Social-Networking@Edge

Miguel Guerreiro Anão Borges

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof Dr. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Dr. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Dr. João Carlos Serrenho Dias Pereira

June 2015

Acknowledgements

This dissertation would not be possible without the support and patience of others. I am forever grateful to them and would like to dedicate this section to them.

I would like to thank to professor Luís Manuel Antunes Veiga for the opportunity to work on this topic, for his endless patience and great feedback. Without his knowledge and guidance, I would not be able to learn this much.

A special thank to my academic and professional colleagues for all technical discussions that helped me find solutions to some of the problems encountered during this study.

Also, I would like to express my sincere gratitude to my parents. To my father who has passed on his interest about this field to me, to my mother who always taught me to pursue my dreams and never give up, and to my family and friends who have always encouraged and supported me to to succeed. This journey would not be possible without their support.

Last but surely not the least I would like to thank Juliana Aniceto who has always been by side during this path. Thank you for loving, supporting and cheering me on.

22nd of June 2015, Lisbon
Miguel Guerreiro Anão Borges

–To my family

Resumo

As Redes Sociais tornaram-se bastante populares nos últimos tempos. Estas organizações têm um número crescente de utilizadores e recolhem *petabytes* de informação sobre as suas interações. Ter a capacidade de entender esses dados é crucial para o seu sucesso porque lhes permite inferir tendências, recomendar amigos e escolher o conteúdo que mais se adequa a cada utilizador.

Foram desenvolvidas algumas ferramentas que permitem analizar tais quantidades de dados. Uma das soluções mais usadas é a *framework de MapReduce*, porque oferece um paradigma de programação simples que facilita o desenvolvimento de aplicações distribuídas e que automaticamente trata dos problemas relacionados com replicação, tolerância a faltas e balanceamento de carga. Contudo, usar estas ferramentas requer uma capacidade de processamento considerável, que normalmente é suportada por enormes infraestruturas centralizadas. A criação e manutenção dessas estruturas acarreta custos altos e uma elevada pegada ecológica. Ao mesmo tempo, as máquinas usadas pelos utilizadores das redes sociais, têm uma capacidade de processamento cada vez maior que normalmente não é usada na sua totalidade.

Esta tese propõe uma solução que faz uso dos recursos computacionais dos utilizadores, enquanto utilizam o site rede social, para executar computação que normalmente seria efectuada em centros de processamento de dados. *Social-Networking@Edge* é uma *framework de MapReduce* que aproveita os recursos livres dos utilizadores para executar tarefas no seu *Web Browser*, tratando dos problemas relacionados com distribuição de dados, tolerância a faltas e balanceamento de carga. Ao fazer uso desta solução, as redes sociais seriam capazes de suportar os mesmos requisitos de processamento de um modo muito mais eficiente em termos de custos e reduzindo a sua pegada ecológica.

Abstract

Online Social Networks (OSNs) have become very popular nowadays. These organizations have millions of users and collect petabytes of data from their interactions. Having the capability of understanding that data is crucial for their success, as it enables them to infer trends, recommend friends and present content that is suited for users.

Some tools have been developed that enable the processing of such amount of data. One of the most used solutions is *MapReduce framework*, because it provides a simple programing model that eases the development of scalable distributed application and automatically handles the problems of replication, fault-tolerance and load balancing. However, using these tools requires considerable computing resources that are normally supported by massive centralized infrastructures. The creation and maintenance of such infrastructures leads to high ownership and environmental costs. At the same time, the machines used by the users of OSNs have an increasing computing capacity that is normally not being fully used.

This thesis proposes solution that makes use of users' computational resources, while they are using an OSN's website, for executing computation that would normally be done at datacenters. *Social-Networking@Edge* is a MapReduce framework that makes use of users' spare-cycles for executing tasks on their Web Browser while handling the problems of data distribution, fault tolerance and load balancing. By making use of such solution, OSNs would be able to support the same data processing requirements in a much more cost-effective manner and producing a smaller ecological footprint.

Palavras Chave

Redes Sociais

MapReduce

Partilha de Recursos

Web Browser

P2P

Keywords

Online-Social-Networks

MapReduce

Cycle Sharing

Web Browser

P2P

Index

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Online Social Networks (OSN) with millions of users are very popular nowadays, especially due to the success of sites like Facebook, Twitter and Google+. Interactions provided on their sites produce enormous amounts of information, leading to data-warehouses with petabytes of data (Thusoo et al. 2010). Having the capability to analyze that data is crucial for their success, enabling them to infer trends, recommend friends and present insights to advertisers.

Some tools have been developed that enable the processing of such amounts of data, among which, the most popular is MapReduce (Dean & Ghemawat 2008) as it presents a simple programming model that eases the development of scalable distributed applications for data processing by automatically handling common problems for distributed application such as replication, fault-tolerance and scheduling. In order to use these tools, large amounts of processing power and storage capacity are required, leading to the creation and maintenance of large data-centers that have an high cost and ecological footprint. As the number of users increases, these data-centers need to grow as well.

At the same time, the computing power made available at commodity machines owned by regular people has been increasing in terms of CPU and memory. However, most of the time, those resources are not being used in their total capability. Systems that make use of processing capabilities donated by users such as BOINC (Anderson 2004) and SETI@home (Anderson et al. 2002) have had great success, enabling the development of research programs in a very cost-effective manner but they are focused on specific projects and do not cope with OSN's requirements.

There have been some approaches that try to solve the scalability and data privacy problems of centralized OSNs by creating totally distributed OSNs, making use of their users' resources to allocate the data and computation needed to enable the OSN. However, building the same functionalities in a totally distributed environment while dealing with privacy issues and high rate of failure from users' computers is a much difficult task, resulting in OSNs with less functionalities and worse usability. At the same time, it is hard to convince users to migrate to a new OSN and donate their resources to the community. Other less radical approaches have also been used but they are mainly focused on saving the OSN's data on their users and do not handle its processing.

We propose a solution that makes use of users' idle resources, while they are using an OSN's website, for executing computation that would normally be done at data-centers. With Social-Networking@Edge it is possible to execute existing MapReduce workloads on the users' Web Browsers, allowing OSNs to offload some of their processing requirements. By using this approach it is possible to increase the OSNs' scalability, reduce their ownership costs and ecological footprint while maintaining the benefits of having a centralized architecture for other tasks.

The motivation for this work considers its applicability on OSNs, but the solution provided here may well be applied on other web based organizations with a large number of users.

1.2 Objectives

The main goal of this thesis is to demonstrate that it is possible to extract some of the computation that is currently being done at the OSNs' data-centers to the computers of their users. This would improve the scalability of the existing OSNs and decrease the costs of maintaining them. At the same time it would also decrease the costs of developing new OSNs.

In this work we will provide a solution for a MapReduce framework that is able to execute jobs on regular Web Browsers while users are visiting the OSN's website. The solution provided should address several issues:

1. Developers using our system must be able to submit MapReduce jobs for execution with a custom specification that allows them to specify the map and reduce code. They should also be able to choose the number of mappers and reducers that participate in the job's execution.

2. The solution must be able to execute MapReduce jobs on regular Web Browsers;

3. It should allow the distribution of MapReduce jobs' execution across a large number of nodes in a scalable manner;
4. The solution should be able to use the available resource of each user's machine. Machines with more computing resources should have a larger contribution to the system's overall resources.
5. The computing requirements for the machines that must be maintained by OSNs should be significantly smaller when compared with existing solutions;
6. The solution presented must also take into consideration that the distributed nodes contributing to the system have a high rate of failure;

1.3 Contributions

The main contribution of this thesis can be summarized as follows:

1. Description of the state of art solutions related with big data processing systems, distributed OSNs and community clouds.
2. A scalable MapReduce framework capable of running MapReduce jobs written in *Javascript* and executed on regular Web Browsers. During jobs' execution, each running node is able to exchange intermediate data with other nodes using P2P communications. Faults of nodes are handled by using a checkpoint based approach. A simple scheduler algorithm is also provided that enables a fair distribution of load across the running nodes.
3. Experimental evaluation of the proposed solution, showing its main benefits, shortcomings and possible topics for future work.

1.4 Structure of the thesis

The rest of the document is organized as follows: in the next chapter (Chapter 2) we introduce the state of the art for three of the main research topics related with this work. Chapter 3 presents the architecture of the proposed solution. After that, on Chapter 4, we describe the main technical decisions and implementation details for the development of the proposed solution. The evaluation of the solution developed and the results obtained from testing the system with popular benchmarks is presented on chapter 5. Finally on chapter 6, we describe a few ideas to improve the proposed solution and draw some conclusions from this work.

1.5 Publications

The work presented in this dissertation is partially described in the following publication:

- Miguel Borges and Luís Veiga, SN-Edge (Comunicação), presented at INFORUM (Simpósio de Informática), Sep. 2014 (Borges & Veiga 2014)

2 Related Work

In this section, we address the state of the art of the research topics that we consider more relevant to our work, namely: Social Networks (Section 2.1), Big Data Processing (Section 2.2) and finally Community Clouds (Section 2.3).

2.1 Social Networks

"An online social network can be generically understood to be some kind of computer application which facilitates the creation or definition of social relations among people based on: acquittance, general interests, activities, professional interests, family, associative relations and so on." (Nettleton 2013).

On an OSN each user has a profile where he/she can define his/her description, which includes the age, location and the main interests of the user. Normally these profiles are public and are available for others to see.

In 1979 people started using *Usenet* to communicate; each user could post an article that was logically organized into hierarchies of subjects. Latter, in 1997, SixDegrees was created and attracted millions of users, enabling them to create a profile, have a friends list, connect with others and meet new people. After that several new OSN followed like *Friendster* (2002), *Myspace* and *Linkedin* (2003), *Facebook* (2004), *Twitter* (2006). Some of the most popular OSNs, in terms of number of active users are: *Twitter* with 200M monthly active users; *Linkedin* with 259M members; *Google+* with 540 million; and *Facebook* with 1.28 billion monthly active users.

The offered features vary greatly between OSNs but the main features are the ones that enable a user to search for other people and the establishment of connection between users. However, there are some commonly offered features, such as a chat, photo albums and a way for a user to publish messages or other content with his friends. Some OSN also offer social games where the user can compete and collaborate with each other.

OSNs have several technological challenges as they need to support an enormous number of user requests, to analyze the users' behavior, detect tendencies, perform data-mining operations on the social graph and detect bugs in the shortest amount of time possible. Their ability to obtain social information is determinant to their success since they can offer a better user experience and present the right information for the right users. For example, they know that if two people have similar interests and common friends, then it is likely for them to become friends.

The biggest OSNs have centralized architectures, i.e., they use Data Centers to build their infrastructure and to support their needs. This offers several advantages in terms of software development and they end up having more and better features.

Centralized OSNs have great success, but since all the content is hosted by a single provider, several issues have been raised by the community, most of them related with data ownership, privacy and possible loss of data in the case of a security breach. The fact that OSNs are normally advertising-driven businesses also creates pressure for using data in ways that violate the user's privacy. Privacy protection policies are provided but they can be changed by the social network along the way. From the point of view of the OSNs there are also some problems with a centralized solution since it is hard to scale in a cost effective manner and the Data Centers used have a high ecological footprint.

2.1.1 Distributed Social Networks

Taking those problems into account, some approaches of building a Distributed Social Network (DSN) have occurred. By distributing the data and processing of the social network to the users' computers they tackle the problems of infrastructure costs and the control of data is done by each user instead of one unique entity. This does not mean that the security problems stop but it is less likely that a major data breach will occur and the social network does not have as much power as it has on a centralized architecture.

DSNs still have to deal with similar issues of centralized solutions as they must scale to a great number of users, have good availability and the content must be secure. The content being hosted on the users' side does not solve all the problems by itself, since several new challenges are introduced on a decentralized approach. For example, it is much harder to develop new features as they have to deal with high rates of failure from the clients' computers and the complexity of the security assurance also increases. DSNs also have to deal with the fact that users may not want to offer their resources to the OSN, and so they need to find incentive mechanisms.

Building a fully decentralized solution can be considered a radical approach; others have tried to extract only part of the centralized infrastructure in order to reduce its costs and ecological footprint.

Next we start by introducing some of those systems, then on Section 2.1.2 we present totally distributed solutions and finally on Section 2.1.3 we refer to some totally distributed solutions that make use of users with more resources and higher availability.

2.1.1.1 User Assisted Online Social Networks

User Assisted Online Social Networks (Kryczka et al. 2010) aims at reducing the infrastructure costs of centralized online social networks by using user's free space. Centralized OSNs need to save all the user's information such as photos, videos, messages, etc. and this leads to high infrastructure costs. Some of this information is valuable to their business and they are not aiming at sharing it. However, some of the least important information could be distributed and stored through the users of the social network while critical information is kept on the social network's data-center. The authors assume that photos and videos are not that important to the OSNs business, but saving that content require lots of storage capacity.

By distributing that content across the users' computers and by using an intelligent placement of files based on historical data about locality and accessibility, it is possible to increase the availability for file's access. By placing content closer to the nodes requesting it, it would be possible to reduce the amount of traffic and the delay to access content.

To the best of our knowledge, the authors do not provide any implementation nor practical results about the system. The focus is on extracting content of the OSN but does not consider exporting computation to the clients.

2.1.1.2 Social Content Delivery Network

OSNs have to support a great amount of requests for reading and updating their content. *SocialCDN* proposes building a C*Content Delivery Network* (CDN) hosted by the users of the Social Network in order to reduce the amount of traffic and content reaching the OSNs' servers.

Some of the users are chosen to host caches and the set of all caches would form a CDN capable of hosting messages, images and videos for the social network. The users of the social network would then read and write content on the provided caches instead of making requests to the OSN's servers, saving them from great part of the load coming from user's requests. Choosing the users that should host caches is an important part of the system as each user should be able to read or write all the content from his friend's caches. The authors propose algorithms that try to minimize the number of nodes required to host caches and make the decisions based on social properties, social tie strength and social traffic patterns.

2.1.2 Fully Decentralized Social Networks

2.1.2.1 PeerSoN

PeerSoN (Buchegger et al. 2009) tries to solve some of the privacy preserving issues that exist on the centralized social networks by using encryption and a distributed access control over a P2P infrastructure, removing the use of a centralized authority. By using a P2P infrastructure the users gain independence from the OSN provider and through the use of encryption, they ensure privacy over their data.

All the files on the social network are saved in a DHT after being encrypted. The status of a user and his/her location is also saved on the DHT and is updated whenever the user changes location or logs out. A centralized public-key infrastructure maintains a list of public-keys for the users who have access to the content. When a user wants to share content with others, a new key is added for that content. The system also supports for revocation of keys that enables the users to stop sharing a given content.

The system also introduces a new feature in which there is no requirement of Internet connectivity for all the transactions, that is, if a peer is disconnected from the Internet is able to reach other peers, then they it can exchange information directly with them. This information can then be propagated through other peers when one of them has Internet connectivity.

PeerSoN information is saved at random nodes of the DHT, not taking into account neither locality information nor other information to increase locality and availability of content.

2.1.2.2 My3

My3 (Narendula et al. 2011) is a distributed OSN where all the social network information is saved on a distributed hash table (DHT) hosted at the user's computer. The social information of one user is saved by his/her friends and when a user does not have enough friends, other unknown users are chosen. As the info is saved by users, it can happen that a user enters on the network and none of the friends hosting its information is online. To minimize those cases, My3 replicates the profile information across several users. The system analyses the times at which users are online and chooses as hosts the ones that use the social network in similar schedules.

The profile information can be replicated across several nodes and each node has a complete copy of the user's profile. When another user needs to access that information, they can choose the node closer to him/her and make a request. It can happen that two replicas have inconsistent versions of the same profile. When this happens, the system propagates the changes to other replicas and provides *eventual consistency* guarantees. As these are weak consistency guarantees, it is possible that two users see different values for the same profile. It is also possible that a user changes something on the profile and then he/she cannot see that change for a given period of time.

When the host of the user's profile is not his/her friend on the social network, the system uses encryption mechanisms for ensuring privacy of data. However, when the two nodes have a friend relation, it is assumed that there is a trust relation between them and that the privacy requirements can be relaxed. In that case the information is saved without any encryption. *Friendstore* (Tran et al. 2008) applies a similar approach by making use of social network's trust relations between users to backup their disks' content. Users wanting to backup their disk content ask for their friends to save it. As the users trust each other, is less likely that security attacks will occur and so the system can use simpler techniques for saving the content.

2.1.3 Fully Decentralized with High Fidelity Users

2.1.3.1 SuperNova

SuperNova (Sharma & Datta 2012) has distributed architecture based on the existence of super-peers. Super-peers are nodes that have more bandwidth, storage space and availability than the normal user nodes.

Users' data can be saved at other users or at super-peers. Incentives are given to users who accept to provide their resources for the OSN as they receive a special statute. This statute can improve the user's reputation as it is visible by others in his community. Nodes that accept to donate their resources and provide great availability, bandwidth and storage space, are considered super-peers, in which case they have a special statute and they can also provide advertisements for the users connected to them.

Super-peer architecture is a good model for companies that want to provide social network services for their workers but are not willing to lose control over data exchanged on the OSN. Enterprises using this model can provide the super-peers that handle data for all the company's users.

Each user can choose where the content is saved by asking friends, unknown users or super-peers. When the user's friends do not provide him sufficient resources, super-peers can suggest them other users. In order to provide better suggestions, super-peers maintain register of the time at which users were online and then suggest user's with similar schedules. This functionality is especially important when a new user joins the social network and initially has no friends who can provide resources.

When a user wants to write on other user's profile, it sends a request and a time-stamp (logical clock) to the user and to others hosting the profile. Each peer receives the request and performs a local update. In case of receiving several update request at the same time, they first order the messages by times-tamp and then apply the updates.

2.1.3.2 Vis-A-Vis

Vis-A-Vis (Shakimov et al. 2008) introduces a solution for DSNs where the content is saved at user's server. Each user provides a server hosted in the cloud, which saves all the users' data such as friend's list, photos and messages. As the server is specific for a user and it contains all information, it is possible for the user to maintain control over his data.

Because the servers are hosted in the cloud, they would provide better availability, more bandwidth and space, facilitating the design of the OSN. However, this forces the user to pay for a cloud provider in order to have this infrastructure.

Users' servers make part of a DHT, forming overlay networks that resemble groups on the Social Network and they communicate with each other as peers through the use of DHT operations. As all the operations are provided by a DHT it is possible to search, join or leave a group in a very efficient and scalable manner. It is also possible to multi-cast information across all the server nodes.

Data can be marked as searchable or restricted. Searchable information is accessible to strangers but restricted information is encrypted and can only be shared by users that have trust among them.

Each user has a list of friends, when a new friend is added, the two users' server use Diffie Hellman protocol to exchange keys between each other. After that, the two parties use those keys for authentication and to establish a secure communication channel.

To ensure that all the nodes are connectable and that the group's information is accessible, the users' servers form a top-level DHT that contains public information for the network such as the list of users and public-groups description. Private groups are not publicly accessible and so the list of users on those groups is saved on the users that belong to the group.

