Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

Internet-scale support for map-reduce processing

Fernando Costa*, Luís Veiga and Paulo Ferreira*

Abstract

Volunteer Computing systems (VC) harness computing resources of machines from around the world to perform distributed independent tasks. Existing infrastructures follow a master/worker model, with a centralized architecture. This limits the scalability of the solution due to its dependence on the server. Our goal is to create a fault-tolerant VC platform that supports complex applications, by using a distributed model which improves performance and reduces the burden on the server.
In this paper we present VMR, a VC system able to run MapReduce applications on top of volunteer resources, spread throughout the Internet. VMR leverages users' bandwidth through the use of inter-client communication, and uses a lightweight task validation mechanism. We describe VMR's architecture and evaluate its performance by executing several MapReduce applications on a wide area testbed.
Our results show that VMR successfully runs MapReduce tasks over the Internet. When compared to an unmodified VC system, VMR obtains a performance increase of over 60% in application turnaround time, while reducing server bandwidth use by two orders of magnitude and showing no discernible overhead.

1 Introduction

The use of volunteer PCs across the Internet to execute distributed applications has been increasing in popularity since its inception in the early 1990s, with the creation of projects such as Distributed.net [a], Seti@home [1] or Folding@home [2]. These Volunteer Computing (VC) systems harness computing resources from machines running commodity hardware and software, and perform highly parallel computations that do not require any interaction between network participants (also called bag-of-tasks).

Existing VC systems support over 60 scientific projects[b], and have over a million participants, rivaling supercomputers in computing power. The most popular middleware, BOINC [3], is currently being used by over 40 projects, from scientific fields ranging from climate prediction to protein folding. The amount of computational power available for large scale computing over the Internet can only keep increasing. On one hand, the number of Internet connected devices is expected to increase exponentially with the advent of mobile devices [4]. On the other hand, Moore's law continuous relevance shows that we can expect a sustained evolution of the hardware in the last mile of the Internet. This translates to an incredible amount of untapped computing and storage potential in machines spread throughout the world.

However, current VC systems have a centralized architecture that follows a master/worker model, as a small number of servers is responsible for task distribution and result validation. This limitation has prevented Volunteer Computing from reaching its true potential. In addition, the single point of failure inevitably creates a bottleneck, as projects expand and storage and network requirements become more demanding.

A Volunteer Computing system is sometimes described as a Desktop Grid (DG). However, in order to differentiate between both concepts we define DG as a Grid Computing cluster that uses idle desktop machines (PCs). A DG provides increased accountability, and typically offers better connectivity and availability than a typical VC environment, which spreads over the Internet. Desktop Grids may include enterprise environments, schools or scientific laboratories. This means that some Desktop Grids support more complex applications, such as MPI [5,6]. While a VC platform may be deployed on top of a desktop cluster, this is not the target environment intended for existing

*Correspondence: fcosta@gsd.inesc-id.pt; paulo.ferreira@inesc-id.pt
Distributed Systems Group, INESC-ID, Universidade de Lisboa, R. Alves Redol, 9, 1000-029 Lisboa, Portugal

systems. Therefore, throughout this document, whenever VC is mentioned we will be referring to large scale systems deployed over the Internet.

1.1 Goal and challenges

Our goal is to build a scalable, and fault-tolerant Volunteer Computing (VC) platform, which improves the performance of VC systems when running complex applications (e.g., MapReduce). This platform, called VMR (Volunteer MapReduce), must have minimal dependency on any central service, by decentralizing some of the mechanisms of existing systems that place an excessive burden on the central server.

As parallel and distributed computing becomes the answer for increased scalability for varied computational problems, several paradigms and solutions have been created during the last decade. Regarding the support for complex applications, among the potential candidates, we have chosen MapReduce [7] as the novel paradigm to support in VC for the following reasons: it is recent and widely adopted (e.g., Amazon's EC2[c]); it is currently limited to clusters (Cloud Computing); many applications can be broken down into sequences of MapReduce jobs; it is a good example of data-intensive computing, requiring task coordination, and is heavily linked to distributed storage. MapReduce may also work as a stepping stone for other paradigms, such as scientific workflows, that could be adapted to a volunteer environment.

MapReduce leverages the concept of Map and Reduce commonly used in functional languages: a map task runs through each element of a list and produces a new list; reduce applies a new function to a list, reducing it to a single final value or output. In MapReduce, the user specifies a map function that processes tuples of key/values given as input, and generates a new intermediate list of key/value pairs. This map output is then used as input by a reduce function, also predefined by the user, that merges all intermediate values that belong to the same key. Therefore, all reduce inputs are outputs from the previous map task. Throughout the rest of the paper, we will be referring to them as map outputs. The adaptation of MapReduce to a volunteer environment is an interesting challenge because its current implementations are limited to cluster environments.

There are several challenges and requirements to consider, in order to achieve our objective.

First and foremost, our solution must be able to take advantage of the huge amount of VC resources that we previously mentioned. We must consider both the hardware capabilities of individual machines and the network bandwidth that is at our disposal, at the last mile of the Internet.

Our system must also be compatible with existing VC solutions (e.g. BOINC [3]), since developing a whole new platform from scratch would be of no practical use once the research was finished. Therefore, we must take into account existing systems and use their infrastructure to come up with a final prototype that can actually be used in the near future, in a real-world scenario. In fact, our solution would undoubtedly bring significant disadvantages if it required that only our system's clients were attached to a project.[d] To avoid this situation we must guarantee compatibility with existing projects. Any client must be able to run any project application. On the other hand, our solution must support existing applications, and successfully schedule tasks on existing clients.

Additionally, the execution of our system on unreliable, non-dedicated resources requires fault tolerance mechanisms. This means it must account for unreachable clients, which have disconnected from the server, or are simply offline.

Another potential problem is caused by byzantine behavior [8]. Clients may maliciously return incorrect results, or inadvertently produce an incorrect output by encountering errors during the computation or data transfers.

Finally, our solution must be able to withstand transient server failures. This is particularly important in our case because we will be dealing with long running applications, with a potentially high level of server interactions. We need to prevent the execution on the clients to come to a halt, as they wait for the server to come back up.

1.2 Drawbacks of current solutions

Existing solutions do not fulfill the goal we described, and are inadequately prepared to meet the requirements mentioned above.

Most Volunteer Computing systems have a centralized architecture, with communication going through a single server. There are few exceptions and they were created with a smaller scope or environment in mind [9]. In BOINC [3], XtremWeb [10] and Folding@home [2], the server or coordinator must fulfill the role of job scheduler, by handling task distribution and result validation.

Current VC systems (not DG) are limited to bag-of-task applications since this architecture creates too much overhead on the server, when considering more complex data distribution or storage. Existing projects such as Climateprediction.net and MilkyWay@home have encountered problems when dealing with large files or having the same data shared by many clients [11]. Although some potential solutions have been proposed [12,13], they have not been deployed in the most widely used systems.

Fault tolerance is strictly confined to the client-side in current VC systems. Although some projects do have a set of mirrors that act as data repositories, all client requests and task scheduling goes through the central server. Therefore, any server fault that prevents it from communicating with clients has a very high probability of disrupting clients and stopping further task execution.

Finally, a considerable limitation of existing VC systems is their focus on bag-of-tasks applications, with little or no communication and without dependencies between the tasks. None of the current platforms support MapReduce, a widely used programming model that adapts well to a data-intensive class of applications. Supporting MapReduce requires fundamental changes on existing algorithms, and the introduction of on-the-fly task creation. This is currently not available on any present system.

In this paper we present VMR, a VC system that is able to execute MapReduce tasks over the large scale Internet, on top of volunteer resources. Our system is compatible with existing solutions (in particular BOINC). VMR is able to decentralize the existing architecture, by using client to client transfers, and minimizing the volume of data sent through the server. This also allows VMR to tolerate transient server failures, as the clients depend merely on other peers for data. It is also capable of tolerating VC clients' failure by using replication (i.e. running the same task on several VC machines). By increasing the replication factor, the probability of a failure of all clients running a certain task is lowered. Finally, byzantine behavior is controlled through the use of task validation in the server. By replicating each task at least twice, it is possible to compare the outcome and accept only the results in which a quorum has been reached.

This paper is organized as follows: VMR is presented in more detail in Section 2; Section 3 describes the most relevant implementation aspects, and presents experimental results, conducted with several different MapReduce applications, on a large scale testbed [14]; related work is discussed in Section 4; and Section 5 concludes.

2 VMR

VMR's architecture consists of a central server, and clients which can assume two different roles: mappers, which are responsible for bag-of-tasks in the map stage; and reducers, which perform the aggregation of all map output in the reduce step. VMR is compatible with BOINC (Berkeley Open Infrastructure for Network Computing), the most successful and popular volunteer computing middleware to date. Consequently, it is able to borrow its mechanisms and algorithms to deal with many of the challenges of Volunteer Computing systems.

