computing by using the most effective task retrieval policies. J. Grid Comput. **7**(4), 519 (2009)
50. White, T.: O'Reilly (2012)
51. Yang, S., Butt, A.R., Fang, X., Hu, Y.C., Midkiff, S.P.: A fair, secure and trustworthy peer-to-peer based cycle-sharing system. J. Grid Comput. **4**(3), 265–286 (2006)
52. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 10–10 (2010)