Classification

	Availability	**Privacy**	**Direct Exchange of Information**
PeerSoN	Encryption	Encryption	Yes
My3	Replication and Schedule heuristics	Trusted Nodes and Encryption for non trusted	No
FriendStore	Trusted Nodes	Trusted Nodes	Yes
SuperNova	High Fidelity Users	Encryption and Overlay Networks	No
Vís-Á-Vís	User's Server	Server keeps all user's content	No

Table 2.1: Distributed Social Networks Classification

In this section we introduced some of the most relevant state-of-art system related with distributed social networks. We have seen that there are some work by the community directed to the decentralization of social networks for various reasons. Those solutions have addressed the problems of availability, privacy and distributed communications. Table 2.1 presents a summary of their main characteristics. However, the main focus of current solutions has been only on data distribution and they have not considered how to distribute the processing of large amounts of data that is crucial for the success of OSNs.

2.2 Big Data Processing

Today organizations tend to acquire increasing amounts of data, for example, Facebook datawarehouse has 300 PetaBytes of data and receive about 600 TeraBytes of new data daily [1], having the ability to analyze that data is crucial for the success of their business [2],but processing such a vast amounts of data introduces new technological challenges as the systems must scale efficiently.

When building programs that process big data sets, programmers have to deal not only with solving the specific problem they have at hand but also with problems related with parallelization, distribution and node failures. As those are not trivial problems, that part of the development ends up taking lot of time from their main task and having several bugs.

Organizations dealing with this kind of problem have developed some solutions that allow them to support big data analysis. The main solutions developed have chosen to parallelize data processing across a large number of machines. The several steps of data processing are expressed as a workflow that is then automatically parallelized by the system across several machines. Some of those solutions will be described on Section 2.2.1. Other solutions aiming at simplify the development of workflows have been presented and will be addressed at Section 2.2.2. On Section 2.2.3 we introduce some MapReduce schedulers.

Workflow solutions are usually optimized to work as batch systems, acting over static data and presenting a considerable delay from the moment the data is produced until the processing results are available. In the case where faster results are mandatory, some systems have been developed that compute the results directly over the data stream in real time. These solutions will be further explained on Section 2.2.4.

[1] https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb

[2] http://www.economistinsights.com/sites/default/files/downloads/EIU_SAS_BigData_4.pdf

2.2.1 Workflows

2.2.1.1 Directed Acyclic Graph Manager (DAGMan)

DAGMan (Couvares et al. 2007) was one of the first workflow systems, its development started in the late 1990's and is part of the Condor Project. Condor (Thain et al. 2005) is a batch system that focuses on providing reliable access to computing over long periods of time, instead of focusing in high performance. It was initially created with the goal of using CPU cycles of idle computers, but since then it has also expanded to work on dedicated computers and grid systems.

DAGMan has a simple interface for expressing workflows that run on top of the *Condor* batch system. Its main goal is to automate the submission and management of complex workflows involving many jobs, focusing on reliability and fault tolerance in the face of a variety of errors.

Data imports are also implemented by the user as jobs and can be represented as a node on the graph. Jobs are expressed by the user as directed acyclic graphs (DAGs) where each node represents a batch job to be executed and the arcs represent the execution order and dependencies between jobs. The reason for the graph to be acyclic is to ensure that the DAGMan will not run indefinitely. The system analyses the restrictions and dependencies between jobs and tries to parallelize as many as possible, automatically.

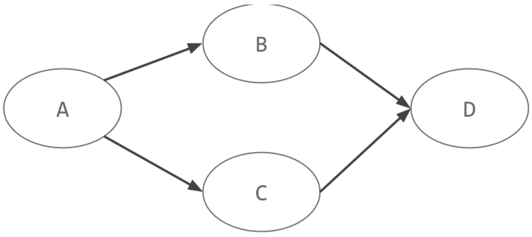


Figure 2.1: Directed Acyclic Graph

If a graph like the one on Figure 2.1 was submitted for execution then DAGMan would start by submitting the job A for execution. When it completes, as B and C have no dependencies, they can be submitted to the batch system at the same time. Then after waiting for the completion of the two jobs, D can be executed. Finally, when D is done, the graph is marked as completed and the results from executing the workflow are made available.

Nodes can execute arbitrary tasks, including load data from an external source or start a new DAG. This way the user code can re-execute the entire graph iteratively and create conditional branches.

Users can also define pre and post scripts that run respectively before and after a job execution. These scripts can modify the job description, and so it is possible to implement conditional graphs.

In the presence of errors, the system tries to overcome or work around them as many as possible. In the case of problems that cannot be solved automatically it requests the user to solve the problem manually.

The node jobs and DAGMan server itself are executed as batch jobs on Condor. When a given node fails, the system automatically retries its execution with a configurable amount of retries. After that number of retries the job is marked as failed. DAGMan server saves the state of the graph execution on a log file and in case of failures it can resume the operations after restart.

DAGMan differentiate data placement jobs from computational jobs. Data placement jobs are handled by a specialized system named *Stork* that provides an abstraction between the application and the data transfer or storage.

Stork takes care of queuing, scheduling, optimizing data placement jobs and fault tolerance for data transfer or storage jobs. On the presence of transient faults, the jobs are retried after a small delay, and if the data server fails a list of alternative data protocols can be used to overcome that failure. The system provides support for several data transfer protocols and data storage systems and support for new ones can be added by the programmer.

2.2.1.2 Dryad

Dryad (Isard et al. 2007) is also a system that provides support for the development of workflows and like DAGMan, the workflow is represented as an acyclic graph where each work unit is represented as a vertex and each communication flow is an edge.

The edges and dependencies between them are specified by the programmer, creating an arbitrary DAG that describes the application's communication pattern and indicate the data transport mechanisms (files, TCP pipes or shared memory FIFOs) that should be used as communication channels between the computation vertexes. Dryad eases the development of large distributed applications by automatically scheduling the execution across the cluster's computers and recovering from failures. Scalable performance is obtained by exploiting data and processing parallelism.

The system's main components are a Job Manager, responsible for control decisions, a Name Server, where the available computers are registered and Vertexes, where the computations are executed. A Job Manager controls the execution of the workflow graph by scheduling the work across the available resources and keeping track of the state at each vertex of the graph.

All the available nodes are registered at the Name Server that, when requested, shows the available resources and respective positions on the network topology. With this information, scheduling decisions can take location into account.

Vertexes are deployed on the available cluster's computers and take care of executing the workflow computation and data exchange. The data is exchanged directly between the vertexes so that the Job Manager does not constitute a bottleneck for data transfers. Several vertexes can run on the same process, in which case, the communication is done with low costs. Several instances of the same vertex can also run at the same time for performance and fault-tolerance reasons.

In order to detect failures, each vertex daemon sends regular heartbeats to the Job Manager. Failures on a Vertex are detected when the Job Manager stops receiving heartbeats over a given period of time. When this happens, the Vertex's computation is rescheduled on other Vertex of the cluster.

Vertexes can detect errors while processing the input; in this case they inform the Job Manager, that marks the input channel as failed and re-executes the Vertex on the origin of the input and the Vertexes that depend on that input. In the case of a Job Manager failure, the jobs under its control are marked as failed.

When compared to other workflow solutions such as MapReduce, Dryad can be considered more flexible because the developer has fine control over the communication graph and the routines that are executed on the vertexes. Also, while on MapReduce it is mandatory that all the computations take a single input and generate a single output set, while on Dryad it is possible to use and generate an arbitrary number of inputs and outputs.

This flexibility comes with the cost of being harder to program the system, but if the programmer understands the structure of the computation, the organization and properties of the system resources, it allows for better optimizations.

2.2.1.3 Taverna

Biology services require computations over large amount of data; more than 3000 molecular biology services are publicly available. However using those services require knowledge in programming that the average biology community member does not master. To help them use several of the biology services available in a faster way, *Taverna* project (Hull et al. 2006) allows the creation of workflows that uses different services without the need for expertise in programming and web-services. It provides a single point of access for multiple web-services and a user interface that facilitates the creation of workflows by integrating several of the available services visually.

2.2.2 Scripting and Querying languages for BigData Processing

2.2.2.1 MapReduce

When dealing big data processing problems at Google, the engineers *"realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately"* (Dean & Ghemawat 2008) and so, they have created MapReduce framework. The framework is based on the functional paradigm where the programmer defines map and reduce tasks that processes the data and the framework takes care of issues related with parallelization, distribution, scheduling and handling of failures. Its main advantage is the simplicity of the paradigm and the scalability of the system.

MapReduce job is defined in two main phases, the map and reduce. On the map phase, the worker takes key/value pairs as input, applies a programmer's defined function and produces key/value as intermediate output. These intermediate results are then *shuffled* by the reducers that apply a distributed merge-sort to the intermediate results, grouping them by key. For each key and respective value, the reducer applies the programmer defined function producing the final result as output.

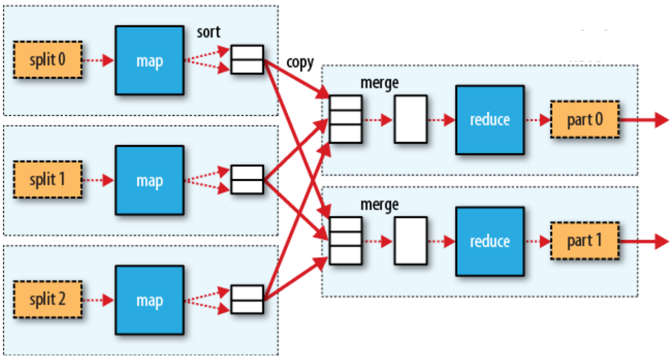


Figure 2.2: Map Reduce Execution

To execute a job, several steps are followed, as shown on Figure 2.2. In more detail:

1. The input data is converted to key/value pairs and split in chunks, normally each split contains 16 to 64 MB of input data.

2. A map Worker reads a split and for each key/value pair it applies the user defined map function, producing as intermediate result key/value pairs. These key/value pairs are maintained in a memory buffer until it reaches a given threshold, after that, the intermediate results are saved to disk.

3. For the case where the reduce function is commutative and associative, the user can define a combiner function. This function is applied by the mapper on the intermediate results produced by the map function and acts as a local reducer, merging the intermediate results with the same key but still producing key/value as output. The main advantage is to reduce the amount of data exchanged over the network between the mapper and the reducer.

4. After the map phase is completed the intermediate results are merged and sorted into a file and the master is notified.

5. The intermediate results are partitioned and assigned to the reduce workers based on their key. Normally a hashing function is used, but other user defined functions can be

applied to optimize this step. The function used must guarantee that records with the same key are assigned to the same partition.

6. The Reduce worker receives the location of the intermediate results, reads all the key/-value pairs from a given partition afterwards it merges and sorts them according to their key, grouping together the values with the same key. Note that a given partition can be spread across several map workers and the reducer must fetch this data from them.

7. For each unique key and respective values, each worker applies the reduce function defined by the user. The results produced are appended to the final output. When the task is completed the master is notified.

8. Finally, when all the reduce tasks are finished, the job is marked as complete and the user program is notified.

After the publication of (Dean & Ghemawat 2008) from *Google*, several implementations have been created and open sourced such as Hadoop MapReduce (White 2013) and Disco (Mundkur et al. 2011). The most popular of which is Hadoop MapReduce, it was created by *Yahoo!* that open sourced the project at Apache [3]. Today, the Hadoop ecosystem is composed by several systems such as Hadoop MapReduce, Hadoop File System (HDFS) (Shvachko et al. 2010), Pig (Olston et al. 2008) and HIVE (Thusoo et al. 2009).

In order to handle fault tolerance and replication of data across the cluster, the MapReduce framework works on top of a distributed file system. The existing implementations have developed different distributed file systems such as Google File System (GFS) (Ghemawat et al. 2003), Hadoop File System(HDFS) (Shvachko et al. 2010) and Disco Distributed File System (DDFS) (Mundkur et al. 2011).

Hadoop MapReduce works on top of the HDFS that handles all the distribution, replication and fault-tolerance of the data. It is designed to run on clusters of commodity hardware where the nodes have a high rate of failure. Files saved on HDFS are assumed to be very large, ranging from hundreds of MB to Terabytes. It is also assumed that the data usage pattern is to write once and after that, perform several reads in order to process a big part of the dataset. For that reason, random reads are not supported, and only appended operations that write at the

[3] http://hadoop.apache.org

end of the file are possible. Instead of trying to achieve low latency for accessing a specific piece of data, HDFS's design favours high throughput. This makes it ideal for batch processing systems. HDFS has proven that can handle large amounts of data, since the existing clusters store up to PetaBytes of data. Its main components are the NameNode and DataNodes. The first handles the files' meta-data and controls the access of files by clients, and the latter take care of the storage of the data and are deployed on each node of the cluster. Each file that belongs to HDFS is split in blocks, each block is replicated across the DataNodes to prevent data loss in case of failure. Clients can request for file operations to the NameNode that are then performed on the DataNodes responsible for those blocks.

There are two types of nodes that manage the work execution, one JobTracker and several TaskTrackers. The JobTracker ensures scheduling of new Jobs and assign tasks to the TaskTracker. Each TaskTracker is responsible for the execution of the tasks, allocating them to the workers depending on their available resources.

The failures of nodes are considered frequent and must be handled properly. In order to detect worker failures, they send regular heartbeat messages to the TaskTracker.

TaskTrackers also send regular heartbeat messages to the JobTracker. Failures are detected when a given number of heartbeat messages are not received. When a worker fails while having assigned tasks, the TaskTracker reassigns those tasks to another worker. If a TaskTracker failure is detected, then the JobTracker reschedules the failed tasks to other TaskTracker.

2.2.2.2 Disco

Disco is also an open source [4] implementation of MapReduce Framework, it was presented by *Nokia* (Mundkur et al. 2011) that has chosen to develop their own MapReduce implementation because, according to them, the Hadoop implementation is heavyweight and difficult to adapt, and so they have developed a version that is much compact and easy to evolve. Similarly to HDFS and GFS, they have also created a distributed file system named Disco Distributed File System (DDFS) specific for the use of MapReduce. The MapReduce layer contains three main components: the Master Node, the Slave Nodes and the Workers. MapReduce activities are co-ordinated by a centralized Master Node that provides client interaction, resources monitoring,

[4] http://discoproject.org

allocation of jobs and tasks. The Slave Nodes are responsible for launching and monitoring the execution of tasks and they also provide storage for DDFS. Lastly, workers are responsible for executing the Map and Reduce tasks assigned. Instead of providing a static implementation Workers, they have developed a protocol and all the workers must implement that protocol. This way the worker is language agnostic and several implementations of workers can coexist in the system.

Similarly with the Hadoop implementation, Disco is ready to handle failures of the Slave and worker node by rescheduling the tasks for other nodes but the Master Node still is a single point of failure.

The worker protocol is established between the Slave and Worker nodes. All the messages are exchanged in JSON and the communication is started by the worker when it requests for a task. The Slave assigns him a Map or Reduce task and sends him the required information about the task. Several locations for the same input file can be given because the files can be replicated. The worker then chooses the replica that is closer to him. After performing the task, the worker saves the result and communicates its location to the slave node. Inputs and results files can be assigned to tags, these tags are then used for partition of the results during the shuffle phase. A hierarchical structure of tags can be built by pointing tags to other tags.

The worker protocol is established between the Slave and Worker nodes. All the messages are exchanged in JSON [5] and the communication is started by the worker when it requests for a task. The Slave assigns him a Map or Reduce task and sends him the required information about the task. Several locations for the same input file can be given because the files can be replicated. The worker then chooses the replica that is closer to him. After performing the task, the worker saves the result and communicates its location to the slave node. Inputs and results files can be assigned to tags, these tags are then used for partition of the results during the shuffle phase. A hierarchical structure of tags can be built by pointing tags to other tags.

2.2.2.3 Pig Latin

Some problems have been noted regarding the MapReduce framework. For example, common tasks such as joins and n-stages cannot be expressed directly because the data-flow must have

[5] http://www.json.org

only one input and two-stages (map and reduce). Also, common operations such as projections and filtering have to be written by the programmer, this causes the user to write too much custom code that is hard to maintain and reuse. That was the main reason why Yahoo! engineers have decided to create a new language named *Pig Latin* (Olston et al. 2008) that fits between the SQL and MapReduce, allowing the programmers to use high-level declarative queries and, at the same time, low-level procedural programming.

The system is open source under Apache and is used on *Yahoo!* for many analytical tasks, like temporal analysis over the search logs, session analysis of web user sessions and user clicks analysis. Its main design goal is to allow programmers to perform ad-hoc analysis of large data sets in a simpler way while maintaining the advantages of MapReduce.

Programmers are able to write a sequence of steps, where each step specifies only a single high level data transformation. When submitted for execution, the *Pig Latin scripts* are compiled into MapReduce scripts that are then executed over an Hadoop cluster. By using this system, the development of data analysis tasks takes less time than developing MapReduce tasks directly and the code is easier to maintain. While the MapReduce functions are opaque and difficult to optimize, in Pig Latin scripts there are opportunities for optimizations since the complete data-flow is known in advance and the operations are more semantic.

In order to ease the process of dealing with unstructured data, the system provides support for four kinds of data representations: (1) Atoms that contain a simple atomic value such as a string or a number; (2) Tuples that associate several values; (3) Bags are collection of tuples; and (4) Maps that associate an item with a key and provide lookup operations by the key.

The system supports operations for loading, processing and writing big data sets. Each of these steps can be customized by using User Defined Functions (UDFs). Some common processing operations, such as grouping, filtering and joining are also provided but can be customized by the user.

The system executes the data-flow by compiling it into MapReduce jobs and then executing it on top of an Hadoop cluster. Before the compilation, several optimizations are applied.

By acting on top of Hadoop, the system automatically gains some properties for data and operations execution, such as, parallelism, load-balancing, scheduling and fault-tolerance without needing to build a complete new infrastructure. However being built on top of Hadoop also implies some overheads because the data must be saved and replicated into the HDFS between jobs and then read again.

2.2.2.4 Hive

Another similar system that aims at simplifying the creation of MapReduce workflows, is *Hive* (Thusoo et al. 2009). It is built on top of Hadoop and works as a data-warehousing infrastructure where the programmer write SQL-like queries that the system automatically translates and executes as MapReduce Jobs. It is a popular open source project [6], receiving several contributions and optimizations from the community and used on the industry.

Hive, has a declarative, SQL-like language but at the same time, the user can specify map and reduce operations. This gives some flexibility for the user but is not as flexible as *Pig Latin* that allows the creation of user defined functions in all steps of the data-flow. Being more similar to SQL, it is easier to learn for developers familiar with SQL.

An important component of Hive is the meta-data server that keeps track of the location of tables, the type of the columns and the serialization/deserialization routines. By having a meta-data server it has fast access to meta-data information that can be used to optimize the scheduling and data-placement decisions

2.2.2.5 SCOPE

MapReduce paradigm can be applied to solve a number of problems, such as counting the frequency of an item, a distributed sort, query the entire dataset, etc. However, not all the problems are simple to solve using this paradigm. For example, querying a dataset with random queries for analytics or doing data mining operations. At the same time, developers are used to SQL-like languages and have to translate their logic to Map and Reduce functions if they want to benefit from what the MapReduce framework offers.