However, BOINC suffers from the fundamental drawback of overloading the server because it follows a master/worker model, in which a central entity is responsible for scheduling and validating tasks. Although it is possible to use mirrors to hold data, many projects use a single machine for both data storage and scheduling. Furthermore, these mirrors act as web servers, as all data is transferred through HTTP, making this impractical to implement on VC clients. Therefore, BOINC projects do not fully exploit users' increasing bandwidth, and deploy compute intensive applications. In data-intensive scenarios, BOINC is unable to quickly propagate input files that are shared by many clients, for example [12,13].

The parameters of the MapReduce job to run on top of VMR are defined by the user, and stored in the server. This includes the number of map and reduce tasks, the executable files used by map and reduce tasks, as well as their hardware and software requirements. Once all the MapReduce job characteristics have been defined, the VMR server creates the map tasks, and stores this information in its database — the VMR database is responsible for holding all persistent information on tasks, clients, and applications being run.

The overall VMR execution model is presented in Figure 1. A group of mappers first requests work from the VMR server's scheduler (1). The server follows a simple scheduling procedure when selecting which available task is assigned to each mapper or reducer; whenever it receives a work request, it matches each task's predefined hardware or software requirements to the client's machine characteristics. These requirements may include memory, disk space, CPU or Operating System specifications. If the client is suitable, the server assigns it a task and saves this information in its database. By taking advantage of the underlying middleware (i.e., BOINC), VMR is able to provide the same scheduling options. Therefore, it is possible to select scheduling techniques that are appropriate for the application. After selecting an appropriate map task for the requesting mapper, the scheduler sends back information on the task that the mapper must execute. This information includes the location of input and executable files, the deadline for task completion and the previously mentioned task requirements. The machines holding input and executable files are called data servers. Most VC projects store the data in a central server, represented in Figure 1 as VMR server, which also holds the remaining VC components (e.g., scheduler and database).

The mapper must then download the required data from the data server (2) before starting the computation (3). After the task execution is completed, the mapper creates an MD5 hash for each of the map output files. Therefore, at the end of the computation, each mapper is left with both the map output files and the same number of corresponding hashes. These hash sums are sent back to the server in place of the output files (4) (so it is compatible with current VC solutions, e.g. BOINC). This greatly reduces the upload volume from mappers to the VMR server (as discussed in Section 3).

The hashes are compared at the server in order to validate each corresponding task (5). If the result is valid, the

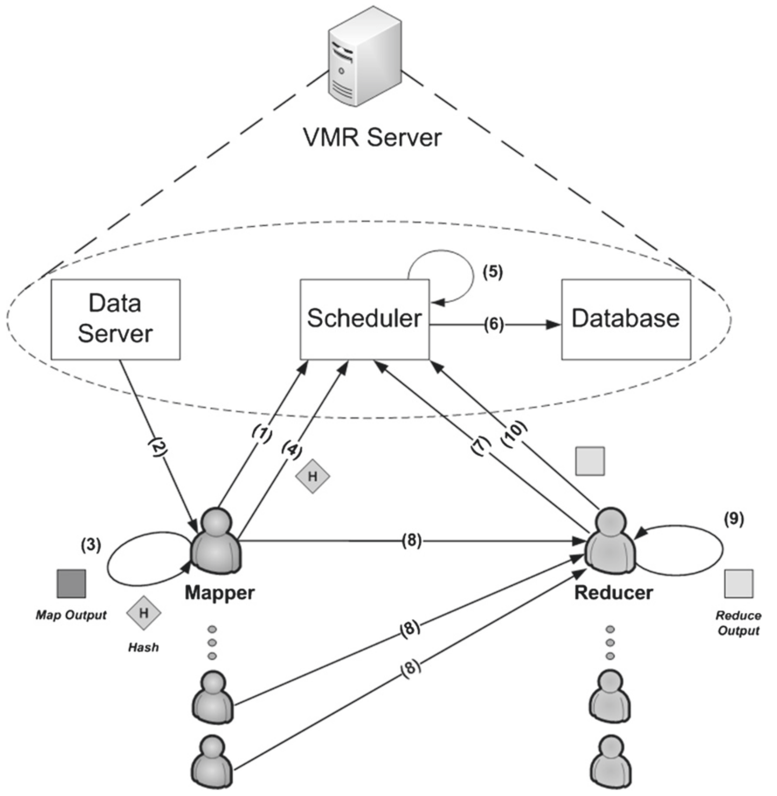

Figure 1 VMR Execution Model.

mapper's address is stored in VMR's database (6). Each time a map result is validated, the VMR server checks if all map tasks have been executed and validated. When this condition is met, the server creates the predefined number of reduce tasks. Existing MapReduce implementations typically allow for the reduce step to start as soon as a fraction of the map tasks are completed. The reducers can start downloading the required input files earlier, thus improving turnaround time. VMR is currently unable to provide this option, but it is considered as future work. We have postponed this improvement due to its complexity in terms of file transfer and data management. A reducer may then send a work request to the scheduler (7), in order to be assigned a reduce task. The VMR server follows the task scheduling procedure defined earlier and looks through the database to find a task that can be assigned to the reducer. If the reducer meets all the hardware and availability requirements, the scheduler replies with a reduce task that fits the request.

MapReduce jobs require communication between map and reduce stages since map outputs are used as input for reduce tasks. In the reduce step, each task performs join operations on the map outputs. Therefore, each reduce task must obtain all the map outputs that correspond to the key range it is responsible for. In order to achieve good performance in MapReduce jobs, we leverage clients' resources by moving as much of the communication as possible to the client-side. This helps reduce the load on the central server, and creates a more suitable decentralized model for data-intensive scenarios, typical of MapReduce.

Note that, as previously stated, in current VC systems all data would have to be uploaded and downloaded from a central server. However, the VMR server stores the addresses of all mappers that returned valid map results. This information is included in the work request reply, and allows reducers to download the map output directly from the mappers, without having to go through the VMR server (8). Once the input files have been downloaded, the reduce task is executed (9) and the final result is returned to the server (10) for validation.

2.1 Byzantine behavior and fault tolerance

As we mentioned previously, dealing with distributed applications over unreliable resources requires mechanisms for byzantine tolerance. The results of a task cannot be blindly trusted as some node may have hardware problems, causing errors in the computation. Additionally, volunteers may intentionally return wrong results or sabotage the execution.

In VMR, byzantine behavior is handled through replication, and using majority voting. Each map and reduce task has at least 2 replicas, running on different clients. Upon receiving sufficient results (i.e., at least 2) for the same task, the server is able to consider it valid if a strict majority of clients return the same output. Unlike current VC systems, VMR uses hashes for task validation, which greatly reduces communication with the central server.

Task replication also helps guarantee fault tolerance against volunteers becoming unreachable or going offline. However, it does not help in case of a server failure. In order to achieve our goal of increasing the scalability of Volunteer Computing systems, VMR offers increased robustness, by being able to withstand transient server failures.

In our system, the VMR server is responsible for assigning tasks to clients for execution. Any fault or error that makes the server unreachable prevents new tasks from being scheduled. However, tasks that have already been scheduled can proceed with their execution if their required input files have been previously downloaded or if they are not stored in the server.

In existing VC systems, if the server becomes unavailable, the reducers are no longer able to obtain the map output files required for their task, preventing its execution. In VMR, however, all the reduce tasks that have been scheduled can proceed as normal, since map outputs are hosted at mappers (whose address is already known to reducers). The reducer's next communication with the server will only be used to report the task completion.

The scenario we have just described has a high probability of occurring because task scheduling is done in a short interval, immediately after the map stage is complete. In addition, the transfer of map outputs and corresponding execution of the reduce tasks takes up most of the MapReduce application turnaround time. Therefore, although VMR does not allow task scheduling if the server is offline, it is still able to keep MapReduce jobs executing if the failure occurs after the map stage. This is achieved due to VMR's use of inter-client transfers. In the case of a server failure during the reduce step, VMR reducers are unaffected.

2.2 Client to client transfer

As we stated previously, each mapper makes its map outputs available for download as soon as the task is finished. The map output files are available for download until a timeout is reached (set at several times the expected application runtime) or the MapReduce job is completed. A VMR mapper only accepts requests for the existing map output files, and discards messages that do not follow a predefined template.

Once the map task is completed and reported back to the server, each mapper's address is stored in the VMR server's database. After receiving a work request from a reducer, the VMR server includes the addresses of mappers that hold each of the map output files on the reply. Since map tasks are replicated, several mappers hold each map output file. Therefore, all reduce tasks sent to VMR reducers have the location of the required input data, as a list of IP addresses of mappers.

The list of mapper addresses is randomly ordered by the server. Upon receiving the reduce task information, the reducer goes through each mapper in the list returned by the VMR server in order. By having the mapper list randomly ordered, we implement a simple load balancing mechanism. This lowers the chance of having several reducers trying to download from the same mapper, and overloading it.

3 Implementation and evaluation

We evaluate VMR by running several tests over the Internet, in a scenario that resembles a typical VC environment. We run experiments with 4 different applications (word count, inverted index, N-Gram, and the NAS EP benchmark), in order to gauge our system's performance under different conditions. Apart from the NAS EP benchmark, used by NASA, the remaining applications are packaged by many MapReduce implementations as benchmarks. All of them have the characteristics expected of data-intensive jobs, and are described further ahead. This section presents the results of our experiments, describes the applications we use and reveals some implementation aspects.

