UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

DEPENDABLE MAPREDUCE IN A CLOUD-OF-CLOUDS

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE EM ENGENHARIA INFORMÁTICA

Pedro Alexandre Reis Sá da Costa

Tese orientada por:
Professor Doutor Fernando Manuel Valente Ramos
Professor Doutor Miguel Nuno Dias Alves Pupo Correia

Documento especialmente elaborado para a obtenção do grau de doutor

2017

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

Ciências
ULisboa

DEPENDABLE MAPREDUCE IN A CLOUD-OF-CLOUDS

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE EM ENGENHARIA INFORMÁTICA

Pedro Alexandre Reis Sá da Costa

Tese orientada por:
Professor Doutor Fernando Manuel Valente Ramos
Professor Doutor Miguel Nuno Dias Alves Pupo Correia

Júri:

Presidente:

- Doutor Nuno Fuentecilla Maia Ferreira Neves, Professor Catedrático, Faculdade de Ciências da Universidade de Lisboa

Vogais:

- Doutor Nuno Manuel Ribeiro Preguiça, Professor Associado, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
- Doutor Mário Marques Freire, Professor Catedrático, Faculdade de Engenharia da Universidade da Beira Interior
- Doutor Nuno Fuentecilla Maia Ferreira Neves, Professor Catedrático, Faculdade de Ciências da Universidade de Lisboa
- Doutor Francisco José Moreira Couto, Professor Associado com Agregação, Faculdade de Ciências da Universidade de Lisboa
- Doutor Fernando Manuel Valente Ramos, Professor Auxiliar, Faculdade de Ciências da Universidade de Lisboa

Documento especialmente elaborado para a obtenção do grau de doutor

2017

Aos meus pais.

Abstract

MapReduce is a simple and elegant programming model suitable for loosely coupled parallelization problems — problems that can be decomposed into subproblems. Hadoop MapReduce has become the most popular framework for performing large-scale computation on off-the-shelf clusters, and it is widely used to process these problems in a parallel and distributed fashion. This framework is highly scalable, can deal efficiently with large volumes of unstructured data, and it is a platform for many other applications. However, the framework has limitations concerning dependability. Namely, it is solely prepared to tolerate crash faults by re-executing tasks in case of failure, and to detect file corruptions using file checksums. Unfortunately, there is evidence that arbitrary faults do occur and can affect the correctness of MapReduce execution. Although such Byzantine faults are considered to be rare, particular MapReduce applications are critical and intolerant to this type of fault. Furthermore, typical MapReduce implementations are constrained to a single cloud environment. This is a problem as there is increasing evidence of outages on major cloud offerings, raising concerns about the dependence on a single cloud.

In this thesis, I propose techniques to improve the dependability of MapReduce systems. The proposed solutions allow MapReduce to scale out computations to a multi-cloud environment, or cloud-of-clouds, to tolerate arbitrary and malicious faults and cloud outages. The proposals have three important properties: they increase the dependability of MapReduce by tolerating the faults mentioned above; they require minimal or no modifications to users' applications; and they achieve this increased level of fault tolerance at reasonable cost. To achieve these goals, I introduce three key ideas: minimizing the required replication; applying context-based job scheduling based on cloud and network conditions; and performing fine-grained replication.

I evaluated all proposed solutions in real testbed environments running typical MapReduce applications. The results demonstrate interesting trade-offs concerning resilience and performance when compared to traditional methods. The fundamental conclusion is that the cost introduced by our solutions is small, and thus deemed acceptable for many critical applications.

Keywords: MapReduce, Big Data, Fault-Tolerance, Cloud-of-Clouds

Resumo

O MapReduce é um modelo de programação adequado para processar grandes volumes de dados em paralelo, executando um conjunto de tarefas independentes, e combinando os resultados parciais na solução final. O Hadoop MapReduce é uma plataforma popular para processar grandes quantidades de dados de forma paralela e distribuída. Do ponto de vista da confiabilidade, a plataforma está preparada exclusivamente para tolerar faltas de paragem, re-executando tarefas, e detectar corrupções de ficheiros usando somas de verificação. Esta é uma importante limitação dado haver evidência de que faltas arbitrárias ocorrem e podem afetar a execução do MapReduce. Embora estas faltas Bizantinas sejam raras, certas aplicações de MapReduce são críticas e não toleram faltas deste tipo. Além disso, o número de ocorrências de interrupções em infraestruturas da nuvem tem vindo a aumentar ao longo dos anos, levantando preocupações sobre a dependência dos clientes num fornecedor único de serviços de nuvem.

Nesta tese proponho várias técnicas para melhorar a confiabilidade do sistema MapReduce. As soluções propostas permitem processar tarefas MapReduce num ambiente de múltiplas nuvens para tolerar faltas arbitrárias, maliciosas e faltas de paragem nas nuvens. Estas soluções oferecem três importantes propriedades: toleram os tipos de faltas mencionadas; não exigem modificações às aplicações dos clientes; alcançam esta tolerância a faltas a um custo razoável. Estas técnicas são baseadas nas seguintes ideias: minimizar a replicação, desenvolver algoritmos de escalonamento para o MapReduce baseados nas condições da nuvem e da rede, e criar um sistema de tolerância a faltas com granularidade fina no que respeita à replicação.

Avaliei as minhas propostas em ambientes de teste real com aplicações comuns do MapReduce, que me permite demonstrar compromissos interessantes em termos de resiliência e desempenho, quando comparados com métodos tradicionais. Em particular, os resultados mostram que o custo introduzido pelas soluções são aceitáveis para muitas aplicações críticas.

Palavras Chave: MapReduce, *Big Data*, Tolerância a Faltas, Nuvem de Nuvens

Resumo Alargado

A computação em nuvem surgiu como um paradigma de infra-estruturas de computação em larga escala que pode oferecer uma economia significativa de custos através da alocação dinâmica de recursos (Armbrust *et al.*, 2009). O termo computação em nuvem reporta-se tanto às aplicações fornecidas como serviços pela Internet tal como ao *software* nos centro de dados que disponibilizam esta plataforma aos utilizadores. Para materializar este conceito, os fornecedores de serviços de nuvem têm construído grandes centros de dados que são muitas vezes distribuídos por várias regiões geográficas para atender eficientemente à procura dos utilizadores. Estes centros de dados são construídos usando centenas de milhares de servidores comuns, e usam a tecnologia de virtualização para fornecer recursos de computação partilhado. O uso de componentes comuns expõe o hardware a falhas que acabam por reduzir a confiabilidade e a disponibilidade do serviço. O desafio de construir nuvens confiáveis e robustas é por isso um problema crítico que merece atenção.

A computação em nuvem permite o processamento de enormes volumes de dados que as técnicas tradicionais de base de dados e *software* têm tido dificuldade em processar dentro de tempos aceitáveis (Snijders *et al.*, 2012). Estes serviços que lidam com grandes quantidades de dados, *big data*, fornecem aos utilizadores a capacidade de usar servidores comuns para efectuar processamento distribuído. Um dos mais populares sistemas de processamento de dados distribuídos que é usado para analisar grandes quantidades de dados num ambiente em nuvem é o MapReduce.

Em 2004, a Google apresentou este modelo de programação — o MapReduce — e sua implementação (Dean & Ghemawat, 2004). O MapReduce é usado extensivamente pela Google nos seus centros de dados para suportar funções essenciais como o processamento de índices para os motores de busca. Este sistema é muito bem sucedido, mas a implementação não está publicamente disponível.

Alguns anos mais tarde, uma implementação da plataforma MapReduce foi desenvolvida num projeto de código aberto da Apache chamado Hadoop (White, 2009). Esta versão é agora usada por muitas empresas de computação em nuvem, incluindo a Amazon, a IBM, a RackSpace e a Yahoo.

O termo MapReduce denomina tanto um modelo de programação como o respectivo ambiente de execução. A programação em MapReduce envolve o desenvolvimento de duas funções: `map` e `reduce` — funções de mapa e redução. Uma execução completa das funções de mapa e redução ocorrem numa unidade de trabalho denominada de *job*. A execução do *job* é composta por várias fases sequenciais. Cada arquivo de entrada é processado pelas tarefas de mapa (fase de mapa), que produzem um resultado que é gravado num ou vários ficheiros de saída. Estes dados de saída são particionados, transferidos e ordenados (fase de *shuffle* e *sort*) para as tarefas de redução (fase de redução) que irão produzir o resultado final. De acordo com Dean e Ghemawat, este modelo pode expressar muitas aplicações do mundo real (Dean & Ghemawat, 2004). O modelo generalizado do MapReduce e a variedade de aplicações que usam a plataforma oferecem evidência clara da sua popularidade.

A plataforma MapReduce foi projetada para ser tolerante a faltas de paragem, porque à escala de milhares de computadores e outros dispositivos (encaminhadores de rede e roteadores, unidades de energia), as falhas de componentes são frequentes. O MapReduce da Google e o Hadoop usam essencialmente dois mecanismos de tolerância a faltas: monitorização da execução das tarefas de mapa e redução; e somas de verificação nos ficheiros gravados no sistema distríbuido para que as corrupções de arquivos possam ser detectadas (Ghemawat *et al.*, 2003; White, 2009).

Embora seja crucial tolerar faltas de paragem e corrupção de dados no disco, outras faltas podem afetar o *resultado* do MapReduce. Estas faltas Bizantinas são conhecidas por acontecerem raramente, mas devido ao crescimento do número de centro de dados, há já evidência destas ocorrências, e há uma grande probabilidade destas faltas acontecerem com maior frequência no futuro (Qiang Wu & Mutlu, 2015; Schroeder & Gibson, 2007).

De facto, os servidores disponibilizados na nuvem são propensos a erros no *hardware* que se podem propagar para o *software* (Li *et al.*, 2008; Schroeder *et al.*, 2009), causando não apenas o abortar da aplicação, mas também causando falhas subtis que podem levar a erros nos resultados da computação. Esses erros podem afetar a execução de processos que causarão o comportamento incorreto do sistema. O MapReduce foi projetado para trabalhar em grandes centros de dados onde esse tipo de erros tende a ocorrer (Nolting, 2012; Spectator, 2015) e os mecanismos de tolerância a faltas do MapReduce original não podem lidar com estas *faltas arbitrárias* ou *faltas Bizantinas* (Avizienis *et al.*, 2004; Lamport *et al.*, 1982).

Um outro constrangimento do MapReduce em relação à confiabilidade é a execução ser limitada a um único centro de dados, ou seja, a uma única nuvem, o que torna esta aplicação vulnerável à falha destas infra-estruturas. Nos últimos anos, temos visto um aumento no número de ocorrências de erros relatados em nuvens públicas que afetaram a disponibilidade dos serviços (Cerin *et al.*, 2013; Clarke, 2015; Dvorak, 2011; Raphael, 2014).

A solução que propomos nesta tese baseia-se no uso de múltiplas nuvens como um ambiente para executar aplicações MapReduce, o que introduz várias vantagens (Lucky, 2015). Primeiro, a capacidade de executar aplicações em diferentes fornecedores de serviço tem o benefício de reduzir a dependência num único fornecedor, o que melhora a posição do cliente na negociação de um acordo de nível de serviço (SLA). Depois a capacidade de trocar facilmente de operadores significa que um cliente pode aproveitar ofertas mais atraentes a qualquer momento sem ter problemas em usar plataformas diferentes. Em terceiro lugar, a possibilidade de ter parte da computação executada em nuvens públicas ou privadas com base em uma variedade de considerações (por exemplo, algumas mais confiáveis que outras) pode aumentar a segurança. Além disso, esta escolha permite colocar os serviços mais próximos da localização geográfica do cliente, podendo resultar num melhor tempo de resposta. Finalmente, o uso de múltiplas nuvens é uma técnica útil para melhorar a resiliência (Garcia *et al.*, 2011), o aspecto em que me centro.