Microsoft developed SCOPE (Chaiken et al. 2008) when they were trying to join the best of two worlds: Parallel databases and MapReduce. SCOPE provides a scripting language similar to SQL but it is also possible to add use User defined functions written in C#. Those scripts

[6] http://hive.apache.org

are represented as optimized DAGs and executed by the system. As the scripting language is similar to SQL, it is easy for users experienced with SQL to learn how to work with the system. The ability for users to write C# code makes it easier to express data processing operations or data manipulations.

SCOPE allows third-party libraries and operators to be defined. Extensions for SQL-like and MapReduce-like are included. Another important feature of the scripting language is a *Sample* command that uses sampling techniques over the dataset to provide faster answers with a given degree of certainty. Developers can use it to get approximate results and it allows them to do more experiments on data. Similar feature has been presented in BlinkDB (Agarwal et al. 2013) that uses stratified sampling on top of MapReduce framework to provide approximate results.

The system has native support for structured data like relational database tables and unstructured streams like user click logs.

Data can be partitioned across the data-center using Hash Partitioning or Range Partitioning. For hash partitioning a hash function is applied on the partition columns generating a number that is used as identifier for the partition, like in MapReduce. When using Range Partition, the dataset is divided by the row number, splinting the data in continuous rows. The system optimizes the partitions automatically by using sampling and calculating distributed histograms. Each partition is sorted locally using a B-Tree index that enables the system to execute fast lookup by key but it is also useful for selecting a portion of the table.

Similarly to Disco, it is also possible to associate a label to each data unit that is then used as an hint for optimizing data locality. The system tries to save data belonging to the same label group as close as possible because it is likely that they will be processed together.

To accomplish this, it first tries to save them on the same machine, if the machine is full it tries to save data on a rack close to the machine based on the network topology. This technique causes the data that is used for the same computations to be located closer together and so the network traffic is reduced and the performance is increased.

Before the query being executed it is compiled to an Abstract Syntax Tree (AST) and then, several optimizations are performed, creating an execution plan. An execution plan is represented as a DAG where the vertex's can be executed on different machines that perform some computation defined on the plan. One of the most important goals of the optimization step is to reduce the number of partition operations as it reduces the number of operations that transport data over the network.

2.2.3 Scheduling

On MapReduce framework, each worker has a configurable number of slots for map and reduce tasks and each slot is capable of running a task at a time. After the creation of a new job, the map tasks are created. The number of map tasks is equal to the number of initial splits, but the number of reduce tasks is specified by the user. These tasks must then be scheduled for execution on the free slots depending on the available resources and scheduler decisions.

Scheduling jobs across the available slots on the cluster is an important task and various approaches have been developed (Rao & Reddy 2011). Several factors can be considered on the development of schedulers such as fairness, the average execution time, resource usage and efficiency. A scheduler is considered fair when all jobs receive a fair amount of execution time. It can also favor the complete usage of the resources on the cluster or try to minimize the average execution time. Next we present some of the available scheduling solutions for MapReduce jobs.

2.2.3.1 FIFO Scheduler

FIFO Scheduler orders the jobs in a FIFO order. When the resources are available, the JobTracker picks the job with highest priority or, in the case where all the jobs have the same priority, it picks the first job that was submitted. Hadoop also tries to choose the job whose data is closest to the free node. This approach is not fair for all jobs, as a job with lower priority may starve for resources if other jobs with higher priority have been submitted. Jobs with high priority may also wait longer than they should because they can stay waiting for a long running job with smaller priority to finish. When several users are submitting different jobs for execution this solution also leads to an unfair scheduling among users.

2.2.3.2 Fair Scheduler

Fair Scheduler (Zaharia et al. 2009) was been introduced by *Facebook*. Its main goal is to multiplex the cluster resources giving every user's job a fair share of the cluster over time. Small jobs should run quickly and long jobs should not starve for resources.

Jobs are grouped on pools and each user is associated with a pool. By default each pool has access to an equal number of slots but it is also possible to configure a minimum number of slots for each pool. The available slots for one pool are split among its jobs and capacity that is not being used by one pool is shared among the jobs waiting on other pools

To guarantee that jobs don't "starve for resources", the system supports preemption. That means that if a pool has not received the fair amount of slots, then it terminates tasks on pools that are running over capacity. To minimize the lost of computation already performed, it chooses to terminate the task that started running more recently because they probably produced less progress.

When a slot becomes available and there are jobs waiting to be assigned, then this slot must be assigned to a pool and then to a job. The assignment tries to favor the jobs that are on deficit, being because the pool is under its minimum share or because the job has the highest deficit factor.

In order to improve data locality it makes use of *Delay Scheduling* (Zaharia et al. 2010). This means that when the first chosen job can not execute locally on the free slot, then it is skipped and waits a bit more for the local machine to free a new slot. To avoid starvation of those tasks, the system keeps track of skipped jobs and when they have waited more than a given time, they are allowed to run non-locally.

With this scheduler each job will receive roughly an equal amount of resources. Shorter jobs will receive the enough resources to finish quickly and, at the same time, the longer jobs still get execution time and are not starved for resources.

2.2.3.3 Automatic Resource Inference And Allocation (ARIA)

ARIA (Verma et al. 2011) is a MapReduce scheduler, that allows the jobs to have Service Level Objectives (SLOs) defined by the developers. For example, they can specify the time at which the job should be completed. The system automatically infers the resources that the job needs to use in order to achieve its SLOs. To make this possible, the system extracts some metrics from the information available on Hadoop's execution log. Using those metrics and a model introduced by the authors for estimation of upper and lower bounds of the job's duration, the system is able to estimate the minimum number of map and reduce slots that it needs to allocate for a job so that it completes within the given deadline.

When a slot becomes free, the scheduler picks the task that has the closest deadline and submits it for execution. If the amount of free slots available is inferior to the required minimum for a job, then the job is allocated with a fraction of the needed resources and the allocation function is recomputed during the job execution, adjusting the allocated resources as necessary.

A similar approach has been taken by the authors of FLEX (Wolf et al. 2010), that reuses the infrastructure from the Fair Scheduler to ensure fairness but, at the same time, allows the cluster administrator to define job optimizations based on the response-time, throughput and other metrics. The system keeps track of such metrics for tasks running on the cluster and assumes that they are uniform. Using the statistical information gathered, they optimize the tasks execution on the cluster according to the user's selected metrics.

2.2.4 Stream Processing

Batch processing systems consider that the data used is static and perform offline queries over the dataset. However, some applications have requirements for processing unbound quantities of data as it arrives at the system and provide results in real time. For example, for online enterprises whose business' is based on advertisement, maximizing the number of users that click on ads is crucial for their business success.

Algorithms have been developed to maximize the probability of a given user clicking on an advertisement by taking into account the user context. However, processing this information must be done on an online basis while receiving thousands of requests per second.

Some of the existing systems implemented stream processing by partitioning input data into fixed-size segments that are then processed by using a batch processing system like Hadoop MapReduce. Note that, the latency is proportional to the length of the segment but the overhead of segmentation and jobs processing must also be taken into account. Reducing the segment size makes more complex to manage the dependencies between segments and the overheads increase.

In stream processing the paradigm is to have a stream of events that flow into the system at a given rate that is not under the system's control and the system should be capable process those events with low response-time. We next present some systems that fit within this paradigm.

2.2.4.1 Simple Scalable Streaming System (S4)

S4 (Neumeyer et al. 2010) is a stream processing system that provides a way for programmers to easily implement applications that process unbounded streams of data. It supports massively concurrent applications while exposing a simple programming interface to the developers. Its design was directed for large-scale applications with data mining, machine learning and real- time processing requirements.

The system architecture is similar to the actors model, where the processing units of the system communicate only through the use of messages and do not share their state. In this way, the system can be highly scalable.

S4's main components are the Processing Elements, the Processing Nodes and the Communication layer. Computations are performed on the Processing Elements, they communicate through the use of asynchronous event emission and consumption. Each event has several parameters associated such as a class, a key and a value. Processing elements know the type of event that they are able to consume, having the possibility to be more specific or more generic depending on the developers' choice. The routing of events to the appropriate receiver is handled by the framework and the creation of processing units is lazy, meaning that a processing element is created only when the first message is to be routed to it.

Processing done at each element is specified by the programmer that, in a way similar to MapReduce, needs to specify a processEvent() and output() functions. For common tasks like aggregate, joins and counts, the system already provides default implementations that can be used. Processing elements keep everything in memory and can flush from time to time if needed.

When the system gets to a point where it does not have enough resources to handle everything in memory it starts killing Processing elements. This way the system is able to continue handling new requests while degrading gracefully. Note that when a process element is killed, all the state that was in memory is lost.

Processing Nodes are the logical hosts of several processing elements, being responsible for listening to events, executing operations on the incoming events, emitting output events and dispatching events to the communication layer. When an event arises, the Processing Node delivers it to the appropriate processing element in the appropriate order.

The communication layer is responsible for providing cluster management, automatic failover and the mapping between physical nodes and processing nodes. This way the event emitters do not need to be aware of the physical resources and it is easier to relocate elements in the presence of failures. This layer is build on top of Zookeper [7] that already provides functionality for managing the cluster and communication between physical nodes.

At present time the system does not have a dynamic load balancing nor live migration of Processing Elements but plans exist for developing this features in the future.

2.2.4.2 Storm

Storm [8] is a distributed system for real-time computation. It was originally conceived by Twitter to analyze the tweets' streams in real- time. Workflows are represented as topologies composed by a network of reading units (spouts) and processing units (bolts). Several topologies can be run in parallel and each processing unit can also be replicated.

The cluster is composed by two types of nodes, the Workers and the Nimbus. Nimbus acts as coordinator and is responsible for distributing tasks across the cluster and monitoring for failures. As in *S4*, tasks can be assigned in a more general or specific way.

Each worker may run a reading unit or a processing unit. Reading units are the source of the stream in a computation. They can fetch data from an external source and produce a stream as output. Processing units read data from the stream, apply some user defined processing over the data and then produce an output stream that can be read by other worker. They can also apply filter, aggregate or join operations over the data stream.

[7] http://zookeeper.apache.org

[8] http://storm.incubator.apache.org

Fault tolerance of nodes is handled by the system and message delivery guarantees are provided even in the presence of failures. Storm communication is also build on top of *Zookeper* that provides cluster management, communication between the nodes and strict message order guarantees.

Classification

This section has detailed some of the state-of-art solutions for processing large amounts of data. There are a vast number of systems that address this problem, we have introduced the main workflow and stream processing solutions and the different techniques used. The existing solutions have been built with the goal of being executed on clusters or data-centers, making difficult for external people to contribute with their computing resources to a given project. Next section will introduce some solutions that try to address this issue.

2.3 Comunity Clouds

As more and more people have computers and the available computing power increases, a great amount of computing resources are available at peoples' homes. At the same time, many computational problems introduced by applications such as image or video encoding or scientific research, still require a great amount of computer resources.

Some initiatives have been presented that make use of idle computing cycles in order to leverage the completion of such tasks. With this approach, the cost of performing those projects is much lower as the resources are normally donated for the project and, at the same time, the ecological footprint is also reduced as it makes use of existing computing resources that were being under used.

This approach also introduces new challenges. As the participants willing to offer resources are normally behind Network Address Translators (NATs) or firewalls, the machines are frequently turned off or disconnected from the Internet and the resources are highly diverse among the clients. Convincing enough users to participate and donate resources is also a difficult task and incentive mechanisms must be provided.

We next present some of the existing approaches in terms of systems that enable public-computing. We start by presenting some of the more used systems for cycle-sharing, then we describe an implementation of a MapReduce framework for cycle-sharing that works on browsers and finally we present some systems that make use of the trust relations among users of social networks.

2.3.1 SETI@home

Search for Extraterrestrial Intelligence is a scientific area whose goal is to detect intelligent life outside Earth. SETI@Home (Anderson et al. 2002) project analyzes radio signals in search for extraterrestrial signals using spare-cycles donated by people interested in the field. By using those resources it is possible to analyze more radio signals in a cost-effective manner.

In order to not disturb the user, the client's software can be configured to only start working when the computer is idle or to be continuously executing with low priority.

A server maintains all the information about radio waves that must be processed and assigns those units to clients requesting for work. A client software gets work units from the server, processes the set of signals assigned, returns the result to the server and then gets more work units. The server also keeps track of all the work units assigned and results submitted.

When all the work units are assigned, the server starts assigning redundant tasks so that it can handle results submitted from malicious users or faulty processors. When redundant tasks are assigned, the server must decide which is the correct result. SETI@home does this choice by using consensus between the results submitted.

The client's software only communicates with the server for requesting new work units and submit their results and the communication is done using HTTP. As firewalls normally allow HTTP traffic, this means that clients can communicate with the server even in the presence of firewalls.

Results from computing the signals are kept in memory and flushed to disk periodically. In case of the host machine being turned off, the client program can continue the computation from the last checkpoint.

SETI@Home project has evolved and is now build on top of BOINC.

2.3.2 BOINC

BOINC (Anderson 2004) is an open source project [9] that evolved from SETI@home and it provides the software infrastructure needed for the creation of cycle-sharing projects. It offers flexible and scalable mechanisms for distributing data, scheduling tasks and tolerating faults.

Users that want to contribute for projects with their spare-cycles need to install a client software. In order to provide incentives for users to donate resources, each user receives credits based on the resources (CPU, memory and bandwidth) provided to the project and those credits are then presented to the user, they can also build a public profile and organize into teams. It is also simple for the project's programmer to configure the application so that it provides screen-savers in exchange for client's resources.

BOINC client has a local scheduler that makes the decision of when it should get more work units in a way that maximizes the resources usage and satisfies the results deadlines.

Each project is associated with one or more applications and a set of input data. A server maintains a relational database with all the information needed for the project, such as the several application versions, input data, results and assigned tasks. Server software is divided into daemons with different functionalities that can run on separate machines or be replicated. This way the server is able to scale and tolerate some faults.

Work units are assigned to clients and each work unit has links for the input files, soft deadline for completion and minimum requirements in terms of memory, CPU and storage. The application is then executed on the client and the results are submitted to the server.

Applications are normally distributed to the clients as an executable program that is then run by the client software. In order to support clients with different platforms, each application can have several versions (one for each supported platform) and the client downloads and executes the correct version. It is also possible to provide the source-code and compilation script that will then be compiled to the client's specific platform.

In order to support failures from the clients, the system assigns redundant computation. Results are then chosen by a consensus function defined by the user. If a consensus is not obtained, the system assigns more redundant tasks until a consensus is obtained or a maximum

[9] http://boinc.berkley.edu

number of results is reached.

2.3.3 Ginger

BOINC eases the creation of new cycle-sharing projects, however there is the need to build a new community for each specific project. Ginger (Veiga et al. 2007) creates an infrastructure for cycle-sharing projects that allows users to receive donate spare-cycles to execute applications and, in exchange, receive spare-cycles from other users to execute other applications. As clients can also submit work, the same infrastructure can support several projects simultaneously.

The clients are organized in a P2P infrastructure and communication is performed directly between them. This way there is no need for install and maintain a centralized server.

Each work unit is represented as a "gridlet" that associates the operations with a chunk of data to be processed. The input is submitted by the user and is partitioned into small chunks (between 64KB and 256KB), so that it can be transferred between clients and executed with low overhead. Gridlets are then distributed across a P2P overlay network formed by the clients. The clients receive, execute and return the result to the user that submitted the work.

Gridlets are also associated with a cost represented as a vector that represents the number of CPU cycles and bandwidth needed to execute it. When a client submits a new gridlet, it estimates its cost by executing and monitoring some of the gridlets locally. In the same way, they also estimate the capacity for each client according to its availability and processing power along a given period of time, and uses this information for task assignment.

Clients maintain in cache the code and input data for the gridlets executed. When a new job is submitted that has the same code or data, the system tries to assign it to nodes that have executed it previously. In order to prevent security issues caused by the execution of applications submitted by other users, the operations code are not executed natively. If the operations is distributed as byte-code, then it is executed in the context of a virtual machine with sandbox for that language. For native code operations, they are executed in the context of a general purpose virtual machine.

2.3.4 JSMapReduce

On the solutions presented previously the users wanting to donate resources have to install a client application and the project programmers have to deal with issues related with parallelization of data processing across the several clients.

JSMapReduce (Langhans et al. 2013) introduces a solution capable of executing MapReduce tasks on the browser using the users' spare-cycles. Users wanting to donate resources just have to open their browser on a given page. As it implements a MapReduce framework, the programmers have a simplified model for the distribution and parallelization of jobs.

The Map and Reduce tasks are written in Javascript and executed on the clients' browser. The system makes use of some technologies provided by the browser such as WebWorkers [10] for processing several tasks and XHTMLRequest [11] for asynchronous communication with the server.

The system maintains a centralized server that keeps track of MapReduce Jobs, assigns tasks to workers and supervises its execution, keeping track of each worker's availability. Failures from workers are detected using timeouts. The server is also responsible for saving the intermediate results from map tasks, sorting the intermediate results and saving final results from reduce tasks.

When scheduling tasks, the system tries to assign chunks with bigger sizes for workers that have better availability. Redundant tasks can also be assigned to ensure quality of service.

Authors of the system refer that this kind of system can be applied to several types of site, but they are most useful on those where users spend more time per screen, as it provides more execution time for tasks.

MRJS (Ryza & Wall 2010) is solution very similar to JSMapReduce, it is also a MapReduce framework that executes tasks in Javascript on the client's browser. A centralized server is maintained that keeps track of the intermediate and final results and is responsible for sorting the intermediate results before creating reduce tasks. When a worker requests for a new task, the scheduler pick the tasks that have the smaller number of workers assigned at the moment.

[10] http://www.w3.org/TR/workers/

[11] http://www.w3.org/TR/XMLHttpRequest/

In case all the tasks have been assigned once, the system assigns redundant tasks. Results are chosen by a majority vote.

Both authors have created prototypes for their solutions and came to the conclusion that the server was their bottleneck and that the system can be better used for tasks with small input data and large computations.

2.3.5 Social Cloud

For the solutions before presented to work, there is a needed to create a new community willing to share its resources. However, forming a new community is a difficult task. For that reason, Social Cloud (Chard et al. 2010) makes use of the existing trust relations between users of the social networks, enabling them to trade resources on an online market.

User can share storage space in exchange for credits, that can then be used for buying other resources on the market. This is a similar approach to the credits system we have introduced in *Ginger* but, instead of having a fixed cost, the resources are exchanged in an online market.

A SLA is established between the provider of the resources and the buyer, and the system checks if the agreement has been fulfilled by monitoring the provider's resources and its availability.

2.3.6 Trans-Social Networks for Distributed Processing

Trans-SocialDP (Apolónia et al. 2012) makes use of existing communication mechanisms provided by OSNs and the communities formed by their users in order to perform resource discovery and improve cycle-sharing systems.

The system uses communication mechanisms provided by the social networks such as posts, comments and messages, in order to advertise resources and to match them with other users' searching for resources. By using the existing communities for resource discovery, it is easier to find other persons interested in contributing and there is no need to create a new infrastructure as they reuse the mechanisms provided by the OSNs.