It is part of our goal to improve the performance of VMR when running MapReduce applications. Therefore, we compare VMR with an existing VC system (BOINC). We use the VMR server in all our experiments, since current VC systems are unable to support MapReduce applications. As we previously mentioned, the VMR server is compatible with clients from existing VC solutions. Therefore, we are able to use VMR and unmodified BOINC clients in our experiments.

Throughout this section, we refer to the existing Volunteer Computing system we use for comparison as *VCS*. We run tests on two versions of our system: *VMR* corresponds to our system, as previously described; whereas *VMR-NH* is a VMR version that does not use hashes for map outputs. The VMR-NH client returns the map output files to the server, exactly as the VCS clients. However, both VMR-NH and VMR use inter-client transfers, while all communication goes through the server in VCS. VMR-NH allows us to assess the impact of using hashes (VMR), when compared to returning the output file back to the server but still using inter-client transfers (VMR-NH).

We measure application turnaround, while differentiating between map and reduce stages in order to pinpoint potential bottlenecks and areas that would benefit most from improvement. Additionally, we monitor network traffic on the server. This allows us to identify the benefits of reducing the dependence on the central server. Finally, we measure the overhead created by our system in terms of memory and CPU to compare the burden placed on the server.

We run our experiments on PlanetLab, a wide-area testbed that supports the development of distributed systems and networks services. We use 50 to 200 PlanetLab nodes that work as the clients. We initially deployed our server in a PlanetLab node, which imposed significant limitations (described ahead in 3.2.5) to our experiments. We then moved the server to a public machine of our local cluster, to run tests on a more controlled setting.

3.1 Implementation aspects

This section presents the most relevant implementation aspects of VMR. Instead of starting from scratch, and potentially creating a whole new platform that would be of no practical use once the research was finished, we extend BOINC, because of its popularity and for being open source. Therefore, we take advantage of its existing features and organization. In the server, a MySQL database is used, while backend components (e.g., file deleter) are daemons written in C or C++, or implemented in Python. VMR is designed on top of a BOINC client version 6.11.1, and server version 6.11.0. For the existing VC clients, we use BOINC's 6.13.0 version.

To support the execution of MapReduce jobs by BOINC clients, we modified the server. The VMR server, besides storing MapReduce job metadata, supports dynamic task creation: reduce work units are inserted into the database whenever all the map tasks have been finished. Furthermore, we have also changed the server to treat all map output data as input for the reduce work units. Therefore, when the reducers receive the reply to their work request, the locations of the input files are already updated and point to the server.

In our work, we try to stay faithful to existing communication and development procedures. Thus, we use (or modify) XML in messages between clients and the server. All data transfers between server and client are done through HTTP, while inter-client transfers are implemented through the use of a TCP connection. We have considered using UDP instead, but have kept TCP due to its inherent guarantees (e.g., message delivery).

In MapReduce, each mapper, with output available for reducers to download, stops accepting connections if one of the following situations occur: the client is shut down; the MapReduce job has completed successfully; or the mapper has reached a timeout in total hosting time.

VMR gives application developers the option to use hashes for validation or to use the traditional validation mechanism, which requires the transfer of output files from the clients. In the latter case, increasing the burden on the server can provide increased data availability. As an example, in MapReduce, reducers are able to download map output files from the server. This is used as a fall-back mechanism for failed inter-client transfers: after n failed attempts to download an output file directly from mappers, the reducer resorts to downloading all missing files from the server. In scenarios with low client availability or bandwidth, this prevents the execution from coming to a halt. However, our clients (i.e., reducers) always attempt to download from another client (i.e., mapper) before resorting to the server.

In that situation (unavailable mappers), the system reverts back to the original BOINC execution mode, in which all files are sent back to the server. In the worst-case scenario, in which all nodes are unavailable for uploading output files, the VMR server will simply work on the assumption that there are only original BOINC clients. Thus, even with client failures, the results would not be worse than the VCS scenario. Furthermore, the system can always resort to increasing the mapper replication until the required reliability is achieved.

3.2 Experimental setup

In order to coordinate the concurrent execution of clients in PlanetLab, we take advantage of Nebula and Plush[e], two PlanetLab tools that allow us to send commands to several nodes simultaneously. In order to evaluate our system, we create a VC project (following BOINC terminology) to run all the MapReduce applications. We describe each of them in turn.

3.2.1 Word count

The word count application is a widely accepted benchmark in MapReduce implementations. Each map task receives a file chunk as input, counts the number of words in it and outputs an intermediate file with "word n" pairs for each word found. The value n corresponds to the number of occurrences for each corresponding word. Typically, map tasks store the output immediately on the file system, so instead of having one line with "word 5", for example, the output file usually holds 5 lines with "word 1". This can be improved through the use of a local reducer, which aggregates identical results in memory before storing them in the file system. However, we decided to use the non-optimized version, and have our mappers always write "word 1" whenever a word is found. The reduce step collects all the map intermediate outputs and aggregates them into one final output.

3.2.2 Inverted index

This is another typical benchmark of MapReduce systems, in which the final output lists all the documents each word belongs to. The map task parses each chunk, and emits a sequence of "word document-ID" pairs. This means that for each word found, the map task identifies the document it belonged to through its ID. The reduce task merges all pairs for a given word, and emits a final "word list(document ID)" pair.

3.2.3 N-Gram

An N-Gram is a contiguous sequence of N items from a given input. The output from N-Gram applications can be used in various research areas, such as statistical machine translation or spell checking. In our case, it is useful to extract text patterns from large size text and give statistical information on patterns' frequency and length.

Each map task receives its corresponding file chunk as input, and counts all sequences of words of length 1 to N. In our case, we defined N as 2, for reasons explained in the next subsection. As in the previous applications, a map output produces a "sequence n" pair for each sequence with 1 or 2 words. Just as in the word count case, and following the typical execution model of MapReduce, n is always 1. Therefore, the map task produces an output file that dedicates a line for each sequence of words found in the text input file. The reduce step collects all the map intermediate outputs and aggregates all coinciding sequences into one final count, thus producing a pattern frequency result.

The map task from the two previous applications produces output files that are a little larger than the initial input. Therefore, the amount of data produced by mappers (and then sent to reducers) is similar to the volume received by mappers as input. On the other hand, for each 5 MB input, N-Gram's map task creates around *30 MB* of intermediate files which must be transferred to reducers. Therefore, N-Gram is helpful in assessing the performance of our system with applications with large intermediate files.

3.2.4 NAS EP Benchmark

The NASA Advanced Supercomputing (NAS) Division developed the NAS Parallel Benchmarks[f], which is a set of programs used to evaluate the performance of parallel supercomputers. This test suite includes 8 benchmarks, which have several implementations, and serve different purposes in evaluating system performance.

In our experiments, we use the NAS Embarrassingly Parallel Benchmark (referred to as NAS EP from here on), which generates complex pairs of uniform (0, 1) random numbers, using the Marsaglia polar method [15]. This application's main advantage is its simple adaptation to a MapReduce job: the map task works as a random number generator, while the reduce step gathers statistics on the results obtained.

Each map task receives a number range, and the seed for the random number generator as input, and outputs pairs of uniform random numbers, between 0 and 1. Each reduce task is responsible for gathering information on the numbers obtained for a set of seeds. This application provides a different challenge, when compared to the previous examples, since it has very small input and output files (unlike N-Gram), but creates a large amount of intermediate data.

We use an initial input text file of 1 GB, divided into 100 chunks (one 10 MB chunk per map task), in the word count and inverted index experiments. For the N-Gram application, we also split the input into 100 chunks, but each chunk is only 5 MB in size, due to the larger size of intermediate files. The NAS EP benchmark receives as input a small file with a number range and an initial seed (for a random number generator).

3.2.5 Limitations of PlanetLab

Each node in PlanetLab may be shared by multiple virtual machines (slivers) at any time. For obvious reasons, users do not have access to slivers they do not own, and cannot predict when they will be executed. As such, our experiments were occasionally influenced by other slivers running at the same time. This was especially notorious in the node acting as the server, as the network bandwidth could reduce suddenly and drastically. We were able to identify these incorrect experiments due to their unusually long execution time, and through the use of HTTP commands to occasionally download files from the server.

PlanetLab has another significant limitation: disk space. Each node running our virtual machine has access to 8 GB in disk, which is very limited for our purpose. This was especially true when running MapReduce jobs on unmodified VC clients (defined in the next section as VCS), since the server has to hold the initial map input, map output files and the final reduce output.

Therefore, we decided to move the server to a local, publicly accessible machine in our lab to overcome these constraints. All experiments involving more than 50 clients were performed with this scenario. This also allows us to correctly calculate the overhead of VMR.

3.3 Application turnaround

We begin by measuring application turnaround on all experiments. We measure the time it took each MapReduce job to finish, starting from the initial download of map input files from the VMR server, and ending with the upload of the last reduce output back to the server. We separate the map and reduce steps in order to identify their respective weight in regards to the overall application turnaround time. The map stage is considered to be finished once all its output has been validated in the VMR server. In our initial experiments, for the word count, inverted index and N-Gram applications, we use 50 clients, and the VMR server is deployed in PlanetLab. The experiments with 100 and 200 clients, for all applications, are discussed further ahead, still in this section.