Foram feitas recentemente algumas propostas no sentido de escalar o MapReduce para várias nuvens (Jayalath *et al.*, 2014; Wang *et al.*, 2013). No entanto, estas propostas centram-se na escalabilidade e não consideram tolerância a faltas.

Como a adoção do MapReduce continua a proliferar em áreas críticas, torna-se necessário lidar com modelos de falhas para além dos previstos pelo MapReduce original. Motivados por tais aplicações críticas, as soluções que proponho nesta tese visam obter uma execução eficiente do MapReduce num ambiente de múltiplas nuvens de modo a conseguir tolerar faltas arbitrárias e maliciosas.

Como primeira contribuição, desenvolvi um sistema BFT MapReduce, que é capaz de tolerar faltas arbitrárias através do uso da técnica de replicação. O desafio foi executar as tarefas de mapa e redução *eficientemente* sem a necessidade de executar $3f+1$ réplicas para tolerar f faltas, o que seria o caso com a replicação de máquinas de estado (por exemplo, Castro & Liskov (2002); Clement *et al.* (2009a); Veronese *et al.* (2009)). A solução desenvolvida usa várias técnicas para minimizar a replicação. Para garantir o bom desempenho do sistema, desenvolvi dois inovadores algoritmos de escalonamento, *especulativo* e *não-especulativo*, que têm como objetivo melhorar o desempenho das execuções no MapReduce. Avaliei pormenorizadamente o desempenho do sistema num ambiente de teste real, mostrando que tem um custo aceitável em comparação com soluções alternativas.

Uma limitação desta primeira solução é considerar um ambiente numa única nuvem, o que a torna incapaz de tolerar interrupções na nuvem. Assim, a segunda contribuição explora a idéia de ambientes virtuais compostos por múltiplas nuvens para evitar a incorreção ou a indisponibilidade de computação devido a faltas arbitrárias, maliciosas e a interrupções na nuvem.

Ressalva-se que o uso de múltiplas nuvens para o MapReduce não é em si novo. A novidade deste trabalho decorre do uso de um ambiente multi-nuvem não apenas para paralelizar a computação, mas também para tolerar de forma transparente este tipo de faltas. Esta solução, denominada Medusa, aborda vários desafios não triviais. Em primeiro lugar, o Medusa é uma solução transparente

para os utilizadores, que podem correr as suas aplicações MapReduce sem necessitarem de alterações. Em segundo lugar, dado que o Medusa não exige modificações no código-fonte do Hadoop, a solução é compatível com várias versões da plataforma. Em terceiro lugar, o Medusa propõe alcançar este nível de tolerância a faltas com um custo mínimo de replicação para garantir um desempenho aceitável. A extensa avaliação experimental realizada em ambientes reais permite demonstrar que estes desafios foram concretizados.

Esta solução tem, no entanto, uma limitação: apenas tolera faltas ao nível do *job*. Numa situação comum em que se executa um *job* composto por múltiplas tarefas, uma única falha numa tarefa exige que todo o trabalho seja re-executado. Assim, a terceira contribuição propõe um sistema, o Chrysaor, baseado num esquema de replicação de granularidade fina que permite recuperar de faltas ao nível da tarefa e não do *job*. Embora esta nova solução tenha o mesmo objectivo que o Medusa, existem diferenças claras a nível da arquitectura. O desafio de alcançar esta granularidade fina é exponenciado por um dos nossos requisitos: não alterar o código-fonte do Hadoop MapReduce. Para atingir este objectivo, o Chrysaor propõe uma nova abstração denominada de "*job* lógico".

A avaliação extensa que fiz do sistema na Amazon EC2 demonstra que a granularidade fina melhora a eficiência na presença de faltas. Isto é alcançado sem incorrer numa penalidade significativa para o caso comum (ou seja, sem falhas) na maioria das execuções.

Em conclusão, as soluções inovadoras que proponho nesta tese - e que partilho em forma de código aberto [1] - permitem melhorar a confiabilidade do sistema MapReduce, sem comprometer o seu desempenho. Deste modo, acredito que estas soluções possam ser uma contribuição importante para os utilizadores destas plataformas, em particular no contexto de aplicações críticas.

Palavras Chave: MapReduce, *Big Data*, Tolerância a Faltas, Nuvem de Nuvens

[1] O código fonte do Hadoop MR BFT encontra-se disponível em https://bitbucket.org/pcosta_pt/hadoop-bft/, o código fonte do Medusa encontra-se disponível em https://bitbucket.org/pcosta_pt/medusa e o código fonte do Chrysaor encontra-se disponível em https://bitbucket.org/pcosta_pt/chrysaor.

Acknowledgements

First and foremost, I want to acknowledge my advisor, Professor Fernando M. V. Ramos, and co-advisor, Professor Miguel Pupo Correia. Their support and guidance were crucial to the success of this doctoral work, and I am appreciative of the confidence and freedom they have indulged me throughout this journey. I also appreciate their professionalism, time, support, wise advises, motivation, and patience. They have been an inspiration for pursuing perfection and for becoming a better academic researcher. My deepest and heartfelt regards!

I would also like to acknowledge that the research that was developed during the Ph.D. was only possible with the support of the current and past members of the Navigator group, and the support of INESC-ID.

I want to acknowledge the Universidade de Lisboa. This institution has shaped and formed many men and women since its birth in 1911. Based on the motto *ad lucem*, this institution lead its students to the light of knowledge. This has been my home for more than a decade and is not without a misty eye and *saudade* that I now leave. I would like to thank a few notable individuals from this community, in particular, from the Department of Informatics (DI) at Faculty of Sciences, for their invaluable assistance, expertise, and friendship: Professors Miguel Pupo Correia, Fernando Ramos, Marcelo Pasin, Nuno Neves, António Casimiro, Alysson Bessani, Ibéria Medeiros, Luís Correia, and Paulo Veríssimo.

I must thank our colleagues of project FTH-Grid for many discussions on topics: Luciana Arantes, Olivier Marin, Pierre Sens, Fabrício Silva, and Julien Sopena.

I am deeply grateful to Yahoo Research and to Xiao Bai for the opportunity to do an internship at their Barcelona lab from April 2nd to June 30th 2013.

A very special and kindly regards to all my past and present colleagues at the Large-Scale Informatics Systems Laboratory (LaSIGE) research laboratory, for their friendship and tireless support. We have discussed and shared research ideas, they have helped me to organize the Smalltalks and the NavTalks, and have given me support to the Navigators cluster. In particular, thank you very much to Vinícius Cogo, Diego Kreutz, Max Alaluna, Bruno Vavala, Ricardo Pinto, André Nogueira, Ricardo Fonseca, Tulio Ribeiro, Luis Brandão, and many others.

A heartfelt appreciation to my parents, João Sá da Costa and Teresa Sá da Costa, who have been the foundation of my character, and without them, I would not be here today. I do not forget my brother, João Sá da Costa, who we have grown together, played a lot and studied in the same schools. I also do not forget the support of my faithful friends, Ema and Vasco.

Not repeating, but reinforcing, a very special appreciation, and expressing my sincere gratitude, to Fernando and Miguel. All this work would not exist without them.

Finally, I gratefully acknowledge the funding sources that made this doctoral work possible. The work developed in the PhD was partially supported by the European Commission FP7 through project ICT-257243 (TClouds), the SUPERCLOUD project (ref. H2020-643964), by the Fundação para a Ciência e Tecnologia (FCT) and EGIDE (Programme PESSOA) through FTH-Grid project, by the FCT through the project CloudFIT, and by funding of LaSIGE Research Unit (ref. UID/CEC/00408/2013), and INESC-ID (ref. UID/CEC/50021/2013).

I warmly thank the Grid'5000 team for the support that was given of the testbed. This testbed is being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

Contents

List of Figures

List of Tables

List of Notations and Acronyms

BFT	Byzantine Fault Tolerant
DAG	Directed Acyclic Graph
DIMM	Dual In-line Memory Model
DoS	Denial of Service
DRAM	Dynamic Random-Access Memory
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
HDFS	Hadoop Distributed Filesystem
IaaS	Infrastructure as a Service
mappers	Map tasks
MQ	Message Queuing Service
MR	MapReduce
reducers	Reduce tasks
SDN	Software Defined Network
SGX	Software Guard Extensions
SLA	Service Level Agreement
SQL	Structured Query Language
TCP	Transmission Control Protocol
TF-IDF	Term Frequency Inverse Document Frequency
TPM	Trusted Platform Module

List of Publications

Journal Papers

CostaPBC13 Pedro Costa, Marcelo Pasin, Alysson Bessani, Miguel Correia. On the Performance of Byzantine Fault-Tolerant MapReduce, IEEE Transactions on Dependable and Secure Computing, 2013

CostaPBC17 Pedro A. R. S. Costa, Fernando M. V. Ramos, Miguel Correia. On the Design of Resilient Multicloud MapReduce, IEEE Cloud Computing, 2017

Conference Papers

Costa:2016 Pedro A. R. S. Costa, Xiao Bai, Fernando M. V. Ramos, Miguel Correia. Medusa: An Efficient Cloud Fault-Tolerant MapReduce, Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-Grid), 2016

Costa:2017 Pedro A. R. S. Costa, Fernando M. V. Ramos, Miguel Correia. Chrysaor: Fine-Grained, Fault-Tolerant Cloud-of-Clouds MapReduce, Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-Grid), 2017

Workshop paper

Correia.12.SRDS Miguel Correia, Pedro Costa, Marcelo Pasin, Alysson Bessani, Fernando Ramos, Paulo Verissimo. On the Feasibility of Byzantine Fault-Tolerant MapReduce in Clouds-of-Clouds, IEEE Symposium on Reliable Distributed Systems, 2012

1
Introduction

Cloud computing has emerged as a popular paradigm of large-scale computing infrastructures that can offer significant cost savings by allowing resources to be scaled dynamically (Armbrust *et al.*, 2009). Cloud computing refers to both the applications delivered as services over the Internet and systems software in the datacenters that provide those services. Service providers have been building massive datacenters that are distributed over several geographical regions to meet the demand for these cloud services. These datacenters are built using hundreds of thousands of commodity servers and use virtualization technology to provision computing resources. The use of commodity components exposes the hardware to failures that reduce the reliability and availability of the cloud service. Therefore, *fault tolerance* in cloud computing platforms and applications is a crucial issue to the users and to cloud providers. As such, the challenge of building dependable and robust clouds remains a critical research problem.

Cloud computing has enabled computation of massive volumes of data that traditional database and software techniques had difficulty in processing within acceptable time bounds (Snijders *et al.*, 2012). These services that deal with *big data* provide users the ability to use commodity infrastructures to process distributed computation across multiple datasets.

One of the most popular distributed data-processing systems that is used to analyze big data in a cloud environment is MapReduce. In 2004, Google presented this programming model and implementation for processing large data sets in a datacenter (Dean & Ghemawat, 2004). MapReduce is used extensively in its datacenters to support core functions such as the processing of indexes for its web search engine. This system is very successful, but its implementation is not openly available. A few years later, an implementation of the MapReduce framework was adopted by an Apache open source project named Hadoop (White, 2009). This version is now used by many cloud computing companies, including Amazon, IBM, RackSpace, and Yahoo!.[1] Other competing versions also exist, e.g., Microsoft's Daytona (Barga, 2011) and the Amazon Elastic MapReduce (Amazon, 2015).

The term MapReduce denominates both a programming model and the corresponding runtime environment. Programming in MapReduce involves developing two functions: a `map` and a `reduce`. A full execution of the `map` and `reduce` functions happens in a *job*. A job execution is composed of several phases. Each input file in a job is first processed by the mappers (map phase), then the map outputs are partitioned, transferred and sorted (shuffle and sort phase) to the reducers to be processed by a reduce function (reduce phase). According to Dean and Ghemawat, this model can express many real world applications (Dean & Ghemawat, 2004). Indeed, the widespread use of MapReduce since its inception and the array of applications that use this framework offer clear evidence of this fact.