The creation and aggregation of tasks is done using Ginger(2.3.3) and a software is run on the client's side that analyzes the resources availability, the CPU idle time and the user's status on the social network. That information is then used by a scheduler that decides when it should start executing tasks without disrupting the users' normal activities.

Classification

System	Communication	Scheduling	Faulty Results	Runtime
BOINC	Client-Server	Maximizes Resource Usage	Redundancy	Native Execution
Ginger	P2P	Data Locality and Execution Costs	-	VM
JSMapReduce	Client-Server	Clients Availability	Redundancy	Javascript
MRJS	Client-Server	Smaller Replication Factor	Redundancy	Javascript
Social Cloud	P2P	-	-	Native Execution
TransSocialDP	P2P	Data Locality and Execution Costs	-	VM

Table 2.2: Community Clouds Classification

These sections presented some solutions for the problem of sharing computing resources. Table 2.2 presents a summary of their main characteristics. We have identified how these solutions manage the communication between participating nodes, their scheduling approaches and the techniques used to handle faulty results. Most of the solutions presented have a high entry cost because they require participants to install custom software. Other solutions have overcome that issue and proved that is possible to have cycle sharing solution that run on regular Web Browsers. However, such solutions have not been able to do it in a scalable way because they have used a client-server communications that caused a bottleneck at the server.

3 Architecture

In this chapter we present our solution for building a distributed platform capable of executing MapReduce jobs on regular Web Browsers. First of all, we present an high level view of the proposed solution (Section 3.1). In Section 3.2 we explain the main interactions between the distributed components. Section 3.3 explains the protocols during the distributed component's interactions. Then, on Section 3.4 we explain our scheduling algorithm. Finally, Section 3.5 explains the mechanisms used to deal with failures of distributed nodes.

3.1 System Architecture Overview

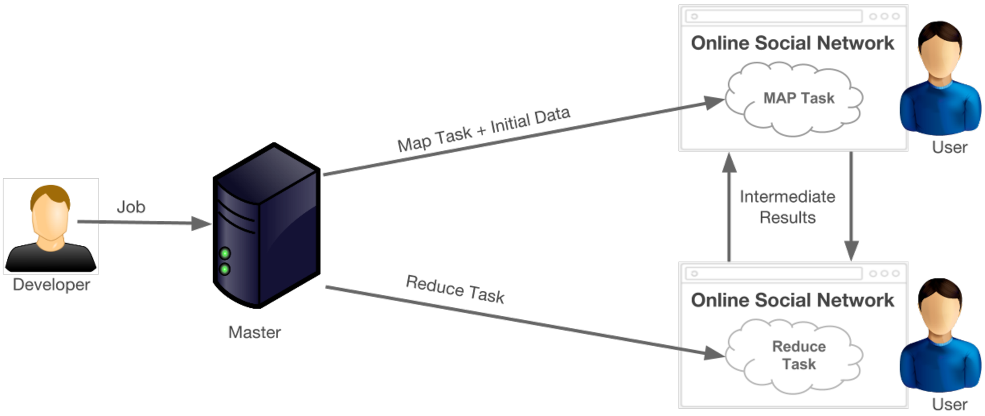


Figure 3.1: Architecture Overview

Social-Networking@Edge is a MapReduce framework, that runs tasks at the browser of OSN's users. A normal use case of the system can be seen at Figure 3.1. Users access the OSN's Web Site as usual and while they do that, a client software is executed in the background of the Web Browser. Developers are able to submit MapReduce Jobs to the Social-Networking@Edge server that are then executed by the available users. A job's specification contains several parameter such as the map and reduce code to be executed and the number of mappers and reducers that should be used to execute the job.

The server will schedule Map or Reduce tasks to the available users' browsers taking into consideration the resources on each machine. In order to improve the scalability of the system, user nodes can also exchange intermediate results of Map tasks with other nodes, enabling data processing to continue without overloading the server. The system ensures the execution of the jobs submitted, taking care of common problems of distributed applications such as parallelization, load-distribution and fault-tolerance. Building such a system brings several architectural challenges that will be addressed on the following sections.

3.2 Distributed Architecture

The solution presented here enables the distribution of MapReduce tasks among a large number of nodes. As we increase the number of nodes, or the capacity of each node, the system is able to improve its performance by raising the level of parallelization.

The system has two types of nodes: one master node and multiple user nodes. The master node's main responsibilities are: a) to keep record of the jobs' specifications submitted by developers; b) to know the availability of the user nodes at a given point in time and their computing resources; c) to distribute MapReduce tasks among the available user nodes according to the job's specification; and d) to save the results of the MapReduce jobs, so that the developers can access them after completion. User nodes are responsible for the execution of MapReduce tasks assigned by the master node.

Figure 3.2 presents a view of the main distributed components that execute on the system and the communication channels used by them. Each user node has a Worker Manager component and several Workers. The Worker Manager creates Workers that are responsible for the communications with the master node and with other user nodes. The number of Workers that is created on each user node depends on the available resources of the user's host machine and the level of participation with which the user is willing to contribute to the system. Workers are responsible for executing the tasks assigned by the master node and delivering the results when the execution ends. Multiple Workers can run in parallel and communicate with the Worker Manager by means of messages exchanged using a bidirectional message channel. The Worker Manager may send and receive information from the server or from other user nodes. Each message is identified with the sender and receiver so that the Worker Managers can dispatch

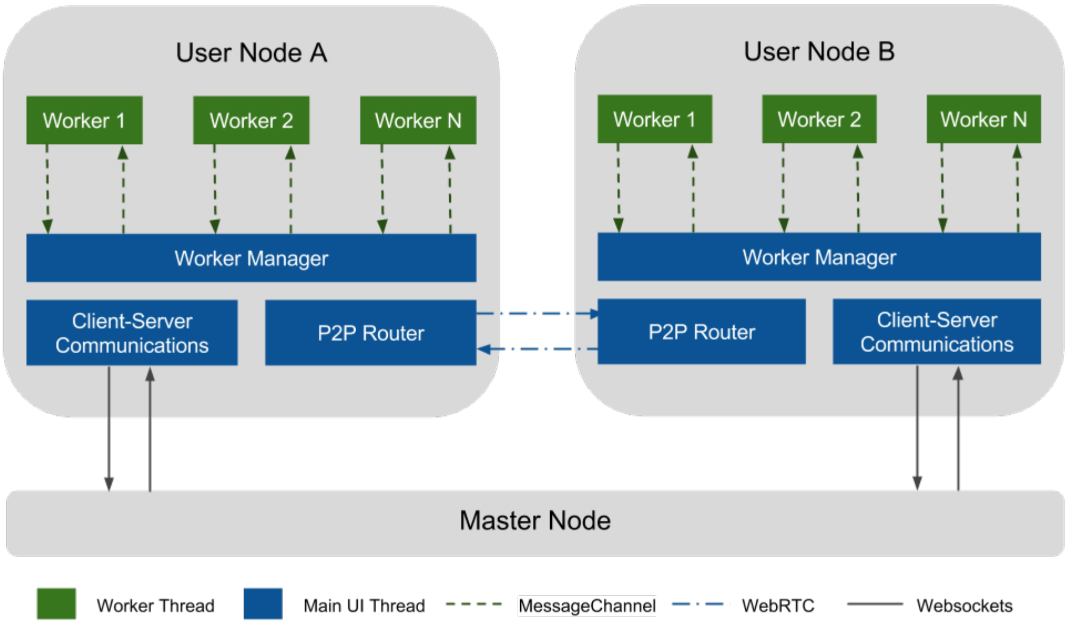


Figure 3.2: Client's Running Components

the messages to the appropriate workers.

Communications between the master and user nodes are done through a Socket based communication that is kept alive while the user is using the system. By using a socket based communication the master node can take the initiative of sending messages to the available users. This is an important property as it allows the server to push tasks to the available users without waiting for the users nodes to request them. The connection with the server is managed at the level of the Worker Manager, meaning that when a user node has a large number of workers, there will be only one connection with the server. This ensures that the number of connections that must be managed by the master node is kept at low levels even when a large number of workers is using the system.

Workers are organized on a P2P network where each worker is a peer and the connections are maintained at the level of the Worker Manager. Similar to the previous case, there must be only one connection between user nodes, even when there are several workers on the same user node. By having a P2P router at this level, the system can also optimize the case where messages are exchanged between workers running on the same user node because they can communicate directly without using the network.

All the messages created by workers identify the sender and receiver before being sent.

The Worker Manager intercepts the messages and delivers them using the appropriate channel. Each worker is identified by a unique pair (UserID, WorkerID), that is assigned by the master node when a user connects. The creation of P2P connections is lazy, meaning that they are only created when the first message between the two nodes is exchanged. This ensures that there are only connections between user nodes that need to communicate.

3.3 Protocols

3.3.1 Communication Protocol

In order to provide a way for the running nodes of the system to interact, there must be a protocol for the messages exchanged during a job's execution. Figure 3.3 shows the interactions between the running nodes of the system and the communication protocol used during the execution of a job. For this example we use a job with 2 mappers, 2 reducers and a replication factor of 1.

When a user node connects to the system, it creates a given number of Workers that depends on the participation level agreed by the user. Workers start their execution by connecting to the master node with a login message. The master node assigns them a unique identifier and marks them as available for executing tasks.

At the time a job's description is created by a developer, the scheduler creates the required map and reduce tasks and assigns them to the available users. When a map task is assigned to a worker, the master node sends the information needed to execute the task such as the split, map and combine code and also the URL from where they should download the input chunk of the task. As a result from that information, workers start loading the specific task's code and downloading the input chunk from the server. After that, they compute the tasks' result by sequentially applying the split, map and combine code to the input chunk.

After all entries of the input's chunk being computed, workers signal the master that the map task is complete. The master node waits for the completion of all map tasks and then asks the scheduler to assign reduce tasks. Mappers need to know which are the reduce workers, so that the map results can be partitioned accordingly. For that reason, the system must assign all the reduce tasks before starting the shuffle phase.

Once all reduce tasks are assigned, the master node sends that information to the mappers and they can start the execution of the shuffle phase. At this stage, workers partition the results of the map phase among the reduce workers of the job. The partition of each entry can be calculated by applying an hash function to the map result's ($reduceWorker(mapResult) = hash(mapResult.key) \mod |reduceWorkers|$). Using this technique, map results with the same key are assigned to the same reducer and the entries are evenly partitioned among the available reducers. After partitioning the map results, each partition is sent to the corresponding reduce worker.

Data transfer performed during the shuffle phase is done through P2P channels established between the workers participating on that phase. Each reduce worker waits to receive data from all mappers and then signals the server that it has completed the shuffle phase. Note that reduce workers have no prior knowledge about which mappers have entries partitioned to them. By that reason, they have to wait to receive messages from all mappers. When the partition for a reducer is empty, the map workers send an empty shuffle message. These extra empty messages allow the shuffle process to be handled by the distributed nodes, using the master node only to signal its completion.

When all workers have completed the shuffle phase, the server sends information required to execute the reduce phase. The reduce tasks' description contains specific code for the reduce function that must be loaded. After loading the reduce code, each worker merges and sorts the received entries and then apply the reduce function to all the entries. Finally, when the task is complete, workers send its result to the master node so that it can be saved persistently and later made available to the developer that submitted the job.

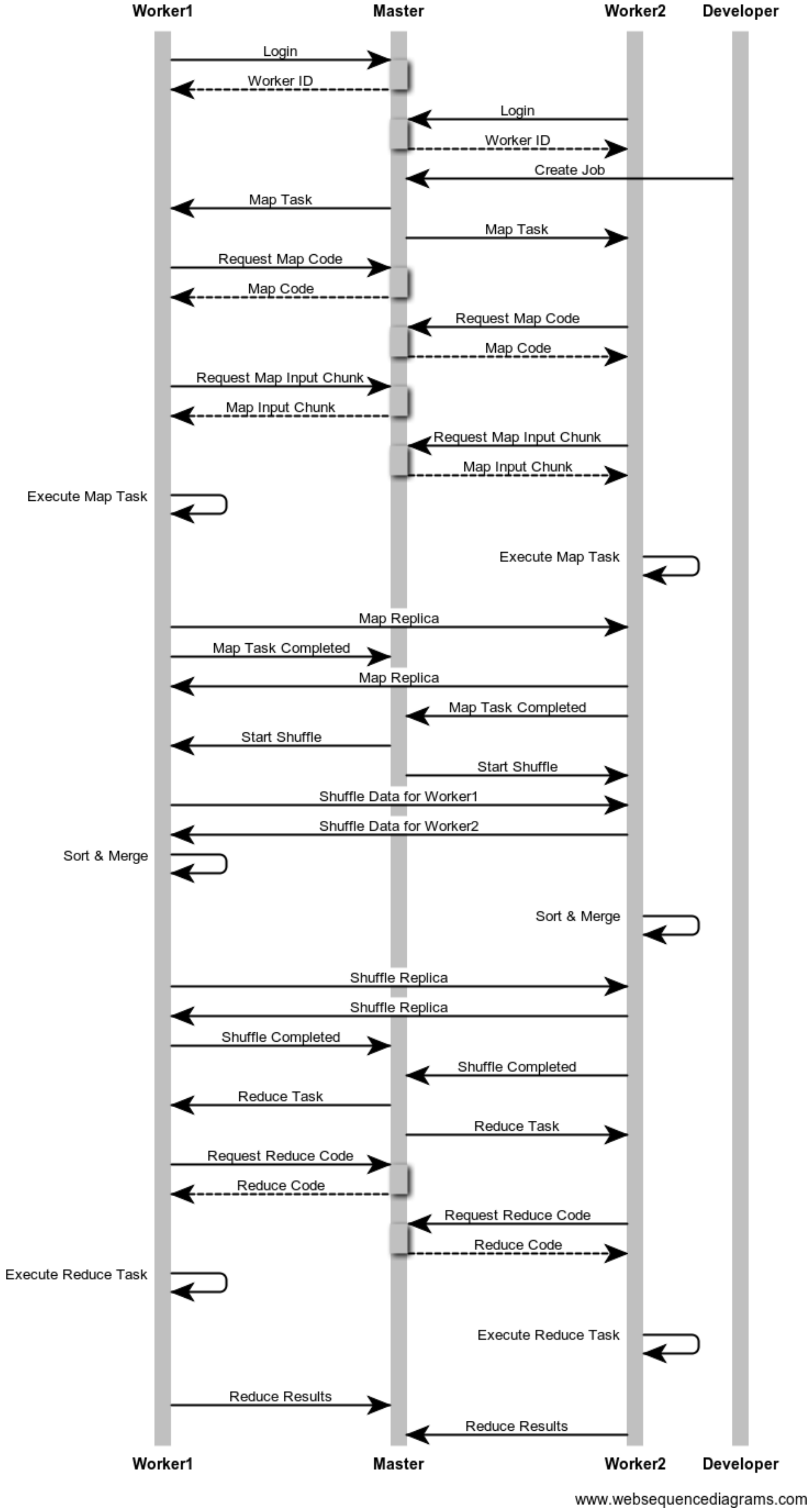


Figure 3.3: Communications during Job Execution

3.3.2 Messages' Protocol

During the execution of MapReduce jobs, several interactions must take place between the running nodes of the system. For these communications to take place, multiple messages must be exchanged between them. We next describe the types of messages used and the information carried by them.

- **Login** messages are sent by user nodes to register their availability. **Work Request** messages have the same goal but are sent by workers after their creation and signal their availability for starting executing MapReduce tasks.
- **MapTask**, and **ReduceTasks** are sent by the server and contain the information needed to start the execution of tasks. It contains the URL from which mappers can download their input chunk and the code that must be executed. In case of failure recovery, these messages also contain the ids of the replicas for that task.
- **Start Shuffle** is sent by the master node when all mappers have completed their execution. This message is sent to all mappers and contains the ids of the workers that will participate in the reduce phase. When a worker is recovering from a failure that occurred during the shuffle phase, this message also contains the ids of workers storing replicas of the map phase's checkpoint.
- **Shuffle Data** messages are exchanged between workers during the shuffle phase and contain the map results partitioned to a given worker.
- **Map Task Completed** is sent by workers to the master node and signal the completion of a map task.
- **Shuffle Completed** messages have a similar goal to **Map Task Completed** in that they signal the completion of shuffle phase. Mappers send this message after sending all the map results to the reducers. Reduce workers send this message after receiving **Shuffle Data** results from all mappers.
- **Reduce Task Result** is sent by reducers after completing the assigned task. This message identifies the task being completed and contains the results of that task.

- **Task Replica** messages are sent by a worker to its replicas. The message contains a checkpoint with the results of a given task's execution. For example, after the execution of a map task, the worker sends a replica of its results to other nodes. The master node decides which are the replicas and ensures that a worker's replica is always located at on a different user node.
- **Replica Read Request** are used to pull checkpoints from replica workers during fault recovery. Replica workers then reply with a **Replica Response** message that carries the requested data.
- **Ping and Pong** messages are used to check if user nodes are alive. The master node regularly pings the user nodes that then reply with a **Pong** message.

3.4 Scheduling

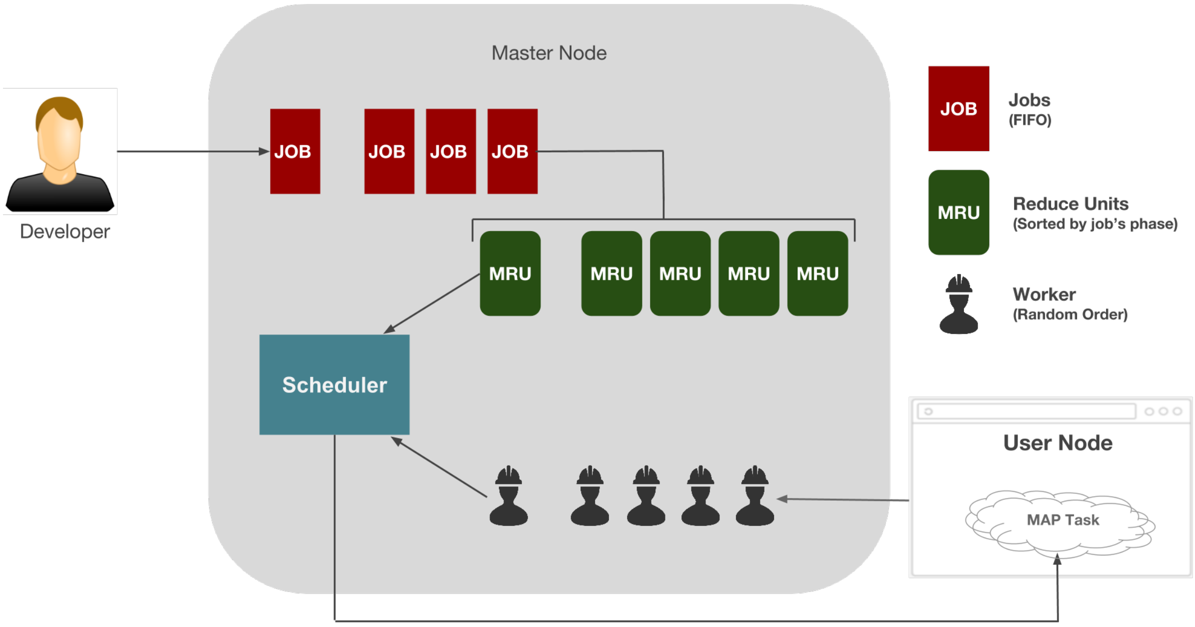


Figure 3.4: Scheduling Process

Our solution for executing MapReduce jobs on Web Browsers supports a large number of users. Each user may also contribute with several workers. Choosing to which worker, each task should be assigned has impact on the system's performance, load-distribution and resources' usage.