We run the word count application with VMR, VMR-NH and VCS. The turnaround time of the word count application for the three alternatives is shown in Figure 2. For this application, the scenario with VMR clients has the lowest turnaround, followed by VMR-NH and then VCS. Both VMR and VMR-NH perform considerably better than VCS in the reduce step, taking only 40% of VCS's time. This speedup can be attributed to the inter-client transfers, which reduce the communication with the central server. On the map stage, VMR-NH and VCS have similar results, with VMR-NH performing marginally better. This was expected as both clients download map inputs and return its output files to the server. The use of hashes yields a considerable improvement on the map step, with VMR reducing its execution interval to 65% of VCS's value. When considering the full job turnaround, with map and reduce execution, VMR is able to cut the time required by existing VC systems in more than half.

The results obtained by VMR-NH in the remaining applications were very similar to the ones just described. VMR-NH is always slower than VMR in the map stage, but consistently faster than VCS in the reduce step. This means that VMR-NH always presents an application turnaround time below VCS, but larger than VMR. For that reason, for the remaining experiments we only show the results for VCS and VMR.

The results of the Inverted Index application support our previous findings, as we can see in Figure 3. VMR's map tasks finished almost twice as fast as VCS. The reduce step is also faster with VMR, with an overall speedup of 1.25.

In the N-Gram experiments the VMR server had to be deployed on a faster PlanetLab node in order to make sure the jobs finished. In fact, due to the large volume of map output data created, N-Gram creates a scenario in which the reduce task would take over 6 hours to complete, when deploying the server on the previous node. The extended length of the experiment increased the probability of other virtual machines using the node, making it slow to a crawl in some cases, or even preventing the tests from finishing. Note that this excessive running time length was observed with the unmodified VC clients. When running our VMR clients, we were able to complete the execution of N-Gram jobs even with the server running on the slower node. This further proves that our approach allows slower machines to be used as server, as the burden is greatly reduced by leveraging VMR clients.

The results obtained with 50 nodes for N-Gram are shown in Figure 4. The first conclusion we can gather from a first look at the graph is that VMR is able to finish the MapReduce job in half the time of VCS. This is consistent with previous results from the word count application. However, in this experiment we can also observe that the reduce stage on VMR is only slightly faster than VCS. This can be explained by the better network connection of the node used as server specifically for this application. Despite its larger bandwidth, inter-client transfers still perform better than the centralized system. On the other

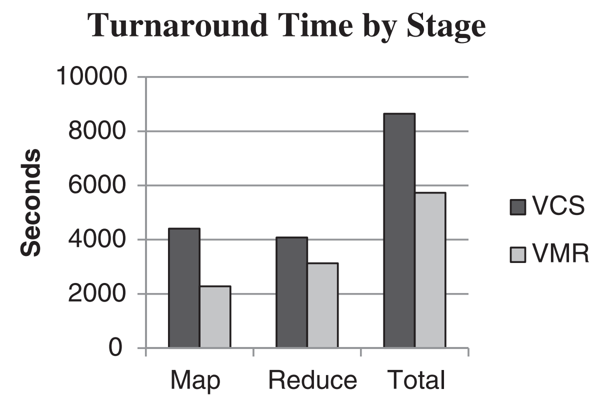

Figure 3 Turnaround Time of Inverted Index Application.

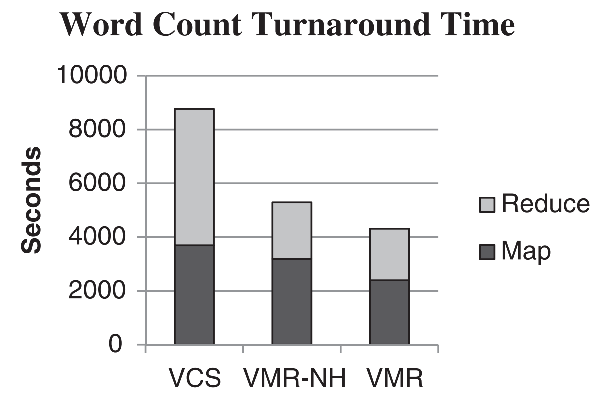

Figure 2 Turnaround Time of Word Count Application.

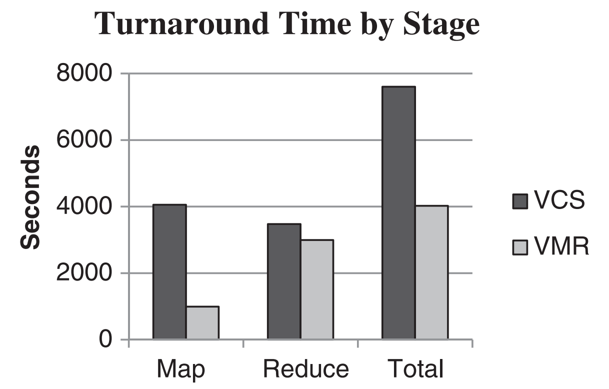

Figure 4 Turnaround Time of N-Gram Application.

hand, the differences in the map step are, as expected, much more significant. VMR is 4 times faster in executing the map stage, which translates to just a quarter of time needed by VCS to validate all its map tasks. This result shows us that VMR performs better with applications that create large intermediate files.

In the experiments presented so far (Figure 2 to Figure 4), we use a default replication factor of 2, for both the map and reduce tasks. This is the minimum required number of replicas, because a task is only considered valid if at least 2 replicas' results match. The use of inter-client transfers may suggest that a higher number of replicas increases data availability, and possibly improve the transfer speed. In order to test that hypothesis, we run further tests with VMR using the word count application, while varying the number of replicas of both map and reduce task.

When selecting the number of replicas to use, we took into consideration the values used in existing VC systems. In BOINC, for example, for each work unit (formal representation of a task, including its metadata), 3 replicated tasks are created and submitted to clients. This setup allows a VC platform to tolerate the failure of a single task. Furthermore, it reduces the impact of slower nodes since only the first two correct results returned are needed to validate a work unit. On the other hand, using more than 3 replicas creates unnecessary overhead, since many clients would be required to perform redundant computations. Thus, we decided to run tests with either 2 or 3 replicas for both the map and reduce tasks.

The results are shown in Figure 5. VMR is the baseline reference that uses 2 replicas for both map and reduce tasks. VMR-M3 creates 3 replicas for each map task while maintaining 2 reduce replicas. On the other hand, VMR-R3 replicates each reduce task three times, whereas each map task is only shared among 2 clients.

The results show that all three VMR versions have similar performance on the map stage. Curiously, VMR-M3, which uses 3 map replicas, has the worst results, which may suggest that the impact of slower nodes at this step was negligible. VMR and VMR-M3 present again close results in the reduce step. This means that, although there are more mappers to download from, there is not a visible gain from that larger pool. VMR-R3 has the worst results at this step, which can be explained by the additional data transfers between clients. Since each map output has to be uploaded to 3 reducers instead of 2, the mappers' upload bandwidth is the bottleneck. We conclude that having a larger pool of reducers does not translate to better performance, with a small number of nodes.

To investigate this issue further, we conduct experiments with both VCS and VMR clients, in larger scale settings, with 100 or 200 nodes. Throughout the remaining experiments on application turnaround, we use a simple notation to identify the number of replicas used in the reduce task: VMR-xRy. The x value corresponds to the number of unique reduce work units that have to be replicated, while y is the number of replicas created for each work unit. Therefore, VMR-15R2 corresponds to a VMR execution, with 15 reduce tasks, each with 2 replicas, while VCS-15R3 describes a VCS experiment, also with 15 reduce tasks, but with a replication factor of 3.

The results for the VMR version running the N-Gram application are shown in Figure 6. The VMR-15R2 version uses 100 nodes as clients, while 200 clients were deployed in all other executions. We only used 100 nodes in VMR-15R2 because after a certain threshold, there is no longer any difference by increasing the number of nodes, since there are only 15 reduce work units. This means that, with a replication factor of 2, there are only 30 reduce tasks being created (the remaining 70 nodes will be idle). Having 200 or 300 nodes does not improve performance since the map outputs are already spread throughout clients

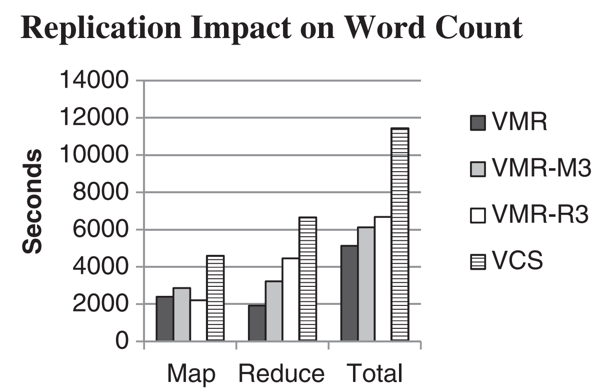

Figure 5 **Word count application execution, with 50 nodes and 2 or 3 task replicas.**

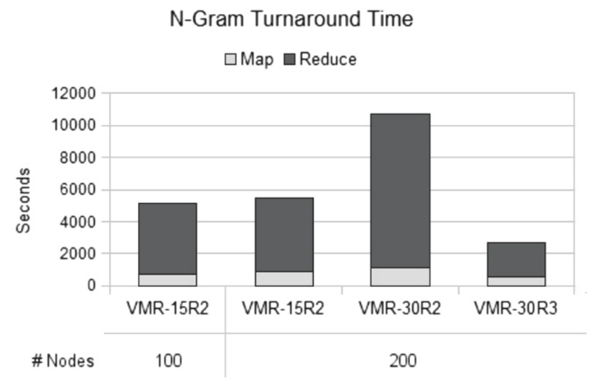

Figure 6 **VMR execution of N-Gram application, with a variable number of nodes, reduce tasks, and replicas.**

with 100 nodes. It is worth noting that despite the significant increase in clients, only the VMR-30R3 experiment showed improvement. This is explained by the fact that the execution of the reduce stage occupies the majority of the turnaround time.