Google's MapReduce platform was designed to be fault-tolerant because, at scales of thousands of computers and other devices (network switches and routers, power units), component failures are frequent. Dean reported the thousands of individual machines, hard drive and memory failures in the first year of a cluster at a Google datacenter (Dean, 2009). The original MapReduce from Google and Apache Hadoop use essentially two fault tolerance mechanisms: they monitor the execution of map and reduce tasks and reinitialize them if they stop; and they add checksums to files with data so that file corruptions can be detected (Ghemawat *et al.*, 2003; White, 2009).

1.1 Problem and Motivation

Although it is crucial to tolerate crashes of tasks and data corruptions in disk, other faults that can affect the *correctness of results* of MapReduce are known to happen, and will probably happen more often in the future (Qiang Wu & Mutlu, 2015; Schroeder & Gibson, 2007). A 2.5-year long study of DRAM errors in numerous servers in Google datacenters concluded that these errors are more prevalent than previously believed, with one third of the machines and over 8% of DIMMs affected by errors yearly, even if protected by error correcting codes (Schroeder *et al.*, 2009). A Microsoft study of 1 million consumer PCs

[1] http://wiki.apache.org/hadoop/PoweredBy

showed that CPU and core chipset faults are also frequent (Nightingale *et al.*, 2011). They have noticed that the number of hardware crashes increases after the first occurrence and that 20% to 40% of machines suffer intermittent faults, including 1-bit DRAM error failures. The authors also claim that 70% to 80% of the uncorrectable errors are preceded by correctable errors, so replacing a DIMM based on the presence of correctable errors is attractive for environments that cannot tolerate downtime. Nevertheless, there remains a substantial number of uncorrectable errors that are not preceded by correctable errors, requiring the investigation of other solutions. Moreover, a study performed at Facebook datacenters over a fourteen month period analyzed memory errors from their entire fleet of servers and observed that higher chip densities of DRAM lead to an increase in the failure rate by 1.8 over previous generations (Qiang Wu & Mutlu, 2015).

In conclusion, hosts in the cloud are prone to soft and hard errors in the hardware that can propagate to the software running atop (Li *et al.*, 2008; Schroeder *et al.*, 2009), causing not only the crash of the application but also causing subtle failures that violate their correctness. These uncorrectable errors can affect the execution of processes causing the system to behave erroneously. Unfortunately, MapReduce is designed to work on large clusters where this type of errors tends to occur (Nolting, 2012; Spectator, 2015), and the fault tolerance mechanisms of the original MapReduce cannot deal with such *arbitrary* or *Byzantine faults* (Avizienis *et al.*, 2004).

Another limitation of MapReduce with respect to dependability is its design based on a single datacenter[1] (i.e., on a single cloud), which makes this framework vulnerable to malicious attacks and cloud outages.

Malicious attacks, which include security attacks that compromise the virtual machine (VM) or the entire server, as well as Denial of Service (DoS) attacks that can make a server, switch, or router to become unavailable. Cloud providers can increase the security of the cloud by replicating the service in independent hardware components. However, if an attacker compromises some replicas inside a cloud, it could use them as agents of a distributed DoS attack. Therefore, replicas located on the same cloud have a similar risk of failure by a DoS attack, even if they are running in independent components (Deshmukh & Devadkar, 2015; Guerraoui & Yabandeh, 2010). According to the Cloud Security Alliance, distributed

[1] We use the terms *cloud*, *cluster* and *datacenter* interchangeably, while acknowledging their difference.

DoS is one of the top nine threats to cloud computing environments (CloudSquare, 2015) and the rate of the attacks is growing at a fast pace (Ventures, 2017).

In the last few years, we have seen an increase in the number of error occurrences reported in public clouds, impacting service availability for hours (Kraft, 2017; Tsidulko, 2015). In 2015, Microsoft Azure service was down in 63 occasions, and Google had its services down 102 times. A cloud outage in these large cloud providers can cause extensive damage. For instance, an outage of Google cloud-based services caused a 40% drop in Internet traffic during a five-minute window (Mack, 2013), or, recently, a five hour outage crashed services belonging to major financial institutions (Dawn-Hiscox, 2017). The number of occurrences has increased, and this trend will continue to grow. In 2016, Salesforce has lost some data due to a "successful site switch" from its primary datacenter, after power supply problems (Sharwood, 2016). The conjunction of a database failure introduced by a file integrity issue and an out of date backup were the cause for the data loss. A few months ago, Amazon has also suffered an outage with great impact on the Internet (Blodget, 2017). One incorrect command sent accidentally by an employee took more servers offline than intended. As result, some clients could not access the service and have lost parts of data that could not be recovered due to an inconsistent snapshot that worked as a backup. These real incidents show that it is necessary to go beyond single-cloud solutions and explore new avenues.

Indeed, using multiple clouds as an environment to run MapReduce applications presents several advantages (Lucky, 2015). First, the ability to deploy applications on different cloud providers has the benefit of reducing dependency on a single provider. The resulting lower level of lock-in improves the customer position in obtaining a better Service-level agreement (SLA). The ability to easily switch operators means that a customer can take advantage of the most attractive offers available at any given time. It is also possible to have some applications that run on both private and public clouds, based on a variety of considerations, such as security, performance or cost optimization. Moreover, deploying applications on a cloud that is closer to the customer's geographical location can result in better response time and performance. Finally, and most significantly in our study, using multi-cloud diversity is a useful technique to improve fault tolerance (Garcia *et al.*, 2011; Lacoste *et al.*, 2016).

Initial work in scaling out MapReduce to multiple clouds has been recently proposed (Jayalath *et al.*, 2014; Wang *et al.*, 2013). However, these proposals focused on scalability and have not considered fault tolerance. For MapReduce to execute geo-distributed operations to cope with the requirement of big data processing, while simultaneously tolerating *Byzantine* faults and cloud outages, the system needs to not only be scaled to multiple clouds, but new techniques have to be developed to tolerate such faults.

Byzantine fault tolerant (BFT) replication techniques are a clear candidate for this purpose, but they are considered expensive as one expects these faults to be rare. However, as explained before, these types of faults are not rare and have, in fact, caused high losses to several companies (Cerin *et al.*, 2013; Clarke, 2015; Raphael, 2014). More importantly, for very critical applications that are heavily dependent on data, such as bioinformatics or finance, any kind of errors and unavailability are unacceptable (Dai *et al.*, 2012; O'Dowd, 2015).

For bioinformatics studies, clouds are crucial for data analysis and knowledge discovery, and delivering a correct computation result is of utmost importance (Truong & Dustdar, 2009). MapReduce is currently being used as a replacement of traditional techniques to speed up several biomedical tasks like identifying unproven cancer treatments or classifying biometric measurements (Ferraro Petrillo *et al.*, 2017; Kohlwey *et al.*, 2011; Mohammed *et al.*, 2014). Financial institutions are also using big data analysis to improve customer's intelligence, reduce risk, and meet regulatory objectives in their solutions (O'Dowd, 2015). The algorithms used for these computations are implemented in MapReduce to make financial predictions such as the future value of stock markets using historical data (Dubey *et al.*, 2015; Mukesh, 2015; Sharma, 2016).

The adoption of MapReduce is continuing to proliferate in critical areas, as these two examples, and the requirement of having to deal with fault models beyond crash failures is becoming of high importance to avoid unintended outcomes. For example, a malicious insider in a cloud that hosts an epidemiological surveillance system who tampers patient's reports may lead to disastrous consequences (Claycomb & Nicoll, 2012). Moreover, temporary unavailability of the financial system in one cloud (due to a cloud outage) may dramatically influence the investment decisions and cause huge financial loss (Blodget, 2017). Motivated by such critical applications, the cost of replication becomes acceptable in order to guarantee that rare faults with devastating consequences do not occur. Moreover, the design approach to follow should aim to minimize this cost.

1.2 Objective

Motivated by the above, we address the referred MapReduce dependability limitations — inability to deal with arbitrary faults and cloud outages — by fulfilling the following requirements: *(i)* scale out MapReduce to multiple clouds, *(ii)* enable MapReduce to tolerate arbitrary and malicious faults, and cloud outages, while *(iii)* guaranteeing acceptable performance.

Scaling out MapReduce computation to multiple clouds is not, in itself, new. The framework has been used over several clouds for large-scale data processing, from the construction of indexes in multi-site web search engines (Baeza-Yates *et al.*, 2009) to the execution of multi-cloud BLAST computations for bioinformatic applications (Matsunaga *et al.*, 2008). More recently, a series of works were proposed (Jayalath *et al.*, 2014; Wang *et al.*, 2013) to perform a sequence of MapReduce jobs across multiple datacenters. All these works have ignored the fault tolerance aspect. The novelty of our proposal arises from the use of a multi-cloud environment not only to parallelize computation but also to achieve fault tolerance. [1]

The arbitrary faults we address in our work cannot be detected using file checksums, so they can silently corrupt the output of any map or reduce task, corrupting the result of the MapReduce job. As explained, this can be problematic for critical applications. However, it is possible to mask the effect of such faults by *executing each task more than once*, comparing the outputs of these executions, and disregarding the non-matching outputs. Sarmenta proposed such an approach in the context of volunteer computing to tolerate malicious volunteers that returned false results of tasks they were supposed to execute (Sarmenta, 2002). In contrast to our setting, the authors considered only bag-of-tasks applications, which are simpler than MapReduce jobs. A similar but more generic solution consists in using the *state machine replication* (SMR) approach (Schneider, 1990). This approach is not directly applicable to the replication of MapReduce tasks and applies only to replicate jobs. With SMR, each replicated job, which is represented by a process, starts in the same initial state and executes the same requests in the same order. In the end, the determinism of the processes will produce the same output. If we assume that each failure only affects one job in the case of arbitrary faults, it is necessary to combine at least *3f+1* outputs produced by

[1] We use the term *multi-cloud* as synonym of *cloud-of-clouds*.

the job replicas to guarantee that a majority of the outputs remains correct even after f failures. This solution offers poor performance, which makes it unattractive when we want to tolerate this type of faults. We could use another simple scheme that would consist in executing jobs sequentially until *f+1* identical outputs were obtained, ensuring the correct result. However, MapReduce is slow in loading a job and reading the input data (Shi *et al.*, 2015) so sequential execution would accumulate these delays. Moreover, each fault would lead to a new job execution, increasing, even more, the execution time. Our objective is to tolerate arbitrary faults while minimizing the replication cost. In addition, we want to replicate the execution across clouds to tolerate the disruption of a cloud service.

Our third goal is to make MapReduce tolerate the faults mentioned above with little impact on performance. By minimizing the number of replicas to execute, as stated above, we aim to improve performance when faults do not occur. To further improve performance, we investigate fine-grained replication techniques and novel MapReduce scheduling algorithms.

As a general goal to foster the adoption of our solutions, we aim for designs that lead to solutions that are *transparent* to users and that minimize or avoid changes to the MapReduce framework.

1.3 Contributions

To fulfill the goals stated before, we make the following contributions in our work:

(i) We start by exploring the notion of replication of computation in order to tolerate arbitrary and malicious faults. We propose a Byzantine fault-tolerant MapReduce framework that increases the dependability of the framework by replicating MapReduce tasks and distributing them efficiently. The challenge is to do this *efficiently*, without the need to run $3f+1$ replicas to tolerate at most f faults, which would be the case with state machine replication (e.g., Castro & Liskov (2002); Clement *et al.* (2009a); Veronese *et al.* (2009)). We use several techniques to reduce the overhead to improve performance, such as running fewer tasks in the scenario with no faults, replicating tasks only when necessary, and offering modes of execution that explore speculation.