Figure 3.4 shows the scheduling process used in our solution. When a developer creates a job, it is added for execution. Jobs are executed according to their creation order and can be stored using a FIFO data structure. Each job is split into several working units. The specific number of working units used depends on the number of mappers and reducers of the job. Each working unit is a map or reduce task that can be executed by a worker. Tasks must be executed in order depending on their type (map or reduce). For that reason, they are stored on a queue sorted by the job phase in which the task should be executed. Whenever a user becomes available, it connects to the master node and creates several workers. Each of those workers is marked as free for execution and added to a queue that is randomly sorted.

Algorithm 1: Scheduling Algorithm

```
input: a FIFO $jobsFifo$ with all jobs to be executed
       and a set $availableWorkers$ with all the available workers
  loop
    $job \leftarrow BlockingDequeue(jobsFIFO)$
    $workUnitsQueue \leftarrow Sort(WorkUnits(job))$
    $workersQueue \leftarrow Shuffle(availableWorkers)$
    while $isRunning(job)$ do
      $workUnit \leftarrow BlockingDequeue(workUnitsQueue)$
      $worker \leftarrow BlockingDequeue(workersQueue)$
      $Assign(workUnit, worker)$
    end while
  end loop
```

The scheduling algorithm can be resumed on algorithm 1. The algorithm is a loop that starts when a job is scheduled for execution. Each iteration of the scheduler removes the first element of the job's working units and assigns it to the first element of the workers' queue. Whenever one of the queues is empty, the scheduler blocks and waits for a new item to be added.

When a task is assigned to a worker, it is notified through messages that follow the protocol explained in the previous Section. Note that, as each user contributes with several workers that are randomly sorted, the algorithm ensures that the system's load can be well distributed across a large number of workers and user nodes.

The algorithm schedules jobs by submission order and there is no concurrency between them. However, one can deploy several master nodes and use a load balancer to distribute the available workers among them. By using several master nodes the system is able to schedule multiple jobs in parallel and scale to a larger number of user nodes.

3.5 Fault Tolerance

Systems that deal with a large number of distributed nodes where each node has a high rate of failure must provide strategies to tolerate some of those failures. In our case, user nodes can fail frequently and our solution must provide resilient ways for the system to detect and recover from failures. During our study, several approaches were considered.

The simplest solution would be to mark the job as failed when a given worker fails. However, with this solution, jobs would fail each time a worker failed. For long running jobs, this is not an acceptable behavior as it would require the re-execution of the entire job. Note that, even this simple solution requires the detection of failures from workers and tracking tasks' execution and workers' state while they are executing tasks.

A better solution would be to detect workers' failures and reschedule their tasks to other workers. Remember that in our system, the state and intermediate results of the tasks are saved at the user nodes. This means that when a user node fails, all jobs' progress would be lost each time a worker failed. Therefore, all failed task would have to restart from the map phase. For failures that occur during the map phase, there are no great consequences, but in case where failures occur on shuffle or reduce phases, the overheads would be significant.

Deploying replicated tasks when one task is started is another possibility. Each task would be assigned to more than one worker and then the master node would accept the results of the one that completes first. This solution normally improves the performance of the system at the cost of using more resources. For the case where a failure occurs, the master node would simply wait for the completion of a replica. By default, this solution allocates more resources than the ones needed for the jobs' completion but it is able to recover from failures in a short amount of time.

We propose a solution similar to the second presented here but with the addition of enabling the partial re-execution of tasks when a failure occurs. During the execution of jobs, workers save checkpoints of the data processed at that moment. Those checkpoints are replicated and saved by other user nodes available at the moment. When a failure occurs, the master node assigns the task to other of the available worker that can continue the task's execution from the last checkpoint. Checkpoints are made after the completion of each intermediate task and before the notification of the server. This ensures that enough replicas exist before the task being marked as complete. The number of checkpoints saved has impact on the quantity of data transfered between the workers and on the time to complete the tasks. By that reason, it is important that developers can configure the level of replication wanted for each job. The decision of which workers should be used as replicas is done by the master node that ensures that a replica is always placed on a different user node.

Figure 3.3 shows the interactions needed for ensuring proper replication. After the execution of map and shuffle tasks, the results are replicated and saved by other workers. When a failure occurs, the system detects the failure and assigns the task to a different worker. That worker receives information about which workers have saved checkpoints of the previous phase and try to restart the task's execution from that point.

User nodes' failures can occur in different occasions. The master node uses three mechanisms to detect them:

- If the network connection is lost, the socket connection between user and master nodes is broken and the system considers that the user node has failed;
- While a user node is connected, the master node keeps sending it regular ping messages that must be replied on a given amount of time. If the user node fails to reply to more than three pings, the master node considers it as failed.
- It may happen that a worker blocks during the execution of a given task. To handle this case, developers may choose a timeout for task's completion. If a worker fails to complete a work unit within that period, the task and worker are marked as failed. The value for this timeout is configurable by developers because different jobs may have distinct expected durations.

Failures may occur at every phase of a job's execution, namely during the map, shuffle or reduce phase. Each case has different properties and must be addressed on a slight different way.

Recovering from map failures consists simply on re-executing the task from the beginning because the task has never been completed.

For shuffle tasks, the recovery starts by transferring the last checkpoint from one of the available replicas. As we explained previously, during the shuffle phase, mappers push their results to the reduce workers that should receive their partitions. However, when a failure occurs, all data received is lost and so, the reduce workers must be able to pull data that belongs to their partition from other map workers. After pulling data from mappers and pushing its map entries to the appropriate reducers, the shuffle task is marked as complete and the job's execution continues normally. Temporary failures during the shuffle phase may also occur, causing shuffle messages to be lost. Reducers are able to detect such failures by a mechanism of timeouts. When a reducer has waited more than a given amount of time for receiving shuffle data from a given map worker, it sends a message to that map worker requesting data to be resent.

The procedure to recover from reduce failures is similar to the previous one in that the recovery worker receives information about where the checkpoint's replicas have been saved and tries to download the results of the shuffle phase. After getting the reduce input, the execution of the reduce task proceeds normally.

Summary

In this chapter, we explained the main challenges of building a system with the characteristics proposed previously. We approached what architecture decisions where taken, explaining its benefits and trade-offs. In the next chapter we will explain the technical details for implementing the solution detailed here.

4 Implementation

Previous chapter explained the main architectural decisions in such a way that the solution may be applied using different technologies. In this chapter we detail the main technologies and technical decisions that have been made for the implementation of a prototype that follows the architectural decisions explained previously.

4.1 Software Components

In order to create a prototype for the proposed solution we developed the software required for the execution of the master and user nodes. Here we explain how that software components are organized and how they interact.

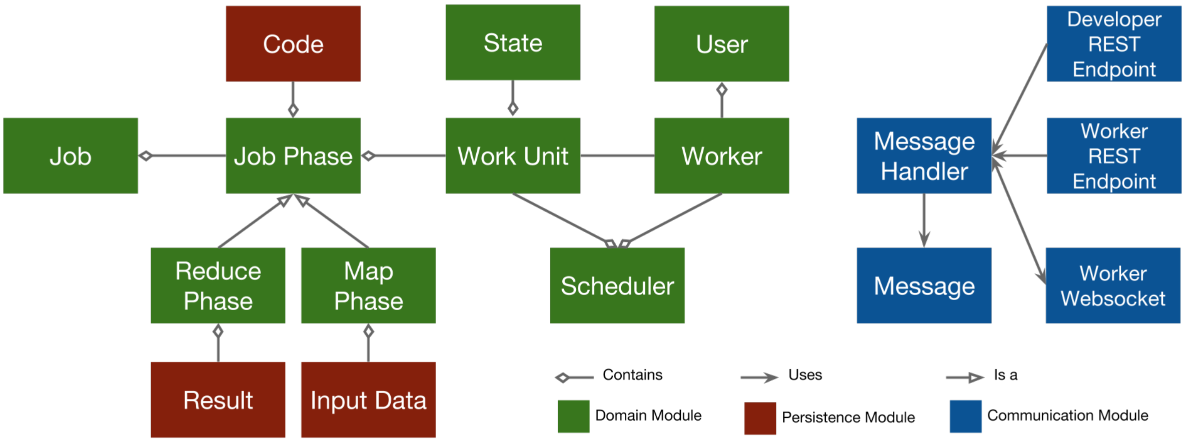


Figure 4.1: Software Components for Master Node

The software at the master node is implemented as a *Java* server and ensures that MapReduce jobs can be properly executed by keeping record of all the necessary meta-data. Figure 4.1 presents the main software components that implement the master node. The code can be divided in three main modules: One responsible for keeping meta-data and controlling the execution of jobs; other module responsible for the persistence of information and finally a module that handles communications.

Jobs can be created by developers and their specification is kept at the master node. Jobs have two phases (Map and Reduce) and each of them keeps the code that should be executed on that phase. Map phase keeps record of the input data chunks that should be downloaded and processed by the mappers. The reduce phase, keeps the location of the result files that are submitted at the end of the reduce phase. Job phases are also responsible for handling failures and notifying workers that a task has been assigned to them. Keeping that code at this level enables the system to handle failures differently depending on the job's phase where the failure occurs. Depending on the number of mappers and reducers that should be used on a job, each job phase is divided into several work units that are then scheduled for execution. As we explained previously, the scheduler keeps two queues, one for work units that must be executed and other for workers that are available for executing tasks.

In order to persist information, the system uses the file system of the master node. All the input data, results and code is saved on disk using a hierarchical structure that takes into consideration the unique id of the job and task. The job's specification and its metrics of execution are also persisted to a JSON file that can be used for analysis of the job's performance.

The communications module was implemented using *Spring-Boot* [1] and provides the necessary interactions with user nodes using a REST endpoint and a WebSocket endpoint for each worker. Using this endpoints it is possible to receive and send information to the user nodes. For each of the message described on Section 3.3 of the previous chapter there is a Data Transfer Object (DTO) encoded in JSON and a Message Handler that is responsible for treating a specific message type.

User nodes (Figure 4.2) are responsible for the the execution of tasks and by the communications that must be performed during its execution. The software at the user nodes has two different contexts: one main thread and several WebWorkers.

The Worker Manager is executed at the main thread and is responsible for the creation of workers and for managing the communications with other nodes. External communications are ensured by the usage of two data channels: a P2P data channel implemented using WebRTC (Salvatore Loreto 2010) that enables the communication with other user nodes during job's

[1] `http://projects.spring.io/spring-boot/`

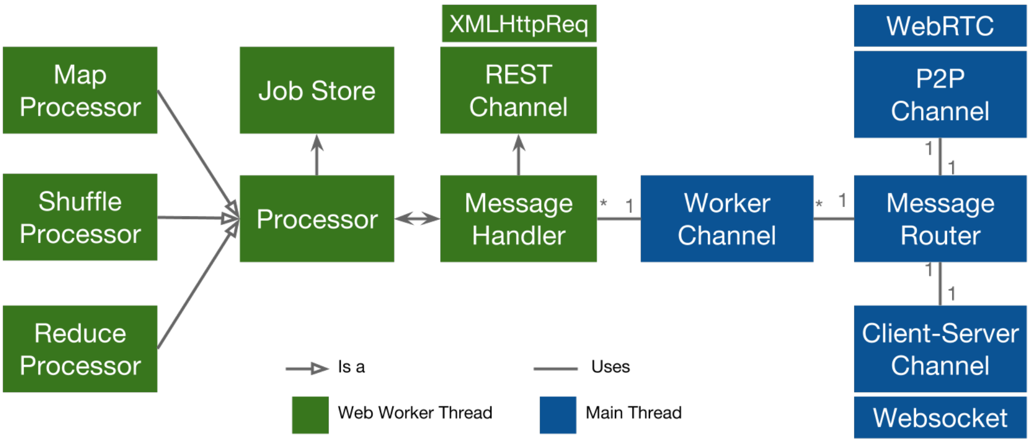


Figure 4.2: Software Components for Worker Node

execution; a bidirectional data-channel implemented using WebSockets is maintained with the master node for exchanging control messages. Communication between the worker manager and workers is done using a dedicated worker channel that is used to send or receive messages from each of the workers being run at the user node. All the messages sent or received by the user node are handled by the Message Router component that knows how to dispatch those messages to the appropriate data channels.

Each worker executes on a different WebWorker and is responsible for the execution of tasks assigned by the master node. For each job's phase (map, shuffle and reduce), there is a processor that is responsible for the execution of that phase and knows how to handle all the protocol's messages that can be received. The message handler component receives data delivered by the Message Router and dispatches them to the appropriate processor. When a message is sent by a processor, the Message Handler delivers it to the Worker Manager that then knows how to dispatch it to the appropriate channel. Workers may also exchange messages directly with the server by using XMLHttpRequests. During the execution of a job, workers must save data (checkpoints and task's results) so that it can later be used. This data is saved by a Job Store that keeps it accessible on data structures during the job's execution.

4.2 Parallel execution of Workers

Nowadays, almost all Web Browsers have support for the execution of Javascript code and it is used for building most of the existing Web Sites. However, up until recently it was not possible to do computing intensive operations on a Web Browser without blocking the user's interface nor it was possible to have P2P communications between Web Browsers. With the introduction of technologies such as WebRTC and WebWroerks, that are now supported by most of the Web Browsers, we have been able to develop a prototype with such features.

Javascript's programming paradigm is different from other languages. All the code execution is managed by an event loop. Events are added to an event's queue and then handled sequentially. Each event is associated with a function that is called once the message is to be processed. Each function is run to completion, meaning that the next event can only be handled once the previous handler has completed its execution. The good practice is that each event handler executes on a short amount of time because the program execution can only continue after it completes. For the case of Web Sites running code on its main thread, this is important because the interface becomes unresponsive which has consequences for its usage experience. This model has several advantages as it eases development but it is not a good model for building computing intensive tasks that, by nature, take a considerable amount of time to complete. To solve this problem a new technology has been introduced - WebWorkers enable the execution of computing intensive Javascript operations on background threads. Each WebWorker runs on a different context from the main thread and the communication between the main thread and the WebWorker's thread can be done by means of exchanging asynchronous messages.

For the development of our prototype we used WebWorkers for the execution of MapReduce tasks. On Figure 3.2 we can see that each computing unit runs on a WebWorker and the Worker Manager executes on the main thread. This enables our prototype to run computing intensive tasks on a Web Browser without blocking its main thread. Several WebWorkers can be executing at the same time on different threads, allowing the execution of different MapReduce tasks in parallel at the same user node. The number of Workers being executed is controlled by the Worker Manager and is given as parameter on its initialization. If no value is given, then the system creates as many workers as the number of cores on the user's host. There are some limitations on the API that can be accessed by WebWorkers. For example, it has no access to WebRTC API. Instead, all the P2P communication must then be handled by the main thread.

4.3 Execution of MapReduce tasks

At the moment of job's creation, developers can specify the function's implementation that should be run for map, combine, partition and reduce functions. They may also choose the input files that will be processed, the number of mappers and reducers. The code is stored at the master node's file system and served when requested by workers.

When a task is assigned to a worker, it receives the links to the code that should be used. It then downloads and dynamically loads those functions that are used to compute the task for which they have been assigned. In order to implement this feature we used a functionality offered by the WebWorkers specification that allows the import of scripts from a given URL served by the master node. When a task is assigned, the worker imports the scripts needed to execute it. After importing the scripts, the execution context has access to the map, combine and reduce functions that are then applied in the right order to process the job's input. As each WebWorker has a different context, each worker may import different functions with the same name without collisions, meaning that they can even execute different jobs in parallel.

4.4 P2P Communications between Workers

Workers need to communicate with each other in order to exchange data during the shuffle phase and checkpoints during the job's execution. They also need to save checkpoints at given points of the job's execution. A naive implementation would use the master node as a broker for delivering those messages. However, such implementation would create a bottleneck at the server that would limit the performance and scalability of the solution. A better alternative is to enable P2P communication where the workers communicate directly with each other. In order to implement this feature we used WebRTC. This technology enables P2P communication between Web Browsers even when the peers are behind NAT routers.

In order to initiate the communications with each other, they first need to exchanged some previous information and signal their intent to start a new connection. To allow this initial exchange of information we have used a signaling server written in NodeJS [2] with which all the participants connect in order to discover and signal other nodes. At the signaling server,

[2] http://peerjs.com/

each peer is identified by the UserID because all the user node's connections are managed by the Worker Manager. We have also used a public server for Session Traversal Utilities for NAT (STUN) (Rosenberg et al. 2008) that is used by the WebRTC protocol to discover the real public address of each node. When a worker has to send a P2P message, it first creates the content of the message and specifies a pair (UserID, WorkerId) that identifies the worker to which the message should be delivered. This message is then passed to the main thread that may connect or reuse and existing connection with the user identified by UserID. The message is sent using the WebRTC channel established between the two nodes. For the case where messages are to be sent to a worker of the same user sending the message, the Worker Manager simply delivers the message to the appropriate worker without needing to use the network.

By default, data exchanged between workers and the main thread is passed by value. This means that for all the messages exchanged, a copy is created and then delivered. For small messages this has no impact, but for larger messages the browser can block, resulting in the Web Page being terminated because the maximum buffer size of the channel is reached. For example, on *V8* [3], which is the Javascript Engine used by *Google Chrome* [4], each process has a memory limit of 512 megabytes or 1 Gigabyte depending if the system is 32-bit or 64-bit. Fortunately the protocol supports passing messages by reference when the type of the object is a native array. However, when sending messages by reference, the thread that sent that message can no longer use the original object. Our solution to overcome the current protocol's limitations was to use the default implementation for small control messages because it has a low overhead. For all the messages that contain data, such as shuffle messages or checkpoints, the worker copies the message to a native array and then sends it by value. With this solution we have a low overhead for small messages and the overhead of creating a copy of the data is on the worker side and can be done in parallel without disrupting the normal execution of main *Javascript* thread.