The VMR-15R2 and VMR-30R2 scenarios have 15 and 30 reduce work units, respectively. However, both use only 2 replicas in the reduce step. With two replicas, each reduce task is indispensable for the MapReduce job to finish because a quorum of two identical results is required for validation. This means that, regardless of the number of clients reduce work units, a single reducer may delay the whole job.

The worst performance was obtained with VMR-30R2, when using more reduce work units, with a low replication factor. This is due to the increased probability of a slow or faulty node being assigned a reduce task. With 30 reduce work units, and a replication factor of 2, there are 60 tasks that may negatively impact the MapReduce job's turnaround time. On VMR-15R2 there are only 15 reduce work units, which translates to half the total number of tasks.

On the other hand, VMR-30R3 uses 3 replicas for each reduce task. In this case, only the first two returned results need to be validated (assuming they are both correct). This approach, despite creating overhead in data transfer and computation, allows the system to overcome lazy or slow workers, and offers a substantially improved performance.

We also performed the same tests in a scenario with VCS clients, because they allow us to better evaluate the impact of replication on the execution of MapReduce jobs. Figure 7 presents the results obtained when running the N-Gram application, which support our previous findings. The results of VMR-30R3 (the best performance settings for VMR) are also included, for comparison. Just as in the VMR example, merely increasing the number of nodes and reduce work units while maintaining two

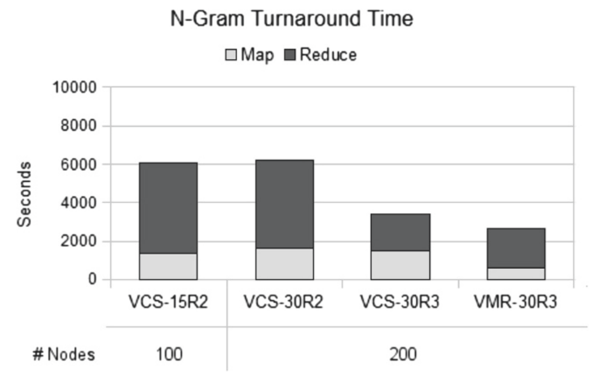

Figure 7 **VCS and VMR execution of N-Gram application, with a variable number of nodes, reduce tasks, and replicas.**

replicas (VCS-30R2) does not yield any benefits. However, a higher replication factor in the reduce step (VCS-30R3) improves the overall application turnaround time.

The results from the NAS EP application were similar, so we do not present them here. Application turnaround was decreased in 50%, mostly due to the reduce execution speedup. This is further discussed in the following section, on Network Traffic.

We conclude that, when using a larger number of nodes, it is worth providing a higher level of replication in the reduce stage, to account for stragglers and possible faulty behavior. Despite the obvious improvements, there were cases in which 2 out of the 3 replicas for a reduce work unit were executed by slow or faulty nodes. In this scenario, the validation of the whole MapReduce job was delayed due to this particular work unit. To overcome this shortcoming, we are considering introducing different scheduling and node selection algorithms in future work.

Finally, we have also evaluated our system's performance against Hadoop, an implementation of MapReduce tailored for clusters. VMR is designed for deployment over the Internet, and therefore cannot be expected to provide the same level of performance as Grid or Cloud Computing platform. However, it is interesting to find out VMR's overhead compared to a cluster execution of MapReduce. Thus, we ran Hadoop in our local cluster, which takes advantage of idle resources in computer labs. The cluster has access to 10 dedicated nodes, but it is able to use up to 90 nodes. For the comparison to be more precise, we ran a word count application, with varying input sizes. The results can be found in Table 1.

Unlike VMR, Hadoop does not replicate tasks, only data. This means that only one copy of the intermediate data (i.e., map outputs) is transferred from mappers to reducers. VMR, on the other hand, replicates the reduce task 3 times, which means that each map output is sent to 3 different hosts (reducers) in every execution. In the word count case, for each 1 GB input, 3 GB of intermediate data is transferred by VMR, while in Hadoop the same amount used as input (1 GB) is sent from mappers to reducers. Therefore, we include results for larger input data to compare scenarios with similar data profiles. The results show that Hadoop is around 5 times faster when

Table 1 VMR and Hadoop execution of Word Count application

Platform	Input (GB)	Intermediate (GB)	Turnaround Time (s)	Speedup
VMR	1	3	2475	1
Hadoop	1	1	508	4.87
	2	2	1079	2.3
	3	3	1485	1.67

executing the word count application on a cluster, when compared to an Internet deployment of VMR, for a 1 GB input. When the same total amount of data is handled by both systems (2 GB input for Hadoop), Hadoop is able to finish execution in around half the time of VMR. When it has to handle two thirds of the data (3 GB Hadoop input), VMR is able to perform only slightly worse than Hadoop.

It is worth noting that our system was not designed to perform better than data-intensive frameworks deployed in a high-bandwidth, low-latency environment. Our main contribution is a system that it is designed for the Internet, as a large-scale deployment, and that takes advantage of free volunteer resources. This creates many challenges and obstacles that are not found in cluster or data-center environments. The trade-off between execution speed and cost is advantageous for VMR whenever tight deadlines are not an issue, since VMR can obtain a similar performance for almost no cost. Clusters, on the other hand, have significant associated costs (i.e., maintenance, administration, and energy).

A summary of the turnaround time of the four applications running on PlanetLab is presented in Table 2. The Data Footprint corresponds to the size of input and intermediate data used by each application. This allows us to guarantee that all applications have to handle similar amounts of data. For each application, the total footprint is between 4 and 6 GB. We have also included the Hadoop results presented previously, to act as a baseline for comparison. The Hadoop implementation of word count is about 5 times faster than applications executed by VMR.

As we can see, the application with the largest footprint (NAS EP) still delivers a good level of performance when compared to the the remaining applications. This can be attributed to the fact that the intermediate files are much larger than the input and output data. By moving all data transfers to the client, VMR is able to better utilize bandwidth from volunteers (in this case, PlanetLab nodes), and achieve good turnaround time. The same can be observed in the N-Gram results.

We believe the results obtained with VMR can be translated to a real-world system, even considering the variable resources of hosts spread over the Internet. This is due to our system's ability to adapt to the available bandwidth between mappers and reducers. In cases where mappers have limited upload bandwidth, for example, we can simply increase the number of map tasks, by splitting map input into smaller chunks. It is also possible to further increase map task replication, thus making it more likely to have faster mappers upload their output to reducers. On the other hand, if the problem is in the reduce step, we can very easily increase the number of reduce tasks, by adapting the hashing algorithm that is used to divide map outputs into different key ranges (each reducer is responsible for a unique range). In our experiments, nodes have an average network download bandwidth of approximately 700 KB/s, while the nodes used as servers have around 10Mbit/s of upload bandwidth. These values are not too far from the average client and server bandwidths.

Table 2 Turnaround time for all four applications running VMR

		Data Footprint	
Application	**Turnaround time**	**Input**	**Intermediate**
NAS EP	2569.5	400 KB	5.9 GB
Inverted Index	2547.7	1 GB	4.25 GB
N-Gram	2678.3	0.5 GB	4.15 GB
Word Count	2475.4	1 GB	3 GB
Word Count (Hadoop)	508.3	1 GB	1 GB

3.4 Network traffic

We measure upload and download traffic in the VMR server (see Figure 1), for VMR and VCS clients while running the applications. Monitoring the network traffic on the server provides a more accurate measure of its overhead. It also allows us to quantify the impact of our solution concerning the decentralization of the VC model. The server download traffic corresponds to the amount of data received by the server from the clients, while server upload traffic consists of the amount sent by the server to the clients. Note that, as mentioned in the previous section, VMR has a much lower application turnaround than VCS. This is why the VMR line in Figure 8 stops around second 4000 (the same happens in Figure 9), while VCS only finishes its execution much later. This can be observed in all experiments presented in this section.