We thoroughly evaluated the proposed solution in a real testbed, showing that it uses around twice the resources of the original Hadoop MapReduce, instead of the alternative that triples the number of resources. This work was published in *IEEE Transactions on Dependable and Secure Computing* (Costa *et al.*, 2013). [1]

(ii) The first solution is limited to a single cloud, so our second contribution proposes a multi-cloud setting. The ability to deal with the new types of faults introduced by such settings, such as the outage of a whole datacenter or an arbitrary fault caused by a malicious cloud insider, increases the endeavor considerably. To tackle this, we propose a system, Medusa, that is able to deal with the new types of faults introduced by cloud environments, such as the outage of a whole datacenter and arbitrary or malicious faults. The system is configured with two parameters f and t: f is the maximum number of faulty replicas that can return the same erroneous result given the same input, and t is the number of faulty clouds that the system tolerates before the service becomes unavailable.

Our solution fulfills four objectives. First, it is transparent to the MapReduce applications written by the users. Secondly, it does not require any modification to the Hadoop framework. Thirdly, despite no changes to the original Hadoop, the proposed system goes beyond the fault tolerance mechanism offered by MapReduce, by tolerating arbitrary faults, cloud outages, and even malicious faults caused by corrupt cloud insiders. Fourth, it achieves this increased level of fault tolerance at a reasonable cost.

We have performed an extensive experimental evaluation in the ExoGENI (Baldine *et al.*, 2012) testbed in order to demonstrate that our solution significantly reduces execution time when compared to traditional methods that achieve the same level of resilience.

This work was published in the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing conference [2].

(iii) The previous contribution favors transparency over performance by replicating MapReduce jobs. Replicating jobs is, however, a coarse-grained solution that limits performance in case of faults. By performing fine-grained detection at the task level, we can improve the overall performance of the system in case of task failure. As such, our third contribution, *Chrysaor*, employs an innovative *fine-grained replication* scheme to tolerate faults by re-executing only the tasks that were affected in case of failures. Similar to Medusa, this system is configured with the parameters f and t.

[1] IEEE Transactions on Dependable and Secure Computing is a journal with impact factor of 1.351.

[2] IEEE/ACM CCGRID is a core A conference in cluster, cloud, and grid computing with Google h-index=38. The acceptance rate is around 20%.

This solution has three important properties: it tolerates arbitrary and malicious faults, and cloud outages at a reasonable cost; it requires minimal modifications to the users' applications; and it does not involve changes to the Hadoop source code.

We performed an extensive evaluation of this system in Amazon EC2, showing that our fine-grained solution is efficient in terms of computation by recovering only faulty tasks without incurring a significant penalty for the baseline case (i.e., without faults) in most workloads.

This work was published in the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing conference and won the Best Student Paper Award.

In summary, we show in Table 1.1 the characteristics of our contributions. These solutions increase Hadoop MapReduce dependability but employ different system models. BFT MapReduce deals with Byzantine faults in a single cluster, and Medusa and Chrysaor deal with Byzantine faults and cloud outages in a multi-cloud environment.

Table 1.1: Summary of contributions

Chapter	System Name	Execution Environment	Replication level	Faults tolerated	Definition of f and t
3	BFT MapReduce [a]	Single cloud	Tasks	Accidental and Byzantine faults	f = max. number of faulty task replicas that return wrong output.
4	Medusa [b]	Cloud-of-clouds	Jobs	Accidental, malicious, and Byzantine faults + cloud outages	f=max. number of faulty task replicas that return the same wrong output; t = max. number of faulty clouds.
5	Chrysaor [c]	Cloud-of-clouds	Tasks	Accidental, malicious and Byzantine faults + cloud outages	f=max. number of faulty task replicas that return the same wrong output; t= max. number of faulty clouds.

[a] Source code available at https://bitbucket.org/pcosta_pt/hadoop-bft/

[b] Source code available at https://bitbucket.org/pcosta_pt/chrysaor/

[c] Source code available at https://bitbucket.org/pcosta_pt/medusa/

1.4 Structure of the Thesis

The dissertation is organized as follows:

Chapter 2 provides the context in which the thesis appears and presents the related work.

Chapter 3 presents a Byzantine fault-tolerant MapReduce that can mask arbitrary faults by executing each task more than once, comparing the outputs of these executions, and disregarding non-matching outputs.

Chapter 4 describes a solution, Medusa, for scaling out MapReduce to multiple clouds and, simultaneously, tolerating the new faults introduced by such multi-cloud environment.

Chapter 5 presents a solution, Chrysaor, that is based on a fine-grained replication scheme that tolerates faults at the task level, contrary to Medusa.

Finally, Chapter 6 concludes the thesis and discusses some future work.

2
Background and Related work

This section provides background on the problem at hand, mainly by introducing the necessary concepts and discussing relevant work done in the area. Section 2.1 introduces the MapReduce programming model and implementation as well as different enhancements to the framework that try to improve specific features to solve particular problems or to adapt it to specific environments. In Section 2.2, we outline the challenges of having services running in multiple clouds and describe the frameworks that successfully make use of this environment in order to improve reliability and resiliency. Section 2.3 explores arbitrary fault-tolerant techniques to achieve this level of resiliency. Finally, as the performance of the computations is very dependent on the scheduling algorithms, in Section 2.4 we describe several schedulers for MapReduce and for multi-cloud environments.

2.1 MapReduce and related models

Over the years, many systems have been proposed to parallelize computation automatically (Blelloch, 1989; Gao *et al.*, 2007). One common characteristic of these works was their restricted programming models, which make them impracticable and unsuitable for typical applications.

In 2004, Google presented the MapReduce programming model and implementation for processing large data sets in a datacenter (Dean & Ghemawat, 2004). Since then, MapReduce (MR) has been attracting a lot of interest as a convenient tool for processing massive datasets, as it adapts well to large real-world computations, scaling to thousands of processors and offering fault tolerance for a variety of applications.

A few years later, an implementation of the MapReduce framework was created as an Apache open source project named Hadoop (White, 2009). This distribution is presently used by many cloud computing companies, including Amazon, IBM, RackSpace, and Yahoo!.[1] The popularity of Hadoop MapReduce comes from several reasons. First, it is an open source framework that can process big data in inexpensive commodity servers. Second, it is an easy to use, highly scalable framework. Third, the myriad side projects have turned Hadoop into an ecosystem for storing and processing big data (Lee *et al.*, 2012).

The research community also contributed to the framework popularity by exploring it and publishing various MapReduce algorithms, extensions, and components to solve specific problems or improve performance. Since its origin, it has been used for a variety of tasks, from page ranking (Bahmani *et al.*, 2011) to climate research (Novet, 2013), from genome analysis (Menon *et al.*, 2011) to high-energy physics simulation (Wang *et al.*, 2013). Commercial versions of the framework also started to appear, e.g., Amazon Elastic MapReduce (Amazon, 2015) and Microsoft Daytona (Barga, 2011).

MapReduce is a paradigm that combines distributed and parallel computation with distributed data storage and retrieval. This programming model enables programmers to write distributed applications without having to worry about the underlying distributed computing infrastructure. A user just needs to analyze data that resides in a distributed file system using two types of functions: *map* and *reduce*. MapReduce offers means to handle data partitioning, task scheduling, distributed computation, and fault tolerance in a cluster of commodity servers, such as those available in common cloud computing services. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are a common occurrence and should be automatically handled by the framework. It deals with network communication costs and data localization, which is essential to a good distributed algorithm. The simplicity of the programming model and the quality of services provided by many implementations of MapReduce attracted a lot of enthusiasm among distributed computing communities.

The term Hadoop has come to refer not just to the base modules above, but also the collection of additional software packages that can be installed on top of, or alongside, Hadoop. In this work, we just focused on Hadoop MapReduce.

[1] http://wiki.apache.org/hadoop/PoweredBy

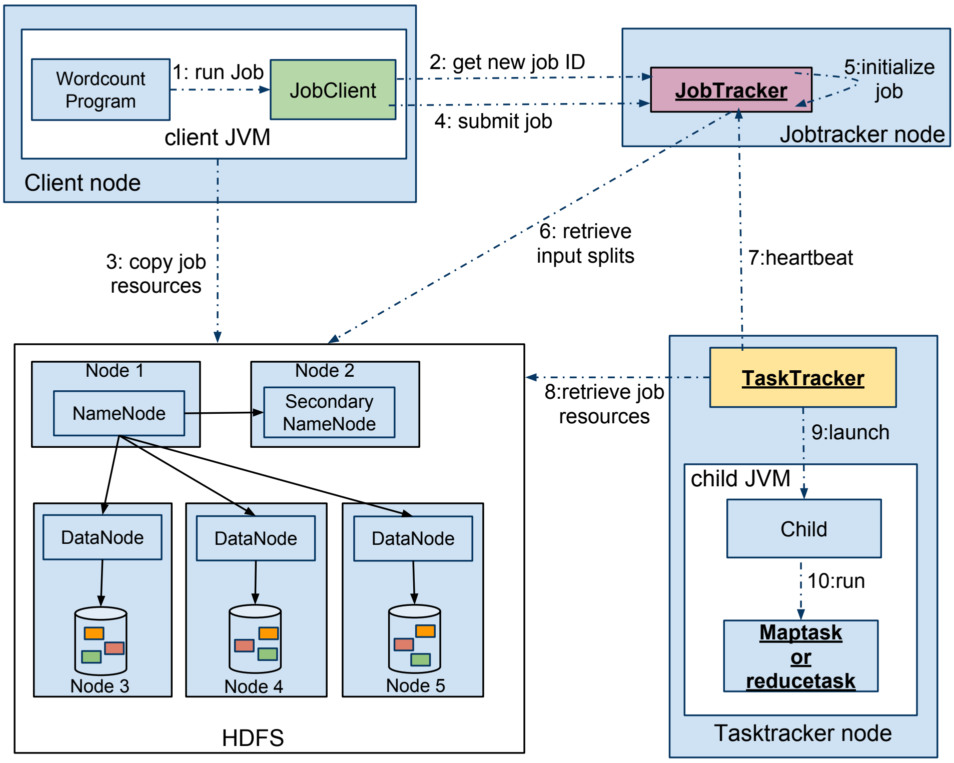


Figure 2.1: Hadoop MapReduce architecture. (White, 2009)

In the following sections (Section 2.1.1 and Section 2.1.2), we detail the architecture of MapReduce and give an example of its execution to help the reader understand the main concepts.

2.1.1 Architecture

Hadoop is an open-source implementation of MapReduce and is the most popular variant currently used by several large companies (White, 2009). We explain Hadoop MapReduce architecture instead of Google MapReduce because the latter is not available, so many details are unclear.

The core of the Hadoop framework consists of a storage part, known as Hadoop distributed file system (HDFS), and a processing part called MapReduce for distributed processing. As illustrated in Figure 2.1, Hadoop consists of a number of different daemons/servers that run inside a single datacenter: *NameNode*, *DataNode*, and *Secondary NameN-*

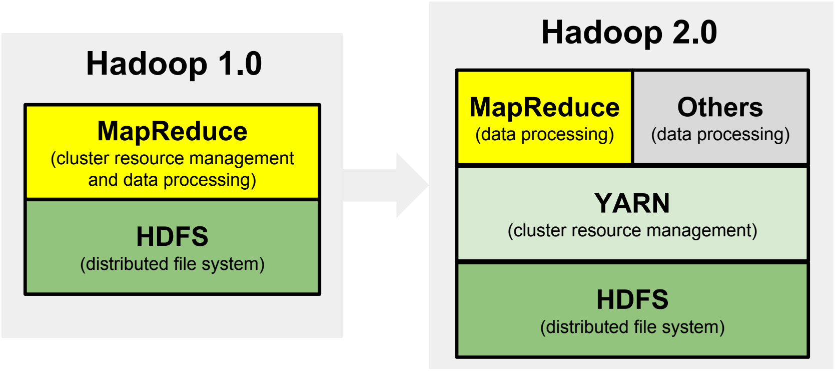

Hadoop 1.0
MapReduce
(cluster resource management and data processing)
HDFS
(distributed file system)
Hadoop 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(distributed file system)

Figure 2.2: Hadoop 1.X vs Hadoop 2.X.

ode for managing HDFS; and *JobTracker* and *TaskTracker* for performing MapReduce.