[3] https://code.google.com/p/v8/issues/detail?id=847

[4] https://www.google.com/chrome/

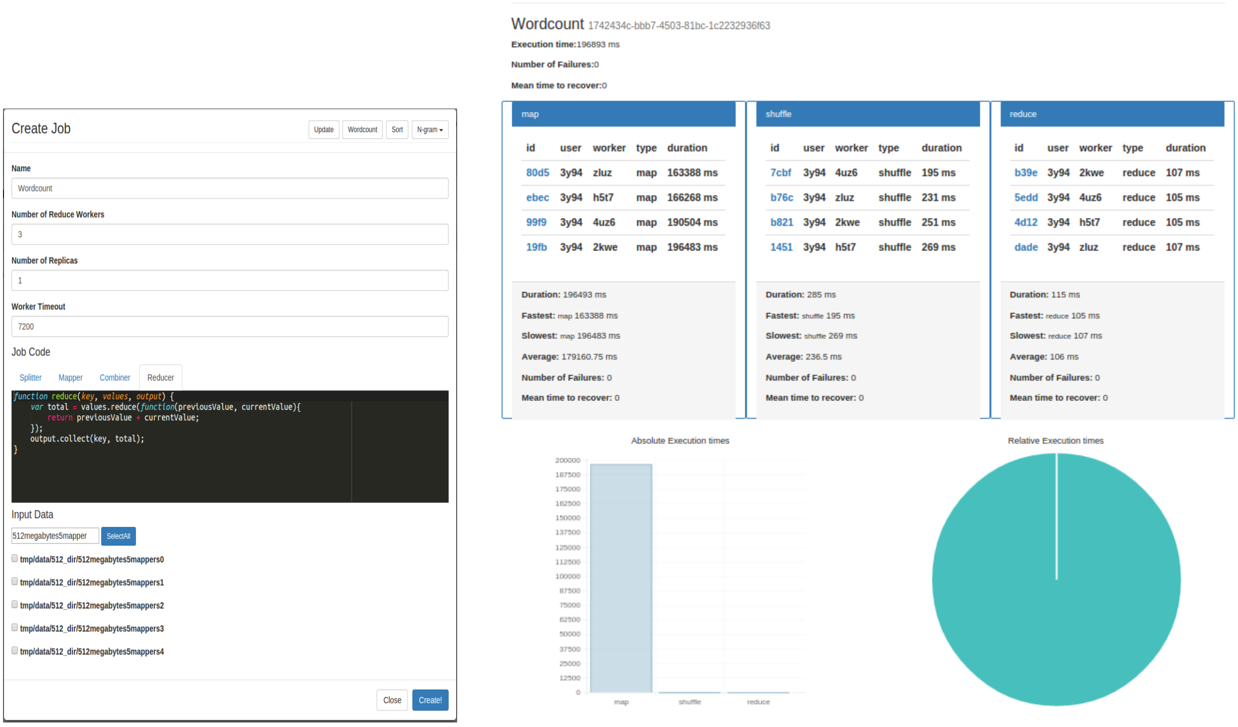
4.5 Client Sever Communications

As we explained previously and showed on Figure 3.2, communications done with the server are handled by the Worker Manager. This communication is implemented using WebSockets. Using this technology, a connection is established at the startup and then it is maintained while the user keeps on the same Web Page. For each user, only one connection is maintained which reduces the number of connections that must be handled by the server when compared to regular HTTP Requests. For small messages, the time for establishing a connection with the server is significant when compared with the time of exchanging data. By having only one connection that is maintained while the user is participating, we were able to reduce the connection overhead. Other interesting property of Web Sockets is that it allows the server to have the initiative of starting a communication. The master node uses this property to push tasks to Workers when they are assigned by the scheduler. When compared with the solution of having all the workers pulling the server for tasks using HTTP Requests, the network usage is much smaller and the server does not have the overhead of handling such messages. All the messages received by the master node are handled asynchronously by a pool of Java workers, leaving the server threads free to handle new connections.

At the beginning of the map phase of a job, workers must download their input chunk from the master node. Similarly, after the completion of reduce tasks, they must send its results to be saved on the master node. Normally, these two types of messages are much larger than the control messages and its transmission times have an high impact on the total execution time of the jobs. Using a unique communication channel between the user node and the master node has several advantages as we explained previously. However, the Worker Manager must copy all the message's content because the communication between WebWorkers and the main thread is done by value. Here we can no longer use the technique we used before of duplicating the content to a native array because it would have an even higher impact on the time that the main thread is unresponsive. In order to improve the transmission time for that type of messages, several optimizations have been applied: At the start of map phase, instead of using the normal communication channel, workers download the input chunks directly using a XMLHttpRequest; The reduce results messages are also sent directly using the same approach; Another important improvement is that the messages exchanged using this channel can be compressed with G-zip, which allowed us to significantly reduce the size of the messages exchanged with the server and the time to complete the transmission.

4.6 Graphical User Interfaces

The master node provides an external API for developers to get information about the status of the system and submit jobs for execution. We implemented this API using REST endpoints for each specific information. To simplify the usage of the system, a graphical user interface (GUI) is also provided by the master node. This feature has been implemented using AngularJS [5]. The interface shown on Figure 4.3a allows developers to submit jobs for execution after specifying the desired properties such as the number of reducers, the level of replication, the code that should be used and the input data. It is also possible to observe some information about a job's execution (Figure 4.3b) such as the time that each working unit has taken to complete and the time spent per job's phase. If the information presented here is not enough, it is also possible to download a JSON file with all the important statistics of the job's execution.



Name: Wordcount

Number of Reduce Workers: 3

Number of Replicas: 1

Worker Timeout: 7200

Job Code

```
function reduce(key, values, output) {
    var total = values.reduce(function(previousValue, currentValue){
        return previousValue + currentValue;
    });
    output.collect(key, total);
}
```

Input Data: 512megabytes5mapper

□ tmp/data/512_dir/512megabytes5mappers0
□ tmp/data/512_dir/512megabytes5mappers1
□ tmp/data/512_dir/512megabytes5mappers2
□ tmp/data/512_dir/512megabytes5mappers3
□ tmp/data/512_dir/512megabytes5mappers4

(a) GUI for job's creation

Wordcount 1742434c-bbb7-4503-81bc-1c2232936f63

Execution time:196893 ms

Number of Failures:0

Mean time to recover:0

map

id	user	worker	type	duration
80d5	3y94	zluz	map	163388 ms
ebec	3y94	h5t7	map	166268 ms
99f9	3y94	4uz6	map	190504 ms
19fb	3y94	2kwe	map	196483 ms

Duration: 196493 ms
Fastest: map 163388 ms
Slowest: map 196483 ms
Average: 179160.75 ms
Number of Failures: 0
Mean time to recover: 0

shuffle

id	user	worker	type	duration
7cbf	3y94	4uz6	shuffle	195 ms
b76c	3y94	zluz	shuffle	231 ms
b821	3y94	2kwe	shuffle	251 ms
1451	3y94	h5t7	shuffle	269 ms

Duration: 285 ms
Fastest: shuffle 195 ms
Slowest: shuffle 269 ms
Average: 236.5 ms
Number of Failures: 0
Mean time to recover: 0

reduce

id	user	worker	type	duration
b39e	3y94	2kwe	reduce	107 ms
5edd	3y94	4uz6	reduce	105 ms
4d12	3y94	h5t7	reduce	105 ms
dade	3y94	zluz	reduce	107 ms

Duration: 115 ms
Fastest: reduce 105 ms
Slowest: reduce 107 ms
Average: 106 ms
Number of Failures: 0
Mean time to recover: 0

(b) GUI for job's statistical information

Figure 4.3: Graphical User Interfaces

[5] https://angularjs.org/

Summary

This chapter explained the main technical decisions and technologies used to implement our solution. We have detailed the main challenges and decisions in terms of the distribution of computation across a large number of distributed nodes, how each node can dynamically load tasks and the communication channels used to enable client-server and P2P communications.

5 Evaluation

In this chapter we describe the methodology used to evaluate the prototype developed and discuss the obtained results. The main evaluation goal was to observe the performance of the solution when submitted to different contexts and to identify potential bottlenecks. Besides the performance analysis, we also intend to prove that the system supports general use cases of existing MapReduce applications.

5.1 Experimental Testbed

5.1.1 Benchmark Suite

We evaluated our prototype using popular benchmarks that are normally used by the Hadoop community, namely: Wordcount and N-gram. There are several public implementations of these benchmarks. The most popular are the ones from Hadoop distribution. However, to our knowledge, there are no open source Javascript implementation of this MapReduce algorithms. For this reason, we implemented a Javascript version that is based on open source implementations for other languages.

WordCount is used in order to count the number of times that a word appears in a given text. Each mapper receives a line of the input and produces a pair (Word, 1) for each word present on that line of text. The pairs produced are then sent to the reducers. All the pairs with the same Word are delivered to the same reducer. This means that the reduce function can simply output the sum of all the received values for a given key. As each Word is mapped to the output, the size of the map phase's results' is the same as the original input. However, this can be optimized by using a combiner function that executes the reduce function locally after the shuffle phase. Using this optimization, the amount of intermediate data exchanged during the shuffle phase is smaller than the original input because all the occurrences of a given word can be combined into a single entry. The input used to perform the WordCount tests was generated

Physical Characteristic	Characteristic Value
Processors Model	Intel Hex-Core CPUs
CPU speed	Between 2.0 GHz and 3.0 GHz
File system format	LVM and QCOW
Internet connection	Gigabit Ethernet

Table 5.1: Characteristics of Digital Ocean's physical machines

using Hadoop random text writer that is normally distributed with Hadoop.

N-gram is an algorithm used in computational linguistics and protein sequencing. It allow us to know the number of times that a given sequence of N words is present on a given input text. For each line of input, the map function produces the combination of N sequential words on that line. The reduce function outputs the total number of times that those sequences occurs. By that reason, during the shuffle phase, the amount of data exchanged between workers is larger than the original input. Our algorithm implementation generates and validates all possible entries. In order to test our system with this benchmark we used some open source books from *Gutenberg Project* [1] as input.

5.1.2 Testing Environment and Settings

In order to test our prototype we needed a distributed environment that allowed us to have a large number of nodes. We used DigitalOcean's [2] platform for executing such tests.

DigitalOcean is a company provides infrastructure as a service on a cloud based solution. They allow the easy deployment of virtual machines with a given hardware configuration. The physical nodes characteristics is presented on Table 5.1. They have eight data centers located on six different countries. When you deploy a virtual machine it is possible to select in which data center it will be deployed. In order to perform our tests we used three machines' configurations, presented on Table 5.2.

As presented on chapter 3, the system has two types of nodes, namely: master nodes and user nodes.

During all tests, we used the same configuration for the master node and it corresponds to the C configuration presented on Table 5.2. In order to deploy a master node we needed to

[1] http://www.gutenberg.org/
[2] https://www.digitalocean.com/

Configuration	Memory	CPU (Number of Cores)	Disk Size
A	2 Gigabytes	2	40 Gigabytes
B	4 Gigabytes	2	60 Gigabytes
C	8 Gigabytes	4	80 Gigabytes

Table 5.2: Virtual Machine's configurations

install two running components:

1. a Tomcat server running the Java code responsible for the master logic explained previously;
2. and a *Nodejs* server that acts as a signaling server for WebRTC protocol.

For the user nodes, the tests performed used different configurations in which regards the virtual machines capabilities, number of machines used and geographical distribution. The specific configuration used is described on each of the tests presented below. Each user node contributed with a number of workers equal to the number of cores available on the virtual machine where it was deployed. The deployment of a user node consists on opening a Web browser (in this case we used Google Chrome [3]) on a URL being served by the master server. As the tests were running on virtual machines with no visual environment and the Web browser required one, we used Xvfb [4]. Xvfb is an X server that performs all the rendering operations in memory without requiring any display output.

[3] http://www.google.com/chrome/

[4] http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml

5.2 Base System Performance

In this section we present the results obtained when executing the benchmarks referred previously. For each test we start by presenting what were its main goals, then we present the settings in which the test was performed and lastly we present and discuss the results obtained.

5.2.1 System performance for an increasing number of participants

Goal One of the initial goals for the system was that it should be able to increase its overall performance simply by adding more resources. This means that when the number of available workers increases, the system should be able to use their computing resources and, by doing that, the time needed to complete jobs should decrease.

In order for the system to use its available resources, the master node must be able to detect their availability and the scheduler should be able to assign tasks to the available workers.

Settings In order to assess the performance of the system for an increasing amount of resources, we tested the system with WordCount benchmark with a fixed input size of 1 gigabyte and an increasing number of participants. The virtual machines' configuration used for all the user nodes was the type A from Table 5.2. This means that each of the user nodes has two cores and so they contributed with two workers.

We started by having 10 machines and 20 workers and then we increased that number until we reached a total of 50 machines and 100 workers. The jobs executed used 1 reducer and 1 replica. The number of mappers was increased at the same rate at which the number of available workers increased.

Discussion Increasing the number of mappers proportionally with the number of available workers has several predictable effects. The most important of which is that the input chunk that each of the mappers has to download from the server is smaller in size. By doing that we were expecting that the time taken by each mapper to download its input chunk would decrease. Also, the number of lines that each of the nodes must process is decreased, meaning that the processing done by each mapper is also reduced.

The reducer node must receive information from all of the mappers. This means that it should be able to keep as many connections as the number of available mappers. On the other side, the amount of data received by each mapper decreases and they can be exchanged in parallel.

Increasing the number of participants has also effects on the master node. Mainly because it must handle a greater number of concurrent connections and requests. Also, the control it must perform over all the workers and running tasks increases.

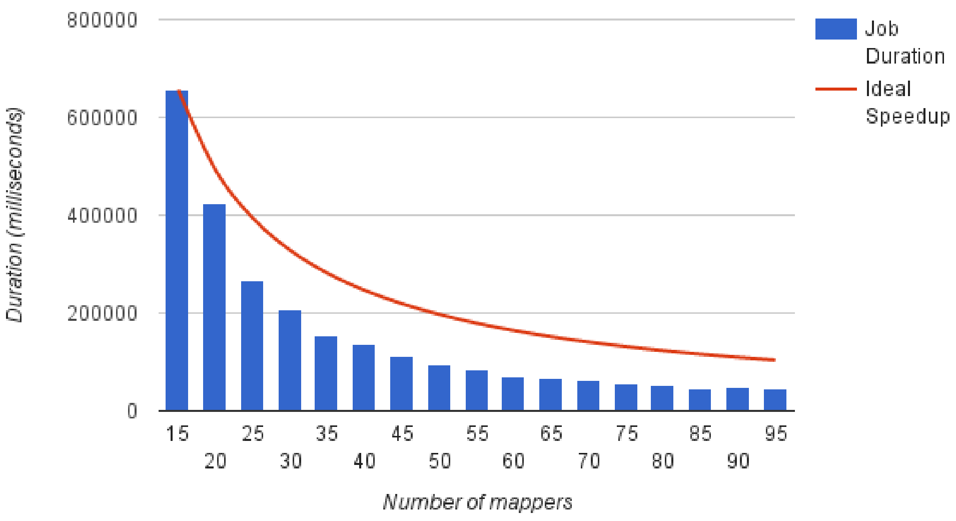


Figure 5.1: Job duration for increasing number of workers

Figure 5.1 presents the execution times of WordCount jobs while increasing the number of available workers and mappers and how it compares with the ideal speedup relative to the first execution.

As expected, the job's execution time decreases as the number of available workers increases. The rate at which the execution time decreases is lower than the ideal speedup. Still, this means that the system is able to scale vertically as the number of workers increases.

Notice that as the number of mappers gets closer to the maximum number of available workers, the relative time improvements start decreasing. There are two main reasons for this to happen. The first is that the overheads of communication between the workers become larger due to the larger number of P2P connections that must be handled by reducers. The second reason is that at the beginning the scheduler can distribute the system load across a large number of idle workers. However, when we get to a number of mappers that is equal to the number of idle workers, all the user nodes have to support more load, meaning that the resources' usage on those nodes is increased. In Section 5.5 we analyze the resources usage with more detail.

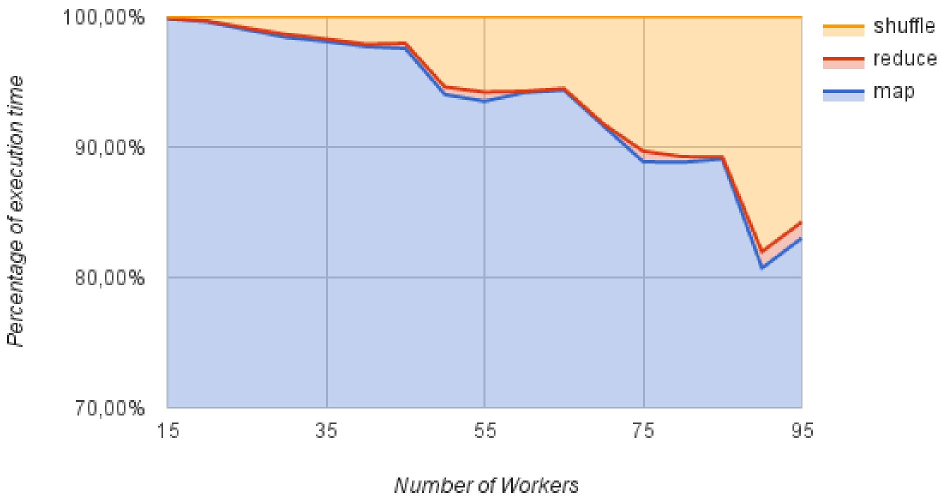


Figure 5.2: WordCount for increasing number of workers - percentage of time per phase

Figure 5.2 presents the percentage of time spent on each job's phases as the number of available workers increases. We would expect that as we increase the number of participants, the relative amount of time spent in the map phase would decrease and the time spent in the shuffle phase would increase. This should happen because the overheads of downloading data from the server are decreased but the number of connections between workers increases. The relative amount of time spent in the reduce phase remains approximately constant because the computations done during the reduce phase are kept the same.

With this test we have proven that the system is able to cope with an increasing number of user nodes. When that happens, the scheduler distributes the system load across the available workers providing an acceptable level of scalability.

5.2.2 System performance for an increasing input size

Goal The main goal of this test is to evaluate the system's performance and its capabilities as we increase the size of the job's input.

Settings To perform these evaluations we used a fixed number of workers and increased the input size. The benchmark used was WordCount with 20 mappers, 5 reducers and replication factor of 1. In order to run the test we deployed 20 virtual machines of type B (Table 5.2), with a total of 40 available workers. We run several tests, starting by using an input of 256 megabytes and progressively increased it by 128 megabytes until reaching 1 gigabyte.

Discussion Increasing the input size of the job has the reversed effect of the previous test. As the initial input size increases, the computational requirements of each participant also increase. The input chunk that each worker has do download and process is larger and the amount of data exchanged between the participants increases.

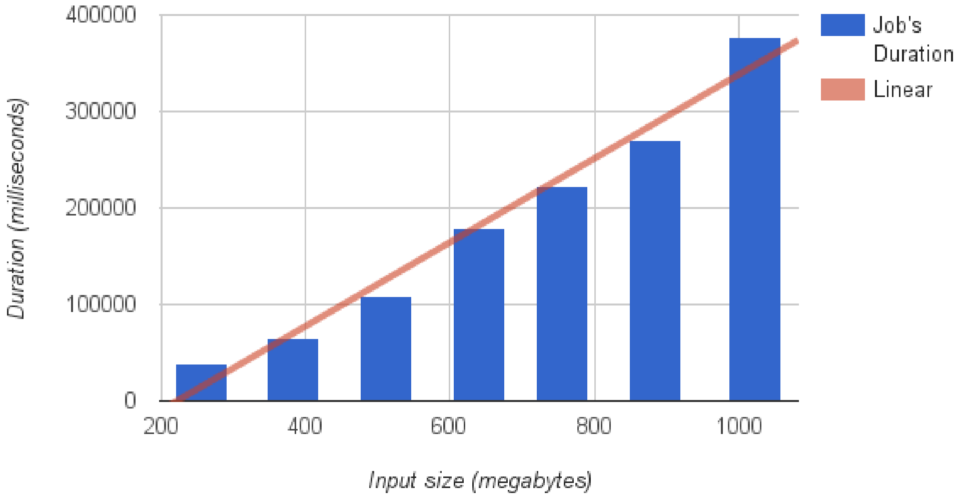


Figure 5.3: Wordcount job duration for increasing input size

Figure 5.3 shows the jobs' duration as we increase the size of the input file. As can be seen, the jobs' execution times increase linearly with the input size.