The server upload traffic while running the word count application is presented in Figure 8. We can see that up until around second 2000, both the VCS and the VMR

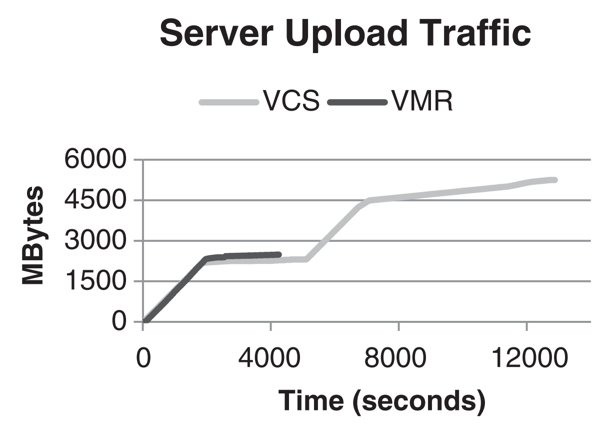

Figure 8 Server upload traffic when running word count application.

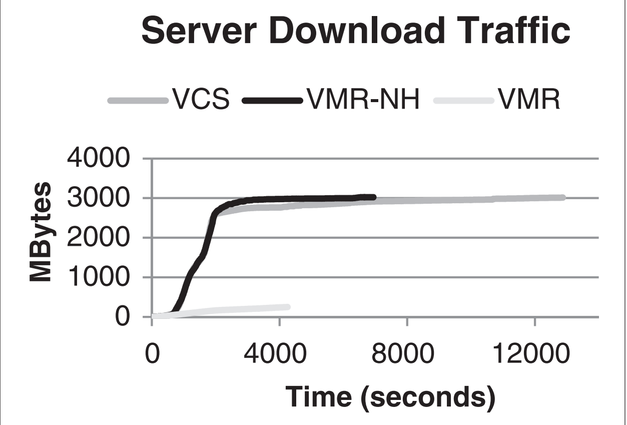

Figure 9 Server download traffic when running word count application.

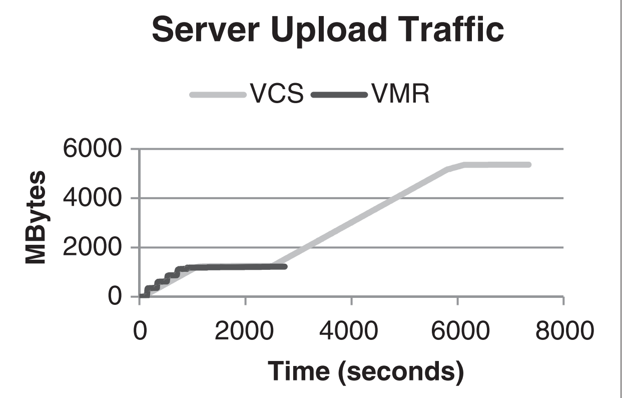

Figure 10 Server upload traffic when running N-Gram application.

server send the required map input files to the clients. However, once that step is completed, the VMR server is no longer responsible for uploading map outputs to reducers, unlike VCS which holds that responsibility. That explains the steep increase in the VCS line, around second 5000. Our server is required to upload 2,5 GB to clients, whereas VCS uploads more than double that amount.

Figure 9 shows the results on server download traffic. We include the results of VMR-NH to show the difference between returning hashes (VMR) or map outputs (VMR-NH). Since VMR-NH clients must return the map output back to the server, exactly as the regular VCS clients, the server receives the same amount of data from the clients in both cases. This is clearly shown in Figure 9, where we can see VMR-NH reaching the same value as VCS. VMR, on the other hand, through the use of hashes has almost completely eliminated data transfers from clients to the server. The VMR server receives a mere 250 MB from clients, a value 10 times smaller than VCS's 3 GB.

The inverted index application experiments yielded very similar results, so they are not shown here. VMR is able to reduce the amount of data sent to to clients from 6.5 GB to 2.3 GB and cut downloaded data by 96%.

N-Gram presents a different scenario from the two other applications, so it is worthwhile to analyze its results. The server upload traffic running N-Gram is shown in Figure 10. It is clear that there is a significant difference in the amount of data sent by the server to VMR and VCS clients. This is due to the large size of intermediate files, which causes the server in the VCS experiment to upload almost 5 times more data than the VMR scenario in the reduce step.

The server download traffic is exhibited in Figure 11. Here, we can see the benefits of using hashes for map task validation. Up until second 2000, the VMR server has received almost no data from the clients. At around that time in the experiment, reducers that finished their task began sending their output back to the server. The VMR server downloads a total of 820 MB from the clients. On the other hand, the VCS server is responsible for downloading all map outputs from mappers, which corresponds to the steep increase up until second 4000. The VCS server is required to download 6 times more data than VMR.

Finally, the experiments with the NAS EP application are conducted with two different inputs. For the first set of tests, we use a smaller number range for input, which translates into a relatively small intermediate data file size: each map task creates around 15 MB of output data.

The upload traffic for a server running NAS EP with the small intermediate data is shown in Figure 12. The server starts by sending the input files and the map executable to VMR and VCS clients. The reduce executable is also sent to the clients in both scenarios. This corresponds to around 500 MB, reached around second 700 in VMR. However, the VCS server must also send the reduce inputs to the clients, unlike VMR. This accounts for the steep

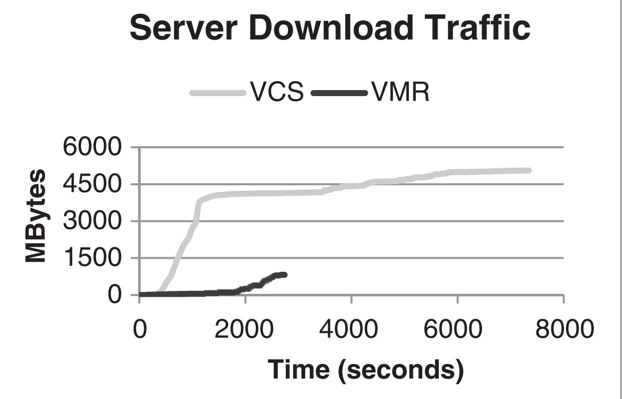

Figure 11 Server download traffic when running N-Gram application.

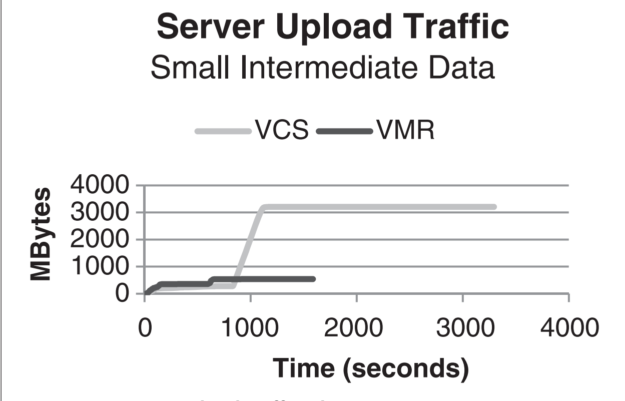

Figure 12 Server upload traffic when running NAS EP application.

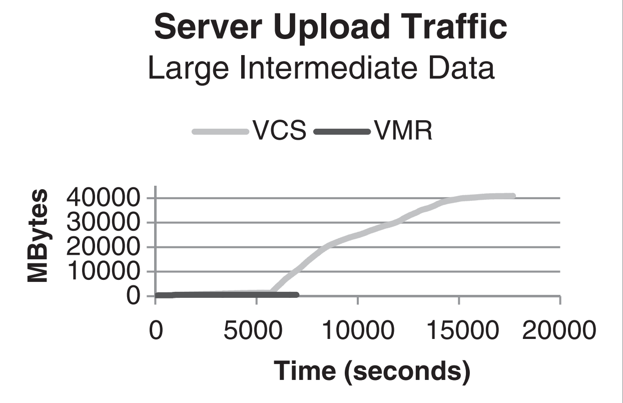

Figure 14 Server upload traffic when running NAS EP application.

increase around second 850. The VCS server is required to send over 3 GB of input data to clients.

However, the biggest difference in server load can be attributed to the download traffic, shown in Figure 13. As we mentioned previously, the NAS EP application produces very small reduce output files. Since VMR clients do not send the map output to the server, almost no data is received by the VMR server during the MapReduce job execution. The VCS version, on the other hand, receives over 3,7 GB of data.

For the final experiment, we increase the number range used as input for the NAS EP application. This in turn increases the intermediate data file size by an order of magnitude: each map task creates up to 300 MB of output data.

The upload traffic for a server running NAS EP with the large intermediate data is shown in Figure 14. As in the previous example, there is an initial upload of input files to clients, which is trivial when compared to the amount of data sent by the server to VCS clients. On the whole, the VCS server sends 40 GB of data to clients, which mostly corresponds to the map output files. As in the smaller input range, the VMR server only has to send around 500 MB to clients.

Once again, the most noticeable reduction in server overhead is found in the data received by the server. As we can see in Figure 15, the VMR clients send a nominal amount of data to the server (a little over 60 MB). The centralized VCS scenario requires all map output files to go through the server. Thus, over 40 GB are sent to the server during the execution. This means that VMR is able to reduce server download traffic by two orders of magnitude, to a value that is more than 600 times lower.