Initially, Hadoop was presented as version 1.X. In this version, there was only a MapReduce framework that worked on top of HDFS (see Figure 2.2). A new Hadoop architecture was created in 2013 – Hadoop 2.X – to improve the performance for specific applications, to support additional processing models, and to implement a more flexible execution engine. The new version introduced a new component called YARN, which is a resource manager that separates the processing engine and resource management capabilities of MapReduce. YARN is often called the operating system of Hadoop because it is responsible for managing and monitoring workloads, maintaining a multi-tenant environment, implementing security controls, and managing high availability features of Hadoop. The idea is to have a global *ResourceManager* and a per-application *ApplicationMaster*. The ResourceManager and the ApplicationManager replace, respectively, the JobTracker and the TaskTracker in the new version. However, in general, they perform the same roles.

In Hadoop 1.X, the JobTracker is responsible for both managing the cluster's resources and driving the execution of the MapReduce job, and the TaskTracker is responsible for launching and managing map and reduce tasks in a server. In Hadoop 2.X, the role of the JobTracker was divided into two separate components: resource management and job scheduling/monitoring. The ResourceManager is responsible for the launching and managing of applications. The role of the TaskTracker was replaced by the ApplicationManager which is responsible for maintaining a collection of submitted applications. In the case of MapReduce, it is capable of creating and managing containers where map and reduce tasks will run.

In the rest of this section, we describe the two most crucial components for our work: *HDFS* and *MapReduce*.

HDFS is a file system designed for storing very large files and optimized for streaming access patterns. Since it is expected to run on commodity hardware, it aims to take into account and handle failures on individual machines. As it is optimized for streaming access of large files, random access to parts of files is significantly more expensive than sequential access, and there is also no support for updating files, only append is possible. The typical scenario of applications using HDFS follows a write-once read-many access model.

Files in HDFS are split into many large blocks (with size usually a multiple of 64 MB) which are stored on DataNodes. A file is typically distributed over a number of DataNodes to facilitate high bandwidth and parallel processing. Data blocks in HDFS are replicated and stored on three machines by default to improve reliability, with one of the replicas in a different rack for increasing availability further. A separate NameNode handles the maintenance of file metadata. Such metadata includes mapping from file to block, and location (DataNode) of the block. The NameNode periodically communicates its metadata to a Secondary NameNode which can be configured to do the task of the NameNode in the case of the latter's failure.

MapReduce: In Hadoop, the JobTracker is the access point for clients. The duty of the JobTracker is to ensure fair and efficient scheduling of incoming MapReduce jobs and assign the tasks to the TaskTrackers which are responsible for execution. A TaskTracker can run many tasks depending on available resources and will allocate a new task sent by the JobTracker when ready. The relatively small size of each task compared to a large number of tasks in total helps to ensure load balancing among the machines. It should be noted that while the number of map tasks to be performed is based on the input size (number of splits), the number of reduce tasks for a particular job is user-specified and defined in a configuration file.

In a large cluster, machine failures are expected to occur frequently, so regular heartbeat messages are sent from TaskTrackers to the JobTracker. In this way, failures can be detected, and the JobTracker can reschedule the failed task to another TaskTracker. Hadoop follows a speculative execution model for handling failures. Instead of fixing a failed or slow-running task, it executes a new equivalent task as a backup.

Failure of the JobTracker itself is not handled, being a single point of failure. If it goes down, all running jobs are halted.

2.1.2 Fundamentals of MapReduce

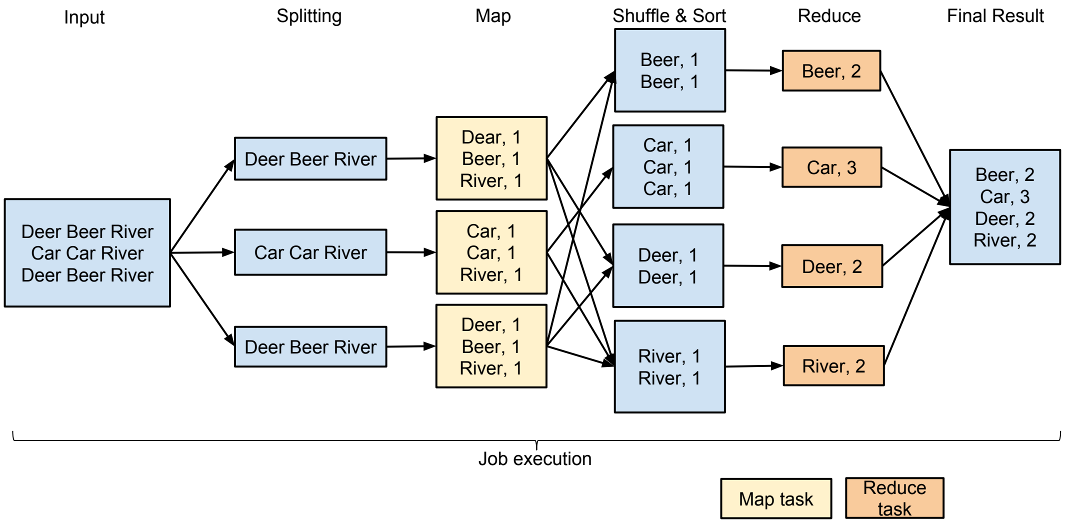


Figure 2.3: Example of WordCount execution in Hadoop MapReduce. (White, 2009)

In Hadoop, a user writes an application that consists of a `map` and a `reduce` function, that will be used by the framework to process data in-parallel on large clusters. In this section, we will use the WordCount application as an example (Fig. 2.3). WordCount is a standard application that counts word occurrences from an input file. A representation of the example is depicted here as a diagram to help understand a job execution.

A MapReduce *job* is characterized by two phases: the *map* phase that is specified by a `map` function; and the *reduce* phase which processes these map results and produces a final output. A `map` function takes key/value pairs as input, performs some computation on this input, and produces intermediate results in the form of key/value pairs. The output of the computation are *shuffled and sorted* before the reduce phase. The shuffle step can be more time-consuming than the other stages depending on network bandwidth availability and other resources. In more detail, considering the WordCount example, when the client submits a job, the MapReduce framework starts an execution that consists of the following six steps:

Input reader: The input reader in its basic form takes input from files (large blocks) and divide them into appropriate *splits*. It is possible to add support for other input types so that input data can be retrieved from a database or even from main memory. The data are divided into *splits* which are the unit of data processed by a map task. A typical split size is the size of an HDFS block, which is 64 MB by default, but this is a configurable parameter. Suppose we want to store 1GB of data in HDFS. The data are split into 1GB/64MB=16 split/blocks that will be distributed across the DataNodes. These blocks will reside on different DataNodes, based on the cluster configuration. The number of map tasks or *mappers* in a MapReduce program is defined by the number of splits. In our example, the input is divided into three blocks containing part of the data, and each block is going to be processed by a mapper.

Map function: A map task takes as input a key/value pair from the input reader, performs the logic of the `map` function on it, and outputs the result as key/value pairs. The results from a map task are initially output to main memory, but when it reaches a certain limit, it spills the data to the local disk. The spill files are in the end merged into one sorted file. In our example, the three mappers produce intermediate data that will contain pairs where for each word is attributed the value 1.

Shuffle & Sort: Shuffling is the process of transferring data from the mappers to the reducers. Without shuffling, reduce tasks would not have input data. Prior to the shuffling, a partition function has the goal to determine to which reducer a key-value pair will go. In this example, it is used a hashing function that creates unique subsets of the same word from the map output for each reducer. The sorting consists in grouping and sorting the values by key. Each reduce task will sort the data when the shuffling end. The sorting happens before starting the reduce phase.

Reduce function: The reduce function is invoked once for each distinct key and is applied on the set of associated values for that key, i.e., the pairs with the same key will be processed as one group. The input to each reduce task is guaranteed to be handled in increasing key order. It is possible to provide a user-specified comparison function to be used during the sort process. In our example, the goal of the reduce function is to sum all occurrences of each unique word.

Output writer: The output writer is responsible for writing the output to stable storage, which can be a database or a file system. In Hadoop, the output of the reduce tasks is stored in HDFS by default. The user can configure the platform to store the output in the local disk or in a database.

2.1.3 Enhanced MapReduce frameworks

Different lines of work try to improve specific features of the framework to solve particular problems or to adapt it to specific environments. In this section, we present related work on trying to enhance MapReduce characteristics.

Hadoop MapReduce relies on shuffle and sort algorithms for grouping map outputs. As a consequence, `map` and `reduce` functions are blocking in that all tasks should be completed to move forward to the next stage or job. This property causes performance degradation. MapReduce Online is an alternative solution that supports online aggregation (Condie *et al.*, 2010). The authors modified MapReduce for the map tasks to push data periodically to the reduce tasks. As result, pipelining the batch processing resulted in better performance. Similar to MapReduce Online, Li *et al.* have adapted MapReduce for I/O intensive operations (Li *et al.*, 2011). This solution uses hash tables for better performance and for incremental processing. As soon as the intermediate data is produced, results are pushed to buckets that are consumed by reducers. Reducers perform the aggregation in each bucket that is stored with a unique key while the map output is being produced.

MapReduce is commonly used in dedicated computing environments where a fixed number of slave nodes and a master node are configured in the platform. By contrast, MOON is a system designed to support MapReduce jobs on opportunistic environments, like volunteering computing (Lin *et al.*, 2010). MOON extends Hadoop with adaptive task and data scheduling algorithms to offer reliable services on a hybrid resource architecture. The adaptive task and data scheduling algorithms in MOON distinguish between various types of MapReduce data and node outage to place tasks and data on both volatile and dedicated nodes. Similarly to MOON, the goal of P2P-MapReduce is to investigate how to modify the master-slave architecture of current MapReduce implementations to make it more suitable for Grid and P2P-like dynamic scenarios characterized by high churn rate (Marozzo *et al.*, 2008, 2012). The authors extended the MapReduce architectural model making it suitable for highly dynamic environments where failure must be managed to avoid a critical waste of computing resources and time. P2P-MapReduce adopts a peer-to-peer model where a set of nodes can act as master or slaves at each time for load balancing purposes.

Iterative MapReduce has also been studied for hybrid cloud environments (Chu *et al.*, 2006; Clemente-Castello *et al.*, 2017). In this work, the authors combined several MapReduce data locality techniques to achieve scalability for iterative applications. The results showed that it is feasible to obtain a speedup for applications that scale on-premise with specialized data locality strategies. Enforcing the exact locality in the scheduling policy can compete against the performance levels of similar single-site Hadoop deployments.

Phoenix is an implementation of MapReduce developed on top of *p-threads* that targets shared-memory systems such as multi-core chips and symmetric multiprocessors (Ranger *et al.*, 2007). The authors use thread and shared-memory buffers to facilitate communication between tasks and minimize the overheads of task spawning and data communication. Phoenix leads to similar performance for most applications when compared with the same application built with *p-threads* while offering a high level of abstraction. Alternatively, LEMO-MR is an optimized implementation of MapReduce for both on-disk and in-memory applications (Dede *et al.*, 2014; Fadika & Govindaraju, 2010). The LEMO-MR is an elastic approach to MapReduce where a cluster can be scaled out without the need to restart the system. This idea contrasts Hadoop MapReduce implementation. The master node can choose which workers will be mappers and reducers, and the work is sent from the master node. In case of a failure, the communication pipe is broken, and the master node can detect who has failed. The LEMO-MR evaluation showed that it could perform much better when running with few workers, but as the number increases, Hadoop MapReduce presents similar results.

Another significant trend is adapting MapReduce for scientific computing, e.g., for running high energy physics data analysis and K-means clustering (Cui *et al.*, 2014), and for the generation of digital elevation models (Krishnan *et al.*, 2010).