Note that when for larger inputs the time to complete the jobs is above the linear level. This happens because we approached the maximum usage of the participants' resources. Currently the system has some limitations on the quantity of data that can be processed. These limitations are related with two design decisions:

1. All data is kept in memory while the workers are processing a given job. This means for the initial input chunk, checkpoints' replicas received from other workers, the results

of processing and sorting data, are all saved on the worker's main memory during the job's execution. When we get to the point where the level of occupied memory exceeds the browser's capacity, the system starts degrading in terms of performance. Google Chrome has a memory limit of 1 Gigabyte per process for 64-bit systems. When that level is reached, the browser crashes and the user node fails.

2. For a given job, a worker can only process one map task. This means that the only way of processing larger files is to increase the number of workers that participate on that job.

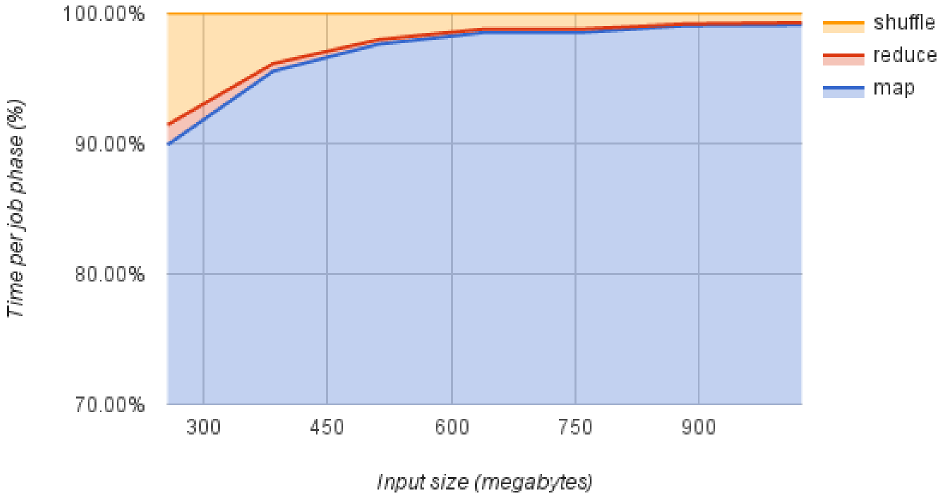


Figure 5.4: Relative job's phases duration with increasing input size when executing Word-Count

Figure 5.4 shows the time needed to complete each job phase when we increase the input size of those jobs. From that image it is easy to understand that the job's execution time is dominated by the time needed to complete the map phase. At the same time, the time needed to complete the map phase is proportional with the input size. This proves that the major overhead of the map phase is the initial download of the file chunk. We tried to reduce this overhead by using compression. In order to further improve the initial download time it is possible to use other techniques such as pre-fetching (more on that in next chapter).

With this test we have evaluated how the systems' performance changes when the input size of a given job increases. The system supports such cases and scales linearly, being limited by the available memory of the available participants.

5.3 System Performance with Increasing Complexity

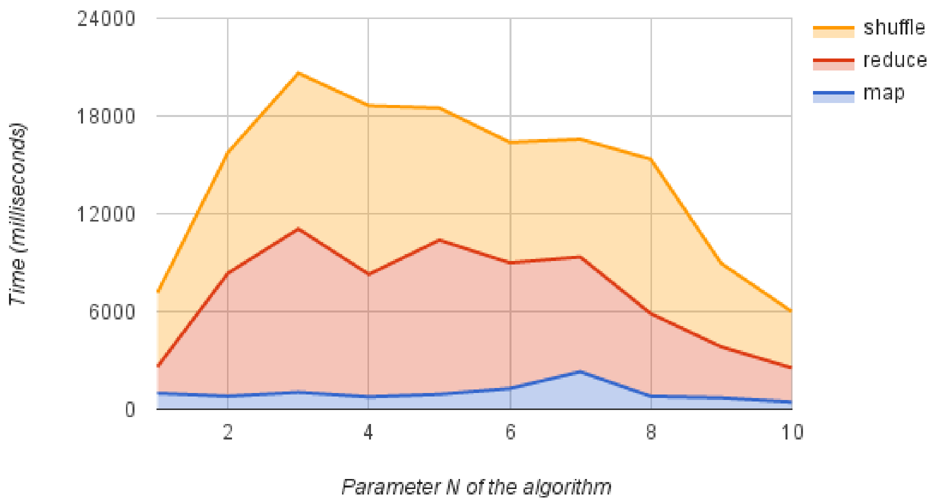


Figure 5.5: Time per phase of N-Gram jobs as the parameter N of the algorithm increases

Goal One interesting use case of the system is the execution of jobs on which the processing done by each node increases and the quantity of data generated by the nodes changes. With this test we evaluate how the system behaves while running N-Gram benchmark with an increasing value of N.

Settings In order to evaluate how the system behaves as we increase the problem complexity, we tested the system using N-Gram benchmark. The job used a fixed number of available workers and input while the N parameter of the algorithm varied between 1 and 10. For all the executions of the job it used 30 mappers and one reducer with a replication factor of 1. All the machines used were of type B (Table 5.2). For the test execution we considered that an N-gram is relevant if it is generated at least once. By doing this we ensured that the results contain the count of all combinations of N sequential words that occur on the document.

Discussion As we introduced in Section 5.1.1, the N-Gram algorithm generates the N sequential combinations possible for each line of the input. For a fixed input, when the value of N increases, the number of combinations that is generated by the algorithm decreases but the size of each combination increases.

Figure 5.6 presents how the size of the final results changes as we increase the N parameter of the algorithm. The input size of the jobs is also presented and was stable for all the executions of these tests. We can see that the algorithm starts by having a compression effect over the original dataset because all the words that occur multiple times are reduced to a single word of the result. From 2-Gram's until 5-Gram's jobs, we see that the result size increases. We see this increase of the result's size because the number of distinct entries generated during the map phase also increase. Remember that the reducer must receive all those entries from the mappers so that it can count the number of occurrences of each combination of N words.

The input file has an average of 8 words per line. Lines that have less than N words are automatically filtered by the map function of the algorithm and will not be present on the result. Also, the number of combinations generated during the map phase is related to the number of words on each line. These are the reasons why we see a relative decrease of the result's size after the 5-Gram until 10-Gram.

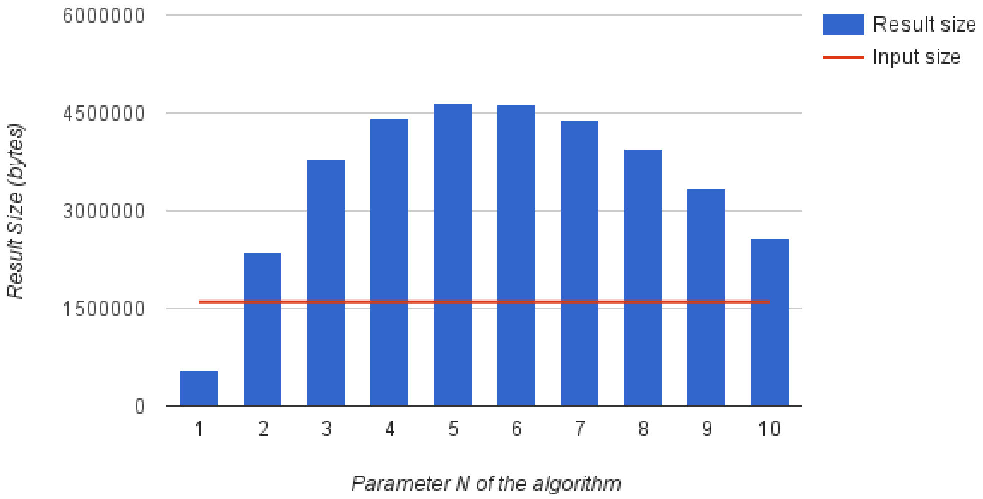


Figure 5.6: Size of the results of N-Gram jobs as the parameter N of the algorithm increases

It is interesting to compare the results size (Figure 5.6) and the the execution time per job's phases presented on Figure 5.5. Looking at the cumulative execution time of the job's phases, we get the execution time of the jobs. Note that the job durations are directly affected by the results' size. This happens because the job's duration is dominated by the shuffle and reduce phases. The shuffle phase has a large impact on the execution time of the job because the reducers must receive data from all mappers and this is a costly operation. The system was able to support a large number of connections on the reduce workers, that received a considerable amount of data. In order to further reduce the overheads of communication between the nodes we have tried to use a compression algorithm. However the stability of such algorithms' implementation in Javascript is far from ideal and had an impact on the result's reliability. The implementation of such algorithms is out of the scope of our study, nevertheless they should be considered for future work.

5.4 System performance with geo-distribution

Goal We initially proposed that our system would use the user's donated resources while they keep their Web Browser open on a specific URL served by the system. The users that donate their resources can be geographically dispersed. This use case is specially important if the system is used on an existing Web Site with a large number of users that can be geographically distant from each other. With this test we want to evaluate how the system reacts to the overheads introduced while we increase the level of geographic distribution of the participants that donate their resources.

Settings This test was run using WordCount as benchmark with an input file of 1 gigabyte. The job used 30 mappers, 5 reducers and a replication factor of 1. During the test we used 20 virtual machines of type B (Table 5.2).

	Number of machines per Data center			
Distribution level	**Amsterdam**	**Frankfurt**	**London**	**San Francisco**
Test case 1	20	0	0	0
Test case 2	10	10	0	0
Test case 3	7	7	6	0
Test case 4	5	5	5	5

Table 5.3: Nodes configuration for increasing geographic distribution tests

In order to simulate the geographic distribution of the participants, we deployed the user nodes on different data centers, located in different countries. Initially all the nodes were located in the same data center and then we increased the number of data centers, distributing the 20 nodes evenly across the used data centers. The specific distribution of machines is described in detail on Table 5.3. For all the test's executions the master node was always deployed on Amsterdam's data center.

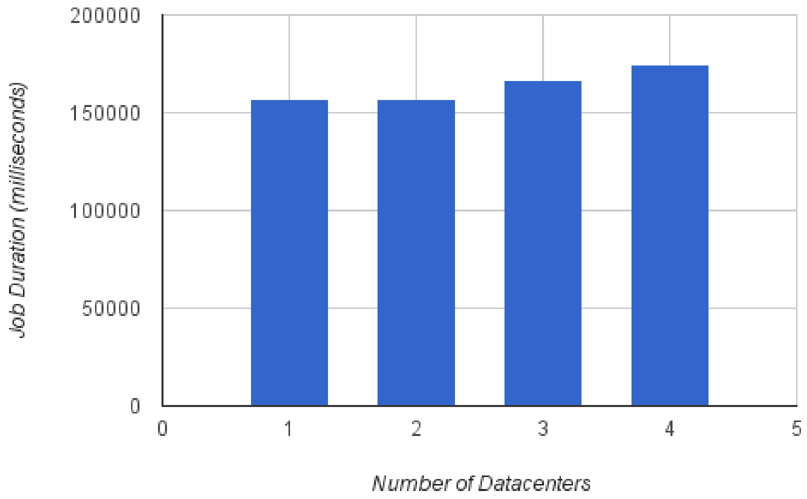


Figure 5.7: Jobs duration for increasing distribution level for 1 gigabyte WordCount

Discussion As explained previously, in this test we increased the distribution level of the participants by deploying them on data centers located in different countries. On our solution, nodes need to communicate during job's execution so that can exchange intermediate data. As we increase the distance between those nodes, the network latency increases which has impacts on the transfer time.

Figure 5.7 shows the time needed to complete jobs as we increase the level of distribution between the participants. As expected, the job's duration gets bigger as the geographic distance between the distributed nodes increases. However, the rate at which the overheads increase is not significant and is not noticeable when we use two data centers.

5.5 Resource Usage and Scheduling

Goal The usage of resources by the user and master nodes is an important topic for the evaluation of our solution. Remember that the initial goal was that our solution should allow developers to use the power of MapReduce computations while reducing the necessity of using large data centers. In order to achieve this goal, the architecture was designed so that only the master node must be maintained by the developers and the user nodes only have to access a given URL to donate their resources. Besides that, the master node should also require very limited resources when compared with current used solutions.

User nodes donate a given part of their resources for the execution of MapReduce tasks on their Web Browser. The usage of resources on the participants is decided by the scheduler component. With this test we want to detect how well the available resources were used and the main overheads.

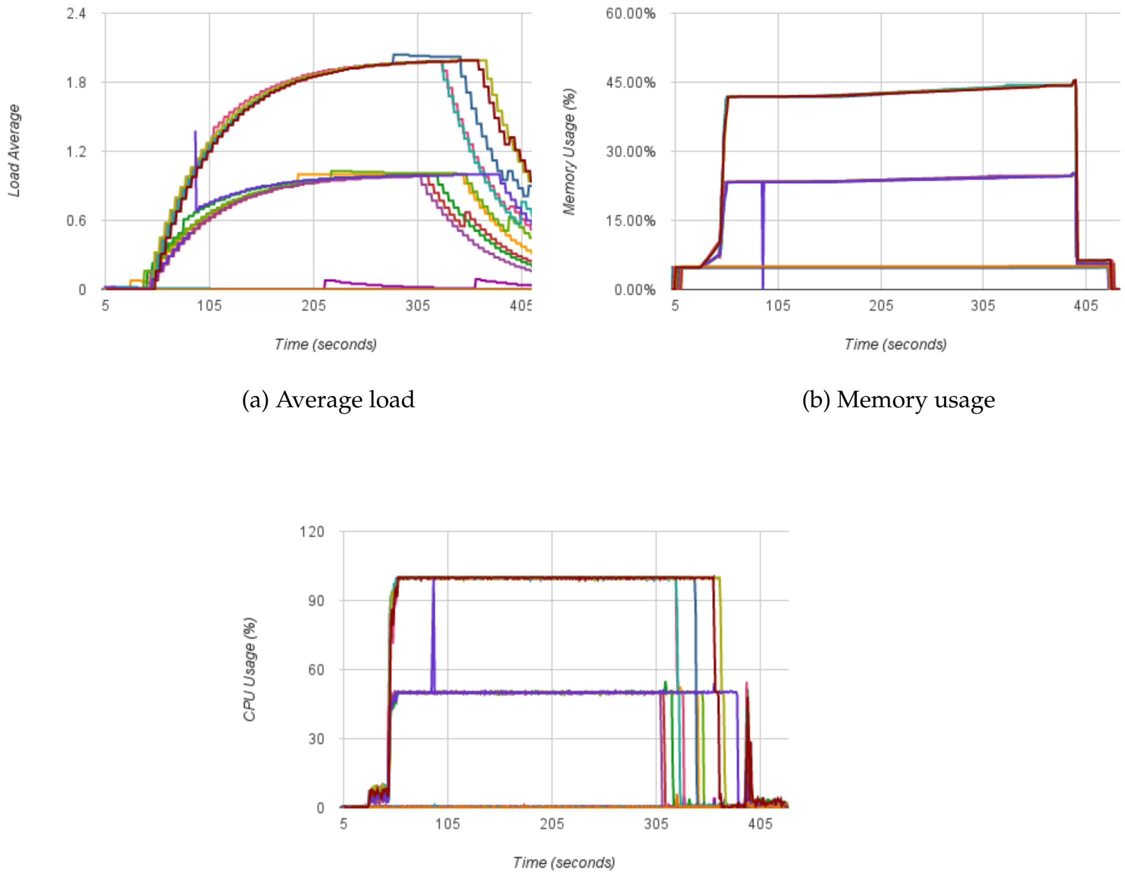
Settings For this test we executed WordCount benchmark with 20 participants of type B (Table 5.2) that originate the participation of 40 workers. The job used 30 mappers, 5 reducers and replication factor of 1. During the job execution we used *dstat* [5] to scan the resources usage in terms of memory, CPU and network on the master and user nodes.

Discussion Figures 5.8 and 5.9 present the resources' usage of the user and master nodes during their participation on this test respectively. On the first figures, each line represents the resources usage of one of the available nodes.

Starting by the user nodes, we can see in Figure 5.8a that the load increases exponentially after the start of the job and then stabilizes at approximately 90% for the majority of nodes and at 190% (because each machine's CPU has two cores) for a minority of nodes. Also, for a small number of nodes we can see that the average load keeps very close to zero percent. The reason why this happens has to do with the level of resources donated by each participant and the scheduling decisions. In this test, each participant has two workers, that is why some of the nodes get an higher load than others. The scheduler distributes the job's working units by the available workers randomly. The users that donate more workers have an higher probability of getting more tasks assigned to them. In this case we can see that the scheduler does an acceptable distribution of load across the available resources and only a small number of nodes have no assigned tasks to perform.