As expected, VMR is naturally suited for applications with intermediate data that is much larger than their input and output. The centralized approach currently used by VC systems, exemplified by VCS, on the other hand creates numerous problems. The massive amount of data that goes through the server not only creates overhead on a single node, but it also creates significant strain on the network.[g]

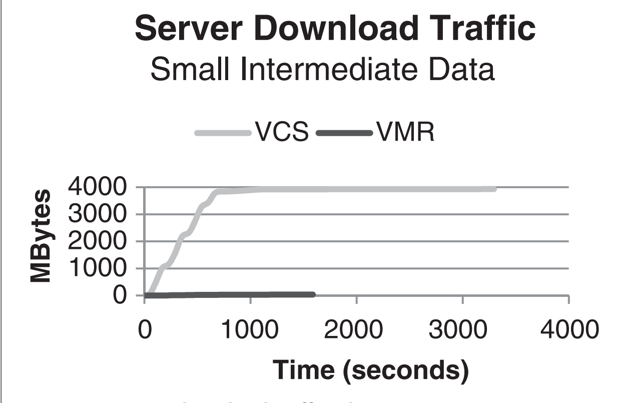

Figure 13 Server download traffic when running NAS EP application.

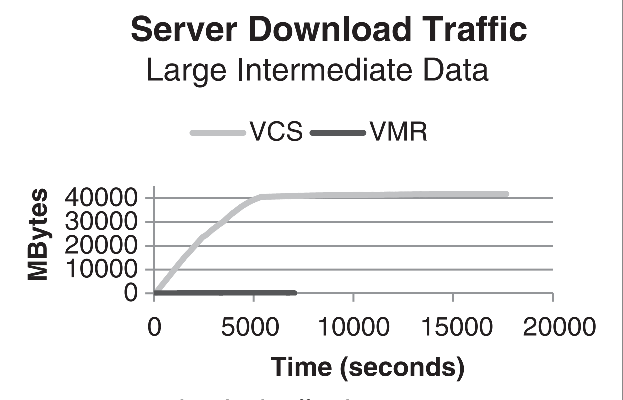

Figure 15 Server download traffic when running NAS EP application.

Therefore, we can conclude that VMR not only can perform better than VCS when running jobs with large intermediate files, but is also able to alleviate the server's network connection.

Performing a similar analysis on network traffic in clients (during the transition from map to reduce) may yield interesting results and help us better understand the performance of VMR. However, it is not easy (or sometimes even possible) to measure network traffic in clients, since they were deployed in PlanetLab nodes. Unlike our server, which is running in a dedicated machine of our lab, each client may be sharing resources with other virtual machines, at any time. This means that there is little to no control on the available bandwidth, and severely reduces the effectiveness of monitoring system resources in clients. Table 3 presents average network traffic and data transferred per client. We present the results obtained by reducers separately, due to the smaller number of nodes that are chosen for that role (90 reduce tasks). On the other hand, all 200 clients are usually mappers since there are 300 map tasks available.

For the inter-client network bandwidth experiment, 200 VMR clients executed the N-Gram application with 30 reduce work units, and a replication factor of 3. This means that out of the 200 hosts, 90 (30 workunits replicated three times) of them are chosen as reducers. The average download speed for the reducer is higher than the remaining nodes, since they are responsible for downloading all input data directly from mappers. These values correspond to the average obtained through system monitoring, during the entire duration of a MapReduce job execution, in which clients may often be either performing local I/O or CPU operations. Furthermore, as we mentioned previously, relying solely on system monitoring is not enough to provide a more accurate evaluation on client to client network traffic.

Therefore, we have instrumented the VMR client to measure the average speed when receiving files from other clients (i.e., acting as reducers). The average transfer speed for reducers downloading map outputs is around 1.1 MB/s. This considerably high download speed can be attributed to the map task replication and the random node selection mechanism when choosing which mapper to download from. By providing a large number of available mappers, the system is able to take advantage of clients' bandwidth. This resource is typically underutilized, as we can see from the average download and upload speed for all clients. Due to the inherent overhead created by middleware instrumentation, VMR currently does not monitor the upload speed or the amount of data transferred by each client. We are considering extending the monitoring features provided, while keeping in mind that any changes must not impact the system's performance.

3.5 Overhead

To better evaluate the impact of our system, we measure both CPU and memory use on the server. The experiments ran with the server deployed on PlanetLab yielded inconclusive results, since during each execution the server node could be running tasks from other virtual machines. Therefore, we present the results from experiments running the N-Gram application that used a local machine as the server.

The CPU utilization measurements are shown in Table 4. VMR and VCS present similar average values. The low CPU utilization can be explained by the data-intensive characteristics of the applications. During most of the execution, the server would either be transferring data or awaiting client requests, instead of performing CPU intensive tasks (e.g., creating work units).

In the course of our experiments we also measure the memory usage. The values observed are shown in Table 5. As we can see from the average memory utilization, our system did not create any overhead during the VMR server execution. In fact, VMR's average values are lower than those of VCS. This can be attributed to the lower memory required to store the intermediate files that were returned by VCS clients, and validated at the server. In VMR only the hashes are returned and compared in order to reach a quorum. These results show that VMR does not impact the VC server in any way in terms of memory or CPU utilization.

4 Related work

Combining the concepts of Cloud and VC was proposed in [16], in which the authors studied the cost and benefits of using clouds as a substitute for volunteers or servers. This comparison was performed using information from the SETI@Home project, taking into consideration its I/O operations, storage and throughput requirements. To calculate the cloud's cost, the authors used Amazon's EC2. The authors conclude that it is only advantageous to deploy a VC server in the cloud for small projects, with at most around 1400 volunteers. Although this paper deals

Table 3 Client Network Bandwidth averages in VMR

VMR clients	Download speed (KB/s)	Upload speed (KB/s)	Data received (MB)	Data sent (MB)
All clients	84	48	243	167
Reducers	178	61	418	143

Table 4 Server CPU utilization

CPU Utilization	VCS (%)	VMR (%)
Average	3.13	0.6
Std. Deviation	6.5	1.31

with the same research areas, its application and goals are orthogonal to our work. It would be possible to use cloud resources as an alternative to volunteers, whenever harder deadlines were set or more resources were needed.

P2P-MapReduce [17] is a P2P model under the MapReduce framework. The system is tailored to a dynamic cloud environment, creating a cloud of clouds. P2P-MapReduce makes use of a general-purpose P2P library, JXTA [18], which organizes peers into groups based on their interests or services offered. This network dynamically assigns the MapReduce master role and manages master failures in a decentralized fashion. Ordinary nodes are "promoted" to masters whenever the percentage of masters in the system falls beneath a certain threshold. This means that the first node of the system will always be a master. This may be a problem, since there is a possibility that machines with lower availability will be given more responsibility, thus slowing down the system.

Although MapReduce was initially developed by Google [7], Apache's Hadoop[h] is the most widely used implementation[i]. Hadoop is open source, unlike Google's MapReduce implementation, thus facilitating its adoption by a larger number of institutions. MOON (MapReduce On opportunistic eNvironments) [19] is a Desktop Grid system that proposes an extension to Hadoop that implements adaptive task scheduling to account for node failure. MOON is tailored for a cluster environment, such as a research lab, in which nodes are trusted or even dedicated. It takes advantage of a two-layer node organization, in which a small set of reliable nodes are used to guarantee a certain level of availability. The remaining nodes are considered to be volatile, and are used as "cheap" resources, often unavailable, but easily replaced. The authors reach an interesting conclusion during their experiments: Hadoop is unable to finish MapReduce jobs whenever there are frequent node failures.

Table 5 Server memory utilization

Memory Utilization	VCS (MBytes)	VMR (MBytes)
Average	6186	4239
Std. Deviation	765	397

MapReduce was also adapted to desktop grids in [20]. The system was designed on top of BitDew [13], a middleware the handles data management through the use of various transfer protocols. The authors claim it is able to run MapReduce jobs on XtremWeb [10], over the Internet. However, their experiments were conducted in a cluster interconnected by Gigabit Ethernet. This environment more closely resembles the common scenario of XtremWeb, which consists of a federation of research labs. Nodes in this system are divided into 2 categories: stable, which are dedicated machines that act as the XtremWeb *master*, and handle the MapReduce and BitDew *services* (this role is typically fulfilled by a single node); and volatile, which correspond to the workers responsible for task execution. In MapReduce jobs, each reducer is sent all map output files for each replicated map task. Once it has obtained a required number of intermediate results, it is assumed that the result that appears most often is assumed to be correct.

Moca et al. [21] studied the effects of sabotage when running MapReduce jobs on the previously described system. The authors try to identify the impact that a faulty node may have on a MapReduce job correctness. The authors only propose and test using majority voting, through simulation, which is able to achieve an acceptable error rate with a replication factor of 3. They also conclude that a higher replication factor would create an unbearable communication overhead.