Yahoo! Research developed a platform for executing chains of MapReduce programs used with Hadoop called Pig (Gates, 2011). The design goal of Pig is to be appealing to experienced programmers for performing ad-hoc analysis of huge datasets. For this framework, the MapReduce paradigm is too low-level and rigid, and leads to a significant amount of custom user code that is hard to maintain, and reuse. Consequently, the new language called Pig Latin combines a declarative language style similar to SQL with the procedural programming of MapReduce and has several features to help a programmer to achieve efficiently parallel execution of data analysis tasks (Grolinger *et al.*, 2014; Olston *et al.*, 2008).

In contrast, Osprey is an SQL system that implements a MapReduce strategy to query data replicated on different databases (Yang *et al.*, 2010). It does so by splitting running queries into *sub-queries* and executing them in parallel. The intermediate results of the sub-queries are merged before the result being sent to the client. Similarly to MapReduce, in a case of a fault, Osprey uses re-execution to minimize the effect of the "stragglers".

MapReduce is suitable to execute easily a wide range of applications using distributed computing, but it is inefficient implementing others (Bu *et al.*, 2010; Ekanayake *et al.*, 2008; Power & Li, 2010). Alternatives to the Hadoop framework try to enhance MapReduce characteristics to enable complex interactions, improve efficiency, and provide a higher level of abstraction (Alexandrov *et al.*, 2014). This is useful when we want to deal with second-order functions (Battré *et al.*, 2010), or combine communication channels with computational vertices to form complex data flow graphs (Gonzalez *et al.*, 2014; Isard *et al.*, 2007).

In 2010, Microsoft lacked a distributed computing framework that could run in Microsoft Azure cloud service [1] such as MapReduce. It only supported basic queue-based job scheduling. At that time, Microsoft introduced a novel MapReduce runtime called Azure MapReduce that improves the original MapReduce in several aspects (Gunarathne *et al.*, 2010). The Azure MapReduce is a decentralized platform that does not contain a master node, avoiding that single point of failure. In addition, it allows to scale up or down the number of computing nodes even in the middle of computation. The Azure MapReduce leverages these characteristics to provide an efficient on-demand alternative to traditional MapReduce clusters.

Twister, a lightweight — yet enhanced — distributed in-memory MapReduce, expands the applicability of the framework to classes of applications that need iterative computations, such as data clustering and machine learning (Ekanayake *et al.*, 2010). Twister uses publish/subscribe messaging (pub/sub) for communication and data transfer, facilitating multicast and broadcast messaging. In contrast, all communication in Hadoop MapReduce is point-to-point. Twister also provides fault tolerance for iterative MapReduce computations. The framework saves the application state of the computation on every iteration so that, in the case of a failure, the entire computation can roll back to the last correct state.

[1] The Microsoft Azure infrastructure is a cloud computing platform created by Microsoft for building, deploying, and managing applications and services through a global network of Microsoft-managed datacenters.

The evaluation results have shown that Twister performs and scales well for many complex iterative MapReduce applications.

Dryad is an alternative solution to MapReduce developed by Microsoft Research (Isard *et al.*, 2007; Yu *et al.*, 2008). Dryad also provides reliable, distributed computing on thousands of servers for large-scale data parallel applications, but, differently from MapReduce, it allows to create more complex data flow graphs, meaning more than a single map and reduce iteration in the flow. This framework allows communication between stages to happen over more than just files stored in the distributed file system, by means of sockets, shared memory, and pipes to be used as channels between elements. It ultimately ends up looking like a directed acyclic-graph (DAG) of user-defined elements. Data flows between elements over a choice of channels, and the elements are all user-defined, bringing the benefit of more efficient communication, the ability to chain together multiple stages and express more complicated computation. This solution is useful to deal with problems that are difficult to solve in a large-scale and concurrent way, such as data-mining applications, image and stream processing.

CIEL is a similar solution to Dryad, which can also make data-dependent control flow decisions to compute iterative and recursive algorithms (Murray *et al.*, 2011). This solution could offer a more powerful programming model and execution engine than Dryad. What differentiates CIEL from Dryad is that this framework can provide dynamic task graphs and control flow requiring low-overhead runtime approaches to not only schedule tasks but also to reschedule tasks when failures occur. CIEL tolerates faults by re-executing those that have failed or stopped responding in the worker nodes. When a failure happens in the master node, CIEL launches a new master to resume execution immediately.

Another framework that improves over MapReduce is Nephele. This framework exploits the dynamic resource allocation offered by clouds for both task scheduling and execution. The need to create more complex data flows led its authors to propose a parallel data processor centered around a programming model of so-called Parallelization Contracts (PACTs) (Battré *et al.*, 2010; Warneke & Kao, 2009). The PACT programming model is a generalization of the map/reduce programming model, extending it with further second-order functions, useful for relational query processing or data mining. Similar to Dryad, Nephele considers incoming jobs to be directed acyclic graphs (DAGs) with vertices being sub-tasks and edges representing communication channels between these sub-tasks. During execution, Nephele allocates resources dynamically in order to match the workload, and allows communication between sub-tasks using network, in-memory, and file channels.

This impressive list of works (and we are only touching the surface) show the importance of the MapReduce programming model to solve complex applications from different areas, but also its limitations and proposals to address them. However, none of them improves the original MapReduce regarding fault tolerance, namely, to tolerate arbitrary faults and cloud outages.

2.2 Cloud-of-clouds systems and applications

The increasing maturity of cloud computing technology is leading many organizations to migrate their systems and applications to operate entirely or partially in the cloud. However, existing cloud computing services are today problematic for running critical applications, which may range from business-critical tasks of companies to mission-critical tasks for the society. Protecting data and services in the cloud is a challenge of increasing importance for governments and organizations across all industries, including healthcare, energy utilities, and banking. Delivering reliable and resilient computing services traditionally implies having trustworthy components that operate inside the cloud infrastructure. The purpose of these components is to achieve higher security and better resilience than current cloud computing services may provide.

Regarding data availability and privacy, current cloud computing systems involve the disadvantage that users do not know where their data is stored or how it is processed. The trust put on cloud providers may be unjustified due to a set of threats that may affect the services, like the loss and corruption of data (Cloud Security Alliance, 2013). Indeed, there are several cases of cloud services losing or corrupting data. In 2009, Sidekick, a company acquired by Microsoft, suffered a data outage that resulted in the loss of contacts, calendars, and personal information of an estimated 800 thousand users (Williams, 2009). The incident caused a public loss of confidence in the concept of cloud computing and Steve Ballmer, CEO of Microsoft, said that it was “not good” for Microsoft.

Another wide-spread security problem that affects clouds is malicious insiders. A cloud insider can be a rogue administrator of a service provider or an insider that exploits a cloud-related vulnerability to steal information or to carry out an attack (Claycomb & Nicoll, 2012; Hanley *et al.*, 2011; Jones, 2017; Narayan, 2016). A security survey on clouds was performed in 2014 and showed that 20% of organizations believe malicious insiders pose the biggest threat to business security, and 44% suggest employees' ignorance could also cause cloud security to fail (AppRiver Guest Blog, 2014).

The current vulnerabilities that affect cloud computing have lead to the *cloud-of-clouds* paradigm introduced in (Bessani *et al.*, 2011). The approach of moving to multi-clouds was employed in the TClouds project (Veríssimo *et al.*, 2012) with a goal to improve resilience against attacks and accidents while preserving the ability for the user to interact with multiple clouds without any predefined coordination between cloud providers. The TClouds platform integrates multiple advanced security technologies in a standard cloud distribution and in commercial cloud systems to add resilience to the infrastructure and guarantee the integrity of a hardened cloud computing platform to users of these services. Cloud applications can run on the top of the TClouds architecture benefiting from the resilience against arbitrary faults, and from the additional resources made available by using a multi-cloud architecture.

A significant contribution from the TClouds project was DepSky. DepSky is a dependable and secure storage system that tackles the vulnerabilities of traditional cloud storage by providing a system that improves the availability, integrity, and confidentiality of data stored in the cloud, by replicating data on different cloud providers (Bessani *et al.*, 2011). DepSky ensures confidentiality of the data stored on the clouds using a secret sharing scheme. This scheme prevents the data to be disclosed with less than $f + 1$ different shares of the secret. As a result, intruders cannot disclose the data if they have compromised f clouds.

2.2.1 Cloud-of-clouds MapReduce

The cloud-of-clouds model arises from two key limitations that affect cloud services: a single cloud may not have enough resources to fulfill all tenant's requirements, and a single cloud is, by definition, a single point of failure. The growing interest in a multi-cloud environment led to the establishment of a federation of collaborated clouds for improving the quality of service. A cloud-of-clouds model entails several benefits, such as the mitigation against cloud-scale disasters, cost reduction by taking advantage of different cloud pricing schemes, among other advantages that arise from reducing reliance on a single provider. Despite the advantages of having a cloud-of-clouds model, the challenge is on materializing it in the real world. It is necessary to find the right abstractions that enable an efficient and transparent interoperation between different clouds, extending the level and the types of service offered to tenants (Group, 2011). Moreover, current solutions do not support a coordinated distribution of the workload into different clouds.

The high data-growth and the multiplication of cloud offers have lead to user's data to be spread across multiple cloud providers. With this new setting, it comes the need to analyze data stored by different applications in different clouds, but that can be burdensome. In fact, scientific applications, click-stream analysis, social networking, and many other applications usually have several distributed data sources with the resulting data collected and spread in separate locations, even across the Internet.

Hadoop MapReduce is designed based on the assumption that there is a centralized master responsible for resource management, and it is assumed that all nodes in the cluster are connected and data is distributed among them. This assumption is not entirely congruent in a scenario in which data resides in a highly distributed environment, and, as such, this framework lacks support for general data analytics for highly distributed infrastructures.

HDM-MC is a recent solution for this problem (Wu *et al.*, 2017). It is a multi-cluster big data processing framework, which is designed to enable the capability of performing large scale data analytics across multi-clusters with minimum extra overhead. This solution is designed to support coordination and execution of general big data applications across multi-clusters infrastructures, but does not target MapReduce.

G-MR is a Hadoop-based framework that can perform a sequence of MapReduce jobs on geo-distributed data across multiple datacenters (Jayalath *et al.*, 2014). G-MR determines an optimized path to carry out a sequence of MapReduce jobs and uses Hadoop MapReduce clusters deployed in each datacenter. G-MR aggregates the results obtained from the several job executions with *aggregators*. Once the output data is generated in one or more datacenters, it is copied to a single destination where it will initiate the aggregate operation and return the final result. G-MR was evaluated against common and naïve deployments for processing geo-distributed datasets, and the results show that using G-MR improves processing time and cost for geo-distributed datasets. G-MR optimizes the data movement, but this approach is highly complex and does not support complicated job sequences well.

A similar approach is G-Hadoop, a solution that explores G-farm, a file system similar to HDFS, with the difference that G-farm can federate local disks of network-connected PCs and compute nodes in several clusters (Mikami *et al.*, 2011; Tatebe *et al.*, 2010). G-Hadoop is another MapReduce framework that aims to enable large-scale distributed computing across multiple clusters, by replacing HDFS with Gfarm (Wang *et al.*, 2013). Thus, users can submit their MapReduce application to G-Hadoop, which executes map and reduce tasks across multiple clusters with the help of Gfarm. Although Gfarm can deal with files stored in several clusters, it uses some strategies to reduce the added latency when compared with HDFS. As a result, Gfarm presents on average similar results to HDFS.