On Figure 5.8b we can see that the system uses 20% of memory when they have only one working unit assigned and approximately 45% when all the worker slots are assigned to the users. It is easy to conclude that the memory usage is higher than expected if we consider that the total job input has one gigabyte and it was evenly distributed by 20 workers. The high usage of memory has to do with the design decision of keeping all the data in memory during the

[5] http://dag.wiee.rs/home-made/dstat/



(a) Average load

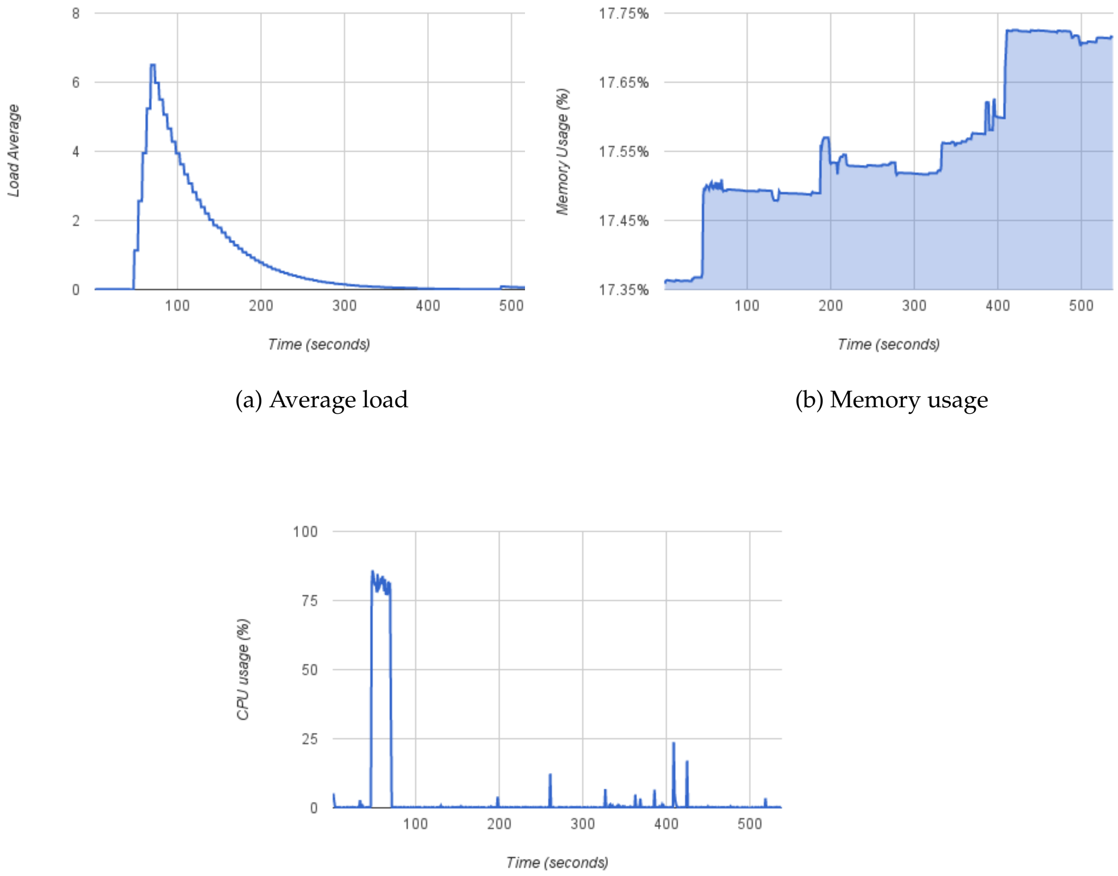
(b) Memory usage

(c) CPU usage

Figure 5.8: Resource usage on user node during the execution of a MapReduce job

execution of a given job. After the execution of the job we can observe that the levels of memory drop to approximately 5%, that is close to the normal usage while the machine is idle. When the job execution finishes, the user node keeps executing the Worker Manager component. It is possible to notice that the resources' usage required by this component is very low.

The CPU usage of the user nodes is presented on Figure 5.8c and they have a behavior similar to the memory usage. From this Figure we can notice that some workers reduce their CPU usage before others. This happens because some of them finish their working unit before others. After the majority of nodes reducing their CPU usage to 0%, there is a small pause and then we can see that a small number of nodes have an high usage. Those nodes are the ones with reduce tasks assigned. The decrease of CPU between the map and reduce phase is due to the exchange of data with other nodes.



(a) Average load

(b) Memory usage

(c) CPU usage

Figure 5.9: Resource usage on master node during the execution of a MapReduce job

Figure 5.9 presents the average load, memory and CPU usage of the master node. If we analyze the load average of the master node that is presented on Figure 5.9a we see that the resources usage on the master node have a peak at the beginning of the job and then decreases and remains low until the end of the job. The reason why we have a peak is the same reason as the CPU usage has high values at the beginning (Figure 5.9c). This happens because all the mappers require their input chunk at approximately the same time. As those input chunks are compressed using G-zip, the master has to perform some computer intensive operations during that period. After that, the CPU is idle and we see only small peaks that match the end of shuffle and reduce phases. Besides the G-zip compression, the master also has to keep control over the execution of the tasks, available workers and job metrics. That state is all kept in memory so that the job managers can follow the job status and statistics of its execution.

When the job is complete, that state is written to disk. Even so, we keep all that statistical data on memory for later access and comparison using an interface visible to developers. That is the reason why even after the job completion the master keeps the same levels of memory usage.

Summary, Assessment and Possible Extensions

In this chapter we evaluated the proposed solution and discussed its results. The performance of the system has been evaluated using two popular benchmarks, one that reduces the amount of data exchanged in relation to the initial input size and other that produces more data than the received initially. The tests covered a variety of distributed configurations, enabling us to draw some conclusions about its advantages and existing limitations.

For increased availability, the system could assess geo-replicated data with divergence bounding guarantees, previous approaches can be leveraged (Esteves et al. 2012). This might be integrated with web caching technologies (Negrão et al. 2014). Data can alternatively also be transferred resorting to BitTorrent-like protocols and optimized by exploiting intrinsic redundancy within and across workloads (Barreto et al. 2012).

Other solutions still require installation of BOINC client, even if extensible, and not just a regular browser (de Oliveira e Silva et al. 2008a), or require the usage of VM technology as in the case of community clouds (Khan et al. 2013). Mobile devices, more prevalent today, can be leveraged following previous efforts to include harvesting computing power from mobile devices (de Oliveira e Silva et al. 2008b).

If executing tasks externally, application transparent checkpointing techniques for Java environments can be employed (Simão et al. 2012). Scheduling of tasks can be deciding automatically, over time, how many tasks to spawn based on resource availability and progress achieved (de Oliveira e Silva et al. 2011). For each task, available resources could be further monitored and managed according to previous work in Java environments, taking application progress into account (Simão & Veiga 2012).Decisions can be expressed resorting to declarative policies easily expressed in XML (Veiga & Ferreira 2004) instead of hardwired in the code. Energy efficiency can also be taken into account to drive resource and task scheduling (Sharifi et al. 2014).

6 Conclusion and Future Work

This thesis proposed a solution that is able to execute MapReduce jobs on regular Web Browsers. We have started this document by introducing the state of art solutions in which regards distributed social networks, big data processing and community clouds. Then we presented the architecture and implementation details that allowed us to develop a prototype.

The prototype that has been implemented provides parallelization of tasks by distributing computation across a large number of nodes. The solution uses P2P communications to reduce the overheads at the server and increase the system's scalability. It is able to tolerate failures on distributed nodes by using replication of checkpoints that can later be used to recover from failures. With the evaluation of our solution, it was demonstrated that it is possible to compute MapReduce jobs using regular Web Browsers and that similar solutions could be used to offload some of the computing requirements of OSNs.

It was shown that the solution is able to scale to a large number of distributed nodes, allowing us to reduce the time to complete jobs. Also, the duration of jobs increases linearly when submitted to an increasing amount of initial input data. The scheduling solution has proven to achieve the desired results by allowing a good load distribution across a large number of distributed nodes. When submitted to an increasing level of distribution, the system was able to complete the execution of jobs with relatively low overheads. The techniques used on the implementation of the master node allowed us to have a solution that requires very few computing resources to be maintained when compared to other MapReduce solutions. The current major limitation of the system is on the quantity of data that each node can process during the shuffle phase. This limitation is inherited from the environment in which it executes and occurs because the current implementation keeps all data in memory during the execution of a job. Next section we propose some topics of interest for future work.

6.1 Future Work

The solution presented on this thesis provides some improvements when compared with the state of art solutions. Still, there are several challenges that can be addressed in future work. This chapter introduces a few areas where, with some minor modifications, it is possible improve the proposed solution.

6.1.1 Scheduling algorithms

Scheduling techniques have an impact on the load distribution, resources usage and response rate of the system. The architecture used in our solution (presented in Section 3.4) enables the scheduling algorithm to be easily changed. We used a simple algorithm that randomly distributes the load among the available workers. Future work may focus on using more advance scheduling techniques.

Some variations of the algorithm can be easily implemented by changing the sort algorithm of the workers queue. For example, if one wants to ensure that more tasks are assigned to workers with better computational resources, then a sort algorithm can be developed that takes into consideration the speed of each worker for executing tasks. The algorithm could compute the ratio between the time taken by a worker to complete a task and the average time to complete tasks of the same job and phase. Assuming that the load of tasks of the same job and phase is evenly distributed, workers that are normally faster completing the execution of tasks would be assigned more frequently than the ones that are slower. The ration might be updated asynchronously after the completion of each job so that the values remain updated without disrupting the scheduler performance.

In the case where the solution is used by OSNs and data processing is done at the users' browsers, a different approached may use locality information available at the OSN to optimize jobs' execution. Users that are geographically located could be assigned tasks of the same jobs. The implementation would consist on sorting workers based on their proximity.

6.1.2 Active Replication of work units

For the current solution, each work unit is assigned to only one worker and fault tolerance is obtained by replicating checkpoints to other nodes. The advantages of using such solution have been presented in Section 3.5. However, the current solution does not consider the case where each user node can not be trusted. In that case, results should be validated. Other systems presented in Section 2.3 have solved this issue by replicating the tasks to more than one worker and validating the results by comparing them. Another advantage of replicating tasks to more workers would be to recover more rapidly from failures because the system could simply wait for a replica (that is executed in parallel) to complete.

6.1.3 Improvements on communications between user nodes

On distributed computing systems, communications done between different nodes have an high impact on the system performance. We have used data compression for the communications between user and master nodes. However, the communications between user nodes is done using an uncompressed data channel. Future work might have significant performance improvements by enabling data compression between user nodes. More specifically, for data intensive jobs, we expect that the shuffle and replication time will decrease.

Another area of interest is to reduce the time needed to start the MapReduce computation. Currently, each mapper has to download its input chunk before starting the execution of tasks. This operation has a significant impact on the jobs' execution time as we have seen in the previous chapter (Chapter 5). The solutions we have analyzed on chapter 2.2 have limited this problem by executing computation locally to where data is saved. This can be done because the nodes that save data are the same that execute computation over that data. Applying similar techniques in our solution is possible but it would require a distributed file system that saves data on user's browsers. A simpler approach to solve the same issue would be to allow worker to prefetch data before the job's execution start and then assign work units to workers that can compute data locally.

6.1.4 Reduce memory usage

In the evaluation section, it was demonstrated that the system has some problems related with memory usage of user nodes. As explained, this problem arises from a design decision of keeping all data in memory during the execution of a job. During the implementation of the prototype we also have tested the usage of File API [1] for saving data on disk. However, this has not solved the problem because the system continued having to load all data to memory for merging and sorting data received during the shuffle phase. We believe that future work might obtain significant improvements on this topic by using other local storage facilities provided by Web Browsers such as IndexdDB [2] or Web Storage [3].

6.1.5 Control resources usage

Currently the users can choose with how many workers they want to contribute. This has effect on the number of threads created and CPU's usage levels. However, there is no control regarding memory or network's usage. Also, it should be possible to use resources only when they are idle. Current Web Browsers do not expose information about the resources' usage of the host computer, however it is possible to develop plugins that integrate with the Web Browser and provide this information. In order to have better control over resource's usage, one could develop a simple browser plugin that would communicate with the Worker Manager frequently, indicating the number of workers that should be maintained. This level would depend on the resources' usage of the user's host at a given moment. When the provided number of workers varies, the Worker Manager would terminate the execution of workers or start the execution of new ones. In order to minimize the computation lost when such event occurs, the Worker Manager could try to terminate the ones that are not executing or that have started the execution of tasks more recently.

[1] http://www.w3.org/TR/FileAPI/
[2] http://www.w3.org/TR/IndexedDB/
[3] http://www.w3.org/TR/webstorage/

6.1.6 MapReduce Workflows

One important usage of MapReduce framework is to build workflows of jobs where one job's result is the input of other. This feature allows the execution of complex workflows and iterative algorithms. The current implementation has no direct support for this but a simple implementation can be created by using the existing REST endpoints by regularly scanning to check if a job is complete and then create new jobs with its results. Direct support can also be implemented but would only make sense when combined with a programmatic API. To enable this feature with better performance, workers would keep the results of reduce phase locally and would use them as the input of the next iteration. For the last iteration of the job, the results would be saved at the master node so that the developers can access them.

Bibliography

Agarwal, S., B. Mozafari, A. Panda, H. Milner, S. Madden, & I. Stoica (2013). Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42. ACM.

Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pp. 4–10. IEEE.

Anderson, D. P., J. Cobb, E. Korpela, M. Lebofsky, & D. Werthimer (2002). Seti@ home: an experiment in public-resource computing. *Communications of the ACM 45*(11), 56–61.

Apolónia, N., P. Ferreira, & L. Veiga (2012). Trans-social networks for distributed processing. In *NETWORKING 2012*, pp. 82–96. Springer.

Barreto, J., L. Veiga, & P. Ferreira (2012, May). Hash challenges: stretching the limits of compare-by-hash in distributed data deduplication. *Information Processing Letters 112*(10), 380–385.

Borges, M. & L. Veiga (2014, September). SN-Edge (comunicação). In *INFORUM (Simpósio de Informática)*.

Buchegger, S., D. Schiöberg, L.-H. Vu, & A. Datta (2009). Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pp. 46–52. ACM.

Chaiken, R., B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, & J. Zhou (2008). Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment 1*(2), 1265–1276.

Chard, K., S. Caton, O. Rana, & K. Bubendorfer (2010). Social cloud: Cloud computing in social networks. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 99–106. IEEE.

Couvares, P., T. Kosar, A. Roy, J. Weber, & K. Wenger (2007). Workflow management in condor. In *Workflows for e-Science*, pp. 357–375. Springer.

de Oliveira e Silva, J. N., L. Veiga, & P. Ferreira (2008a, October). nuBOINC: BOINC extensions for community cycle sharing. In *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (3rd IEEE SELFMAN workshop)*. IEEE.

de Oliveira e Silva, J. N., L. Veiga, & P. Ferreira (2008b, December). SPADE: scheduler for parallel and distributed execution from mobile devices. In *ACM/IFIP/USENIX 9th International Middleware Conference (6th International Workshop on Middleware for Pervasive and Ad-hoc Computing - MPAC 2008)*. ACM.

de Oliveira e Silva, J. N., L. Veiga, & P. Ferreira (2011, September). A2HA - automatic and adaptive host allocation in utility computing for bag-of-Tasks. *Journal of Internet Services and Applications (JISA) 2*(2), 171–185.

Dean, J. & S. Ghemawat (2008, January). Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

Esteves, S., J. N. de Oliveira e Silva, & L. Veiga (2012, August). Quality-of-Service for consistency of data geo-replication in cloud computing. In *International European Conference on Parallel and Distributed Computing (Euro-Par 2012)*. Springer, LNCS.

Ghemawat, S., H. Gobioff, & S.-T. Leung (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, Volume 37, pp. 29–43. ACM.

Hull, D., K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, & T. Oinn (2006, July). Taverna: a tool for building and running workflows of services. *Nucleic Acids Research 34*(suppl 2), W729–W732.

Isard, M., M. Budiu, Y. Yu, A. Birrell, & D. Fetterly (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, New York, NY, USA, pp. 59–72. ACM.

Khan, A., L. Navarro, L. Sharifi, & L. Veiga (2013, October). Clouds of small things: Provisioning infrastructure-as-a-service from within community networks. In *2nd International Workshop on Community Networks and Bottom-up-Broadband (CNBuB 2013) , colocated with 9th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2013, Lyon, France, October 7-9, 2013. IEEE 20*. IEEE.

Kryczka, M., R. Cuevas, C. Guerrero, E. Yoneki, & A. Azcorra (2010). A first step towards user assisted online social networks. In *Proceedings of the 3rd workshop on social network systems*, pp. 6. ACM.

Langhans, P., C. Wieser, & F. Bry (2013). Crowdsourcing mapreduce: Jsmapreduce. In *Proceedings of the 22nd international conference on World Wide Web companion*, pp. 253–256. International World Wide Web Conferences Steering Committee.

Mundkur, P., V. Tuulos, & J. Flatow (2011). Disco: a computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pp. 84–89. ACM.

Narendula, R., T. G. Papaioannou, & K. Aberer (2011). My3: A highly-available p2p-based online social network. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pp. 166–167. IEEE.

Negrão, A., C. M. S. Roque, P. Ferreira, & L. Veiga (2014, December). Adaptive semantics-aware management for web caches. *Journal of Internet Services and Applications (JISA)*.

Nettleton, D. F. (2013). Data mining of social networks represented as graphs. *Computer Science Review* 7(0), 1 – 34.

Neumeyer, L., B. Robbins, A. Nair, & A. Kesari (2010). S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177. IEEE.

Olston, C., B. Reed, U. Srivastava, R. Kumar, & A. Tomkins (2008). Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, New York, NY, USA, pp. 1099–1110. ACM.

Rao, B. T. & L. Reddy (2011). Survey on improved scheduling in hadoop mapreduce in cloud environments. *International Journal of Computer Applications 34*.

Rosenberg, J., R. Mahy, P. Matthews, & D. Wing (2008, October). Session traversal utilities for nat (stun). RFC 5389, RFC Editor. http://www.rfc-editor.org/rfc/rfc5389.txt.

Ryza, S. & T. Wall (2010). Mrjs: A javascript mapreduce framework for web browsers. *URL http://www. cs. brown. edu/courses/csci2950-u/f11/papers/mrjs. pdf*.

Salvatore Loreto, S. P. R. (2010). *Real-Time Communication with WebRTC*, Volume 1. O'Reilly Media.

Shakimov, A., H. Lim, L. P. Cox, & R. Cáceres (2008). Vis-à-vis: Online social networking via virtual individual servers. *submitted for publication*.

Sharifi, L., N. Rameshan, F. Freitag, & L. Veiga (2014, December). Energy efficiency dilemma: P2P-cloud vs. mega-datacenter (best-paper candidate). In *IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE.

Sharma, R. & A. Datta (2012). Supernova: Super-peers based architecture for decentralized online social networks. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pp. 1–10. IEEE.

Shvachko, K., H. Kuang, S. Radia, & R. Chansler (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10. IEEE.

Simão, J., T. Garrochinho, & L. Veiga (2012, September). A checkpointing-enabled and resource-aware java VM for efficient and robust e-Science applications in grid environments. *Concurrency and Computation: Practice and Experience 24*(13), 1421–1442.

Simão, J. & L. Veiga (2012, September). Qoe-JVM: An adaptive and resource-aware java runtime for cloud computing. In *2nd International Symposium on Secure Virtual Infrastructures (DOA-SVI 2012), OTM Conferences 2012*. Springer, LNCS.

Thain, D., T. Tannenbaum, & M. Livny (2005). Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience 17*(2-4), 323–356.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, & R. Murthy (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2(2), 1626–1629.

Thusoo, A., Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, & H. Liu (2010). Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1013–1020. ACM.

Tran, D. N., F. Chiang, & J. Li (2008). Friendstore: cooperative online backup using trusted nodes. In *Proceedings of the 1st Workshop on Social Network Systems*, pp. 37–42. ACM.

Veiga, L. & P. Ferreira (2004, September). Poliper: Policies for mobile and pervasive environments. In *3rd Workshop on Adaptive and Reflective Middleware (5th ACM International Middleware Conference)*, Volume 6. ACM.

Veiga, L., R. Rodrigues, & P. Ferreira (2007). Gigi: An ocean of gridlets on a "grid-for-the-masses". In *CCGRID*, pp. 783–788. IEEE Computer Society.

Verma, A., L. Cherkasova, & R. H. Campbell (2011). Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 235–244. ACM.

White, T. (2013). *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

Wolf, J., D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, & A. Balmin (2010). Flex: A slot allocation scheduling optimizer for mapreduce workloads. *Middleware 2010*, 1–20.

Zaharia, M., D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, & I. Stoica (2009). Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*.

Zaharia, M., D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, & I. Stoica (2010). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pp. 265–278. ACM.