Having the reducers receive all map replicas completely removes the server from the intermediate validation process. While this eliminates client-server intermediate file transfers, it also creates 2 significant problems in communication overhead and Byzantine fault tolerance. First, this algorithm requires all map outputs to be sent to reducers. In the case of a higher error or fault rate, this would create a large volume of unnecessary data transfers, as many map outputs would be incorrect. Even in a typical scenario with lower error rates, all incorrect files would still have to be uploaded to reducers for them to be validated. Secondly, this creates a problem in identifying Byzantine behavior. Whenever an incorrect reduce result is obtained, the system has no way of knowing if there was an error in the map or reduce phase, since it did not have access to map outputs. This can be aggravated if a reducer is exhibiting Byzantine behavior, either by purposefully sabotaging the execution or by simply encountering bugs or hardware faults.

XtremWeb and MOON are Desktop Grid systems, meant for deployment on distributed clusters and data centers. VMR, on the other hand, is actually tailored for a truly volunteer environment over the Internet. By moving from benchmarks and proof-of-concepts to actual applications in a realistic testbed, we can state with more certainty what are the advantages and shortcomings of this paradigm on a volunteer computing environment.

5 Conclusion

We have presented VMR, a Volunteer Computing platform that leverages client resources in order to execute MapReduce applications over the Internet. Our system is able to tolerate volunteer faults, and transient server failures. Furthermore, it is compatible with existing VC systems (in particular BOINC). VMR significantly reduces the dependence on the central server, which is typically overburdened in current VC platforms, thus allowing it to obtain a better performance.

We evaluated VMR by measuring the application turnaround, server network traffic and overhead while running widely used MapReduce applications, which are representative of MapReduce jobs deployed in production environments. Our solution was able to improve the performance of all the MapReduce jobs we tested. The map stage was up to 4 times faster than in an existing VC system. The reduce step also showed an improvement, thus reducing each MapReduce job's execution time down to less than half.

Increasing task replication does not improve the system's performance with a smaller number of nodes. However, with 200 nodes, we were able to conclude that having a larger pool of clients does correspond directly to a performance boost. This was attributed to the lower impact of slow or faulty nodes on the job turnaround time.

Regarding the server's network traffic, VMR reduced server download traffic by an order of magnitude on the word count and inverted index applications. The N-Gram application provided a scenario with large intermediate data and large outputs. Therefore, we were able to witness a decrease in uploaded data to 20% of the existing VC system server's value. However, it was the NAS EP application that showed the biggest advantage of VMR on server traffic reduction. Due to its large intermediate files, and small output and input data, VMR was able to reduce network traffic consumption by two orders of magnitude. It reduced server download bandwidth more than 600-fold, when compared to VCS.

We were able to conclude that VMR not only can perform better than VCS when running jobs with large intermediate files, but is also able to significantly remove the burden on the server's network connection. It is important to note that the changes we introduced did not create any significant or visible overhead on the server side. Considering these results, and taking into consideration the typical compute-intensive nature of VC applications, we can determine that MapReduce jobs with large intermediate files are more suited (than others with smaller intermediate files) for VC. This can be explained by the reduced overhead on the server, since clients handle most of the heavy data communication among themselves. Furthermore, small input and input data can reduce the influence of server-client transfers. There are quite a few examples of real-world applications with these characteristics, such as image rendering [22], market basket analysis, and N-Gram based functions (e.g., biological sequence analysis [23]).

Endnotes

[a]Distributed.net website. `http://www.distributed.net`

[b]List of active VC projects. `http://www.distributedcomputing.info/projects.html`

[c]Amazon EC2. `http://aws.amazon.com/ec2`

[d]A VC Project runs on top of existing middleware (e.g. BOINC) by developing an application and defining all parameters concerning its execution. The middleware takes care of all the mechanisms necessary for executing a distributed application over the Internet, making life easier for Project developers. They only have to make sure their tasks are properly configured and provide a publicly accessible machine to act as the VC server. Each volunteer chooses projects to attach its VC client to, based on the applications the user would like to run. A client may be attached to several projects at the same time, sharing the machine's resources in a round-robin fashion.

[e]Nebula and Plush. `http://plush.cs.williams.edu/nebula/`

[f]NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html

[g]This was very apparent during our experiments. In fact, it was so inconvenient that all HTTP traffic to our server was blocked by the network administrators during one of our runs.

[h]Apache Hadoop. `http://hadoop.apache.org/`

[i]List of institutions that are using Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors read and approved the final manuscript.

Acknowledgements
This work was partially supported by national funds through FCT – Fundação para a Ciência e Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/113993/2009 and PEst-OE/EEI/LA0021/2013.

Received: 27 July 2013 Accepted: 28 October 2013
Published: 12 November 2013

References

1. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) SETI@home: an experiment in public-resource computing. Commun ACM 45: 56–61

2. Beberg AL, Ensign DL, Jayachandran G, Khaliq S, Pande VS Folding@home: Lessons from eight years of volunteer distributed computing. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 23-29 May 2009, IEEE Computer Society, Washington, DC, USA, pp 1–8
3. Anderson DP (2004) BOINC: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International workshop on grid computing, GRID '04, IEEE Computer Society, Washington, DC, USA, pp 4–10
4. Cisco I (2012) Cisco visual networking index: forecast and methodology, 2011–2016. CISCO White Paper. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html
5. Snir M, Otto SW, Walker DW, Dongarra J, Huss-Lederman S (1995) MPI: The Complete Reference. MIT Press, Cambridge, MA, USA
6. da Silva e Silva FJ, Kon F, Goldman A, Finger M, de Camargo RY, Filho FC, Costa FM (2010) Application execution management on the InteGrade opportunistic grid middleware. J Parallel Distributed Comput 70(5): 573–583
7. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun. ACM 51: 107–113
8. Lamport L, Shostak R, Pease M (1982) The Byzantine generals problem. ACM Trans Program Lang Syst 4(3): 382–401
9. Cirne W, Brasileiro F, Andrade N, Costa L, Andrade A, Novaes R, Mowbray M (2006) Labs of the world, unite!!! J Grid Comput 4: 225–246
10. Cappello F, Djilali S, Fedak G, Herault T, Magniette F, Néri V, Lodygensky O (2005) Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. Future Gener Comput Syst 21: 417–437
11. Costa F, Kelley I, Silva L, Fedak G (2008) Optimizing data distribution in desktop grid platforms. Parallel Process Lett (PPL) 18(3): 391–410
12. Costa F, Silva L, Fedak G, Kelley I (2008) Optimizing the data distribution layer of BOINC with BitTorrent. Parallel Distributed Process Symp Int 0: 1–8
13. Fedak G, He H, Cappello F (2008) BitDew: a programmable environment for large-scale data management and distribution. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, vol 1-45. IEEE Press, Piscataway, NJ, USA, p 12
14. Chun B, Culler D, Roscoe T, Bavier A, Peterson L, Wawrzoniak M, Bowman M (2003) PlanetLab: an overlay testbed for broad-coverage services. SIGCOMM Comput Commun Rev 33: 3–12
15. Marsaglia G, Bray TA (1964) A convenient method for generating normal variables. Siam Rev 6(3): 260–264
16. Kondo D, Javadi B, Malecot P, Cappello F, Anderson DP (2009) Cost-benefit analysis of Cloud Computing versus desktop grids. In: Proceedings of the 2009 IEEE International symposium on parallel& distributed processing, IPDPS '09, IEEE Computer Society, Washington, DC, USA, pp 1–12
17. Marozzo F, Talia D, Trunfio P (2012) P2P-MapReduce: Parallel data processing in dynamic Cloud environments. J Comput Syst Sci 78(5): 1382–1402
18. Gong L (2001) JXTA: A network programming environment. Internet Comput IEEE 5(3): 88–95
19. Lin H, Ma X, Archuleta J, Feng Wc, Gardner M, Zhang Z (2010) MOON: MapReduce On Opportunistic eNvironments. In: Proceedings of the 19th ACM International symposium on high performance distributed computing, HPDC '10, ACM, New York, NY, USA, pp 95–106
20. Tang B, Moca M, Chevalier S, He H, Fedak G (2010) Towards MapReduce for desktop grid computing. In: Proceedings of the 2010 International conference on P2P, parallel, grid, cloud and internet computing, 3PGCIC '10, IEEE Computer Society, Washington, DC, USA, pp 193–200
21. Moca M, Silaghi G, Fedak G (2011) Distributed results checking for MapReduce in volunteer computing. In: Parallel and distributed processing workshops and Phd Forum (IPDPSW), 2011 IEEE international symposium on, IEEE Computer Society, Washington, DC, USA, pp 1847–1854
22. Perry R (2009) High speed raster image streaming for digital presses using the Hadoop file system. HP Laboratories, HPL-2009-345, Technical Report. http://www.hpl.hp.com/techreports/2009/HPL-2009-345.html
23. Ganapathiraju M, Manoharan V, Klein-Seetharaman J (2004) BLMT. Appl Bioinformatics 3(2-3): 193–200

doi:10.1186/1869-0238-4-18
Cite this article as: Costa *et al.*: Internet-scale support for map-reduce processing. *Journal of Internet Services and Applications* 2013 **4**:18.