Resilin is an Elastic-MapReduce system that enables the execution of MapReduce computations across multiple clouds besides Amazon EC2 (Iordache *et al.*, 2013). Resilin supports multi-cloud job flows by allowing users to dynamically add new resources from different clouds to a running job flow. This system provisions Hadoop VMs when it receives a user request to submit a MapReduce job flow, and it can commission nodes in the cluster to scale up and down the service. When scaling up, Resilin starts the VMs and configures the Hadoop services to join the cluster. When scaling down, the solution decommissions the Hadoop services when all running tasks have finished and if there is enough disk capacity in the remaining services. The system also monitors several services to enable fault tolerance to the VMs and request distribution of the work among several instances in order to prevent individual services from being overloaded. A full MapReduce execution with Resilin includes: *(1)* the transfer of the input data from the cloud storage (e.g., S3) to HDFS; *(2)* execution of the MapReduce application; *(3)* transfer the output data from HDFS to the cloud storage. The S3 filesystem can be used as a replacement for HDFS or as a repository to transfer data between several HDFS filesystems. As S3 does not guarantee data locality when it is used as a replacement for HDFS, it results in some degradation of the performance of MapReduce jobs. Overall, the experiments showed a performance degradation when using a platform built from resources provided by multiple clouds.

Unlike our work, these alternative solutions do not address the MapReduce dependability limitations — inability to deal with arbitrary faults and cloud outages — and most of them do not attempt to be *transparent* to users.

2.3 Arbitrary fault tolerance

Arbitrary faults, also known as Byzantine faults, are the most serious type of fault that affects a distributed system. An arbitrary fault can be transient or permanent, such as a bit flip or a stuck-at bit. A system that suffers an arbitrary fault can produce an output that should never be produced. Worse yet, a faulty server can produce intentionally incorrect outputs due to a malicious action.

Algorithms to tolerate Byzantine faults were first introduced 35 years ago with the objective of enabling a system to defend against arbitrary behavior. The goal of these algorithms is that all participants have a globally consistent view of the system. The term *Byzantine* refers to an agreement problem designated as the Byzantine Generals' Problem (Lamport *et al.*, 1982). Achieving agreement, or consensus, can be troublesome when we are in the presence of traitorous generals (faulty nodes).

In Byzantine fault-tolerant algorithms, it is up to all parts of the system to agree on a correct value. The key technique to control faults is to use redundancy and accept the outputs of the majority. Although there are solutions, Byzantine faults are often ignored due to the difficulty to address them, and the fact that they are incorrectly assumed to occur with zero or very low probability. However, the explosion of commercial off-the-shelf technology and the increasing number of critical systems show the opposite. Large services have been disrupted by arbitrary faults, resulting in long periods of unavailability (Amazon S3, 2011; Anderson, 2017), and the hardware error rate is expected to increase with the upcoming hardware generations (Borkar, 2005; Qiang Wu & Mutlu, 2015).

Unfortunately, practical fault-tolerant distributed systems such as MapReduce are not built to tolerate arbitrary faults (Dean & Ghemawat, 2004). As this system is usually deployed on commodity hardware (Barroso & Hoelzle, 2009), because hardware fault-tolerant computers are expensive and usually an order of magnitude slower than commodity hardware (Faccio, 2011), so the distributed algorithm it implements is designed to tolerate benign faults only, e.g., process crashes and message omissions.

Solutions that are Byzantine fault-tolerant involve consensus to provide coherence among and between nodes. Consensus is a classical distributed systems problem that involves an agreement among a number of processes to decide for a single data value. The consensus problem is deeply connected to the state machine replication (Schneider, 1990) and atomic broadcast algorithms (Correia *et al.*, 2006; Hadzilacos & Toueg, 1994). One way of achieving consensus is using voting to obtain a quorum. Consensus, state machine replication, and quorum are methods that are used to develop Byzantine fault-tolerant systems.

A Byzantine failure is the loss of a system service due to a Byzantine fault. Many algorithms that helps multiple processes maintain a consistent state in a model where failures can happen involve solving a consensus problem. The original paper discussed the problem of consensus in the context of unauthenticated and authenticated messages (Driscoll *et al.*, 2003; Lamport *et al.*, 1982). For unauthenticated messages, it is concluded that to tolerate f Byzantine faults, it is necessary $3f+1$ processes using $f+1$ rounds of message exchange. For authenticated messages, consensus is possible with $2f+1$ processes using the same number of rounds. Since the possibility of implementing efficient Byzantine fault-tolerant (BFT) replication was demonstrated in (Castro & Liskov, 1999), several algorithms appeared (Amir *et al.*, 2006; Bessani *et al.*, 2014; Clement *et al.*, 2009b; Veronese *et al.*, 2010, 2013) that tolerate f faults with different number of replicas and communication rounds, for various system models.

Quorum systems are used to ensure the consistency and availability of replicated data. A *quorum* is obtained from a minimum number of votes that any given operation must obtain to be performed and have to be defined in such way that conflicting operations always intersect some replicas. Several algorithms rely on quorums and exploit the fact that two (read and write) quorums overlap in at least one process. For instance, consider a system with N processes, serving as a distributed replicated database. There is a single process writing to the database and there can be multiple readers. One approach to doing this would be to have the writing process write to at least $\lceil N/2 \rceil + 1$ replicas. If the reading process reads from at least $\lceil N/2 \rceil + 1$ nodes, there will be at least one replica which has seen the latest write. This is a simple quorum example that can be used in scenarios where only less than half the nodes can fail. The key idea is that the intersection property guarantees that operations done on distinct quorums preserve consistency.

A weighted voting scheme can be used for maintaining replicated data (Gifford, 1979). Every copy of a replicated file is assigned some number of votes for read and write transactions. With weighted quorums, if N is the total number of votes assigned to all nodes, then a quorum of r votes is required for a read operation, and a quorum of w votes is required for a write operation. The quorum values are such than $r+w > N$ and $2w > N$. Therefore, there is always a representative quorum to guarantee any transaction. The voting scheme has a high communication cost as a request to an operation is typically sent to all nodes. Other solutions try to reduce the size of the quorums to reduce the communication cost. For instance, the hierarchical quorum system proposed by Preguica & Martins (2001) is based on an n-ary tree construction in a triangle shape to present better availability and load in grid-based constructions. As result, the system requires a quorum smaller than the quorums average size. It also showed an improvement in the load and availability with results close to the optimal load.

In a system prone to arbitrary faults, two majority quorums may not intersect in a correct process due to arbitrary behavior. Therefore, a majority achieved in a *Byzantine quorum* tolerating f faults happens when there is a set of more than $(N+f)/2$ processes (Guerraoui & Rodrigues, 2006). Byzantine quorum systems are used to implement data stores with several concurrency semantics (Alvisi *et al.*, 2001; Malkhi & Reiter, 1997), even in the cloud (Bessani *et al.*, 2011). These solutions are inadequate to implement an arbitrary fault-tolerant MapReduce for the simple reason that the framework is not a storage service, but a computational system.

One of the initial works on BFT for task processing is due to Sarmenta (Sarmenta, 2002). He proposed a voting mechanism for sabotage-tolerance in volunteering computing. This scheme estimates the credibility of results and of individual nodes as the probability of being correct given a computed result. The author considers two mechanisms: *(i)* the identification of malicious nodes is done by verifying if the computed result is correct — this helps estimate the nodes reliability and exclude faulty nodes from the computation; *(ii)* in order to increase confidence in a result, it computes a task redundantly until a certain number of results agree. Overall, Sarmenta wants to avoid more than a number of false results to be obtained during computation.

To the best of our knowledge, only a couple of proposals deal with BFT data processing in cloud contexts: CloudBFT and ClusterBFT. CloudBFT was created to evaluate the feasibility of a Byzantine fault-tolerant architecture in cloud computing environments. This architecture is designed for critical applications that tolerate up to f physical or virtual machine failures (Nogueira *et al.*, 2014). It uses redundant virtual machines placed on different physical machines, availability zones, or regions to avoid common faults and guarantee correct computation. This platform uses the MinBFT algorithm to achieve Byzantine fault-tolerance (Veronese *et al.*, 2009), uses a Trusted Platform Module (TPM) to sign and verify messages, and allows to create groups composed of $2f+1$ replicas to tolerate a fixed number of f Byzantine failures, plus another machine to order the requests. The proposed design uses a relational database to have synchronization among different replicas to guarantee totally ordered requests.

Although CloudBFT could be used to implement a BFT MapReduce, it would require a heavy execution based on the necessity to launch $2f+1$ replicas. Moreover, ordering requests would also increase the cost.

A different system for Byzantine fault-tolerant data-flow processing in clouds is ClusterBFT (Stephen & Eugster, 2013). ClusterBFT models jobs as DAGs, similarly to Dryad and Nephele (Section 2.1.3), allowing the creation and replication of sub-graphs. At the end of execution, the system produces a digest of each output and verifies that $f+1$ executions produced the same result. The use of sub-graphs has the intention to reduce overhead and improve utilization by preventing the whole graph to be replicated, and only the output of the last step being validated. When a fault happens — when the platform receives a wrong digest or does not receive a digest from nodes executing the data-flow — the faulty nodes will be isolated, and the sub-graph will be rescheduled. This system secures computation in the cloud leveraging BFT replication with fault isolation, but disregarding the case an adversary manipulates the cloud service provider to the point of provoking cloud outages. Moreover, the system is deployed on a cloud service that leases out virtual machines to users. Particularly, each physical machine in the cloud can host multiple virtual machines.

State machine replication is not adequate to make MapReduce Byzantine fault-tolerant because the cost would be high, *i.e.*, 3f+1 replicas of the job would have to be executed. ClusterBFT acknowledges this evidence and uses graphs to avoid replicating the whole execution in several nodes. This solution is restricted to a single cloud, and thus it does not tolerate cloud outages or attacks from malicious insiders.

2.4 Scheduling

Scheduling is the process of assigning work to resources, which is a fundamental problem when it is necessary to share resources among different users. The utility of a scheduling algorithm depends on its requirements. In MapReduce, a scheduler chooses the order in which a series of jobs are executed and how the tasks will be distributed. When one considers a multi-cloud environment, a scheduler also needs to deal with the heterogeneity of the cloud resources, with its varying capacities and functionalities to distribute work. In summary, different applications have schedulers with different purposes. In the following sections, we present work on scheduling for Hadoop MapReduce (Section 2.4.1) and other distributed environments (Section 2.4.2).

2.4.1 MapReduce Scheduling

Hadoop MapReduce was designed mainly for running large batch jobs. For this purpose, when users submit jobs, they are added to a queue, and the Hadoop scheduler will run them in order, i.e., following a FIFO order. The main advantage of Hadoop's FIFO scheduler is the simplicity in assigning resources to jobs.

In case several jobs are waiting to execute, and there are still resources available, the scheduler launches the next job according to the FIFO order. When no spare capacity is available, the waiting jobs stay in the queue until resources are freed. However, this default scheduler is not very efficient, especially, when skew occurs. For example, if a job has a map or reduce task that is a straggler, then the whole system might halt. Similar to the case with tasks, a high-priority job can still be blocked by a long-running low priority job that started before. Fortunately, Hadoop has a speculative mode of execution that launches new tasks at other nodes when stragglers are detected. Ultimately, when we need to share resources to provide an assured capacity to production jobs and good response time, new scheduling algorithms are necessary.

Unlike the FIFO scheduler, Hadoop's Fair scheduler assigns an equal share of resources to every user over time. This scheduler organizes jobs into sets and divides resources between them (Apache, 2013). As a consequence, it lets short jobs finish in reasonable time while not starving long jobs. Over time, this scheduler has grown in functionality to support hierarchical scheduling, preemption, and multiple ways of organizing jobs.

These two schedulers assume that a cluster is built with homogeneous servers. In other words, they assume that all nodes in the same cluster have the same characteristics, and, as such, the assumption is that the computation will be done at the same rate. This assumption is problematic since, given the large size of clusters dedicated for data-intensive applications, it is likely that these clusters are comprised of different generations of server platforms (Nathuji *et al.*, 2008). In fact, the heterogeneity of servers that are inside datacenters provides diversity on performance needs, ranging from real-time to batch computation. For instance, Amazon EC2 had 2–3 generations of hardware in 5–7 years (Amazon, 2017). Unfortunately, FIFO and Fair Scheduling offer poor performance in these heterogeneous environments. With this motivation, new work started to appear to adapt MapReduce to more diverse cluster configurations.

LATE was the first work to focus on the heterogeneity of the clusters (Zaharia *et al.*, 2008). This algorithm prioritizes tasks based on how much they will delay the execution, selecting the faster nodes and capping speculative tasks to prevent trashing. As a result, LATE performs significantly better than Hadoop's default speculative execution algorithm.

LATE mainly focuses on the detection and mitigation of stragglers, but does not look to its cause. Recently, Zhang *et al.* proposed a model-based optimization, MrHeter, that efficiently allocates tasks between heterogeneous nodes in order to improve performance of job execution (Zhang *et al.*, 2016). Unreasonable task allocation between different nodes in the map tasks increases the execution time of the shuffle and sort stages, and creates a bottleneck in the network due to the necessity to transfer the map output to perform the shuffle stage. An optimal allocation of the map tasks will make them finish around the same time making the shuffle and sort stage start sooner. Similarly, an optimal allocation of reduce keys avoids the necessity of creating stragglers and balance execution time between heterogeneous nodes. MrHeter uses a knowledge base that is built from previous executions to estimate the execution time for the map, shuffle, and reduce stage for each node, allowing to build an optimal allocation scheduler. As result, MrHeter works more efficiently than the original Hadoop, especially in environments of heavy-workloads and large difference of computational power between nodes.

In another work, Verma *et al.* considered performance modeling of MapReduce environments through a combination of measurement, simulation, and analytical modeling for enabling different service level objectives (Verma *et al.*, 2011, 2014). The authors use job profiling to estimate the resources required for processing a certain job based on past job executions. The profiling decides whether it is necessary to add additional resources or adjust the scheduling to guarantee termination of the execution within the specified time constraints. The profiling model predicts the job completion time based on the job profile, input data size, and allocated resources. Normally, it is a user decision to estimate the number of resources needed to perform a task within a time, but in this case, it is the model that takes this decision. In the event of failures, this algorithm quantifies the impact of the failure on the job completion time to respond correctly.

We have witnessed the emergence of more complex MapReduce workloads that are composed of interconnected jobs. Still, in these cases, it is necessary to have scheduling algorithms for flows of MapReduce jobs. FlowFlex attempts to optimize some metric based on the completion times of the flows and resource analysis (Nagarajan *et al.*, 2013). The goal is to minimize either the total cost or the maximum cost of the flows concerning performance. In the end, FlowFlex showed better performance when compared with other standard schedulers like the Fair and FIFO schedulers.

In conclusion, with the recent progress on scheduling, Hadoop started to consider to work in heterogeneous environments or to allocate tasks optimally without incurring the risk of losing performance.

2.4.2 Scheduling in Distributed Environments

The concept of connecting resources together (meta-computing) has been studied extensively, especially in the area of grid computing. For the majority of meta-computing systems, scheduling is not a particular problem, but a set of complex problems (Christodoulopoulos *et al.*, 2009; Xhafa & Abraham, 2010), due to the different requirements and the many characteristics of each group of resources. In a multi-cloud environment, the complexity of scheduling is increased due to the nature of the environment, which may scale dynamically, the heterogeneity of the resources, and the resource sharing based on service level agreements (SLAs). Moreover, providers are constantly changing, which can affect the availability of the resources.

Local and meta-schedulers are two fundamental solutions for scheduling in large-scale distributed systems, such as grids or clouds. Local schedulers are used at the cluster level to manage resources and achieve better load balancing (Buyya *et al.*, 2010), while meta-schedulers organize multiple resource managers into a single view allowing to assign jobs based on a great variety of criteria (Huang *et al.*, 2013). Therefore, a solution for scheduling in a multi-cloud environment has to bridge the gap between local grid or cloud resource managers using meta-schedulers.

Scheduling algorithms are defined based on the topology of the resources that are managed, organized, and administered. Since our work focuses in a multi-cloud environment, decentralized scheduling solutions are in principle suitable for this type of infrastructure. In a centralized scheme, schedulers may have a complete knowledge of the infrastructure, something that is unrealistic in multi-clouds. In a decentralized scheme, this information is incomplete, and the jobs received from the meta-scheduler are assigned to the local scheduler in the same or in a different host. As all the jobs are submitted locally, distributed schemes allow jobs to be transferred to remote hosts to achieve better local resource utilization leading to a global load equilibrium, as required in multi-clouds. So, meta-scheduling schemes have to adapt to the dynamics and unpredictability of the multiple clouds.

In terms of decentralized scheduling methods, Weissman & Grimshaw (1996) proposed a wide-area scheduling system based on a local resource manager and a wide-area scheduler. The goal of this scheduler is to provide high performance for the jobs and exploit resources in remote sites. For the authors, a network is organized as a collection of *sites*, *i.e.*, a collection of administrative domains with certain security policies, file systems, and most importantly, computing resources. Each site manages the local resources, and a wide-area scheduler shares the information between remote scheduling managers in order to perform global scheduling.

In 2010, Wang *et al.* suggested bidding as an alternative means of resource selection in grid computing (Wang *et al.*, 2010). However, under the bidding model, there is no global information in the dynamic environment that forms the grid to make a decision. For this reason, the authors proposed a resource selection heuristic approach to minimize the turnaround time in a non-reserved bidding based grid system. By conducting a series of experiments, they claim that dissolve-probabilistic heuristics perform better than other selected heuristics. However, this work does not consider scheduling variables such as job workload, CPU and memory capability, job execution deadlines, network features and dynamic availability of resources.

Scheduling tasks in distributed environments using deadlines to provide quality of service, at scales of hundred thousand nodes, is a challenging problem. Celaya & Arronategui (2011) presents a fully decentralized scheduler that allocates tasks with high responsiveness, which is unique for systems with such scale. This scheduler operates on bag-of-tasks applications with time, memory and disk restrictions. By using a best-effort approach, the scheduler executes tasks before the deadline. The scheduling is divided into two layers, local and global. A local scheduler contains a task queue and reports its status to the global scheduler. In turn, the objective of the global scheduler is to aggregate the information of the local schedulers, and allocate tasks to nodes that can execute the work before the deadline.

The authors have shown using simulations that the scheduler can operate on applications with many tasks, and with a network up to a hundred thousand of nodes. The scheduler shows competitive performance very close to a centralized scheduler, with little overhead and small allocation times.

Another recent work introduces a decentralized dynamic scheduling approach called community-aware scheduling algorithm (CASA) (Huang *et al.*, 2013). This paper presents a two-phase scheduling solution comprised of a set of heuristics to facilitate job scheduling across decentralized distributed nodes. In the first phase, job submission phase, the goal is to find the proper node from the scope of the overall grid. However in the second phase, dynamic scheduling phase, the motivation is to keep improving the previous scheduling decision by allowing queued jobs to be kept rescheduled in accordance with changes in the grid. Both phases work together to ensure a rapid job distribution and an optimized rescheduling process. CASA's great difference in comparison with other approaches is that it aims for an overall performance improvement, rather than boosting the performance of individual hosts. Applying CASA in a decentralized scheduling setting could lead to similar performance compared with a centralized solution. Also, the job slowdown and waiting times are dramatically improved because it is not necessary to ask for detailed information of participating nodes. Nevertheless, the authors suggest that further enhancements should be considered to include local scheduling algorithms, like backfilling methods and shortest job first.

Different cloud providers address different needs, yet they share the same characteristics in term of resource provisioning. Therefore, multi-clouds environments should allow tasks to be exchanged by exploiting resources from different providers. Recently, the authors of (Sotiriadis *et al.*, 2015) considered these points and presented a novel inter-cloud job scheduling framework, named Inter-Cloud Meta-Scheduling (ICMS), that implements policies to optimize the performance of participating clouds. They have focused on the performance optimization of Infrastructure as a Service (IaaS) using the meta-scheduling paradigm to achieve an improved job scheduling across multiple clouds. The framework is based on a novel message exchange mechanism to allow optimization of job scheduling metrics. The resulting system improves performance due to a better request distribution in comparison with current approaches.

All the solutions described in this section have the goal of choosing wisely the resources to launch work and to optimize the performance. We leverage on several of these works for the design of our multi-cloud schedulers. In the Table 2.1, we summarize the schedulers for the reader to grasp the main ideas of each one.

Scheduler	Type	Target	Key idea
FIFO	Centralized	MapReduce	Simple scheduler that assigns resources to jobs as they are launched.
Fair Scheduler	Centralized	MapReduce	Scheduler that assigns an equal share of resources to all jobs over time (Apache, 2013).
LATE	Centralized	MapReduce	Scheduler for allocation of tasks that detects and mitigates stragglers (Zaharia *et al.*, 2008).
MrHeter	Centralized	MapReduce	Scheduler for allocation of tasks in heterogeneous nodes (Zhang *et al.*, 2016).
Resource Allocation Framework	Centralized	MapReduce	Framework that offers resource sizing and provisioning services in MapReduce environments to meet given service levels objectives (SLOs) (Verma *et al.*, 2011).
FlowFlex	Centralized	MapReduce	Scheduling algorithm for flows of MapReduce jobs connected by precedence constrains (Nagarajan *et al.*, 2013).
Wide-area scheduler	Decentralized	Internet	A distributed algorithm for wide-area scheduling that exploits resources in remote sites and provides high performance (Weissman & Grimshaw, 1996).
Dynamic resource selection	Decentralized	Grid	Dynamic resource selection heuristic that uses bidding model for resource selection (Wang *et al.*, 2010).
Highly Scalable Decentralized Scheduler	Decentralized	Cloud, grid, or volunteering computing	Decentralized scheduler that uses information of the nodes to allocate tasks with deadlines (Celaya & Arronategui, 2011).
K-Dual and K-distributed scheme	Decentralized	Grid	Distributed schemes that offer better performance than concurrent centralized and decentralized schedulers (Subramani *et al.*, 2002).
CASA	Decentralized	Grid	Community-aware scheduling algorithm to achieve optimized scheduling performance over different grids (Huang *et al.*, 2013).
ICMS	Decentralized	Cloud	Multi-cloud job scheduling framework that improves performance by exploiting resources from multiple clouds (Sotiriadis *et al.*, 2015).

Table 2.1: Summary of scheduling algorithms

2.5 Summary

This chapter provided background on the problem at hand, mainly by introducing the necessary concepts and discussing relevant work done in the area. We organized the related work in four main areas of interest. First, we introduced the necessary concepts and architecture of Hadoop MapReduce. We stated that the popularity of the framework has lead to an impressive list of works that try to improve specific features to solve particular problems or to adapt it to specific environments. However, none of these works improve the original MapReduce regarding fault tolerance, namely, to tolerate arbitrary faults, malicious faults, and cloud outages.

Second, we acknowledge that it is hard to deliver reliable and resilient computing services inside the cloud infrastructure. Existing cloud computing services are problematic for running critical applications, which may range from business-critical tasks of companies to mission-critical tasks for the society. In fact, the trust put on cloud providers starts to be questioned due to a set of incidents that affect users' services. As such, using multiple clouds to mitigate cloud-scale disasters becomes an interesting possibility to investigate. With this in mind, we have looked at systems that use multiple clouds to run MapReduce applications. Although these solutions enable the capability of performing large scale data analytics across multiple clouds, none of them enhances the dependability of the framework.

As our goal is to increase the fault tolerance of MapReduce, and as the distributed systems literature is rich in this topic, next we focused on the several techniques like consensus and quorums that are used to create Byzantine algorithms. We have investigated Byzantine fault-tolerant solutions for cloud environments to understand if it would make sense to run MapReduce using traditional techniques. We have realized this option to be expensive in terms of MapReduce execution time.

Finally, we acknowledge that increasing dependability has a cost, but, as we care about performance, we have looked at scheduling, a fundamental problem to share wisely resources among different users. Respectively, we have presented literature on MapReduce scheduling and on distributed scheduling, some of which has inspired our own solutions.

3
Dependable MapReduce in a Single Cloud

MapReduce is often used for critical data processing, and there is evidence that there are arbitrary faults that may corrupt the results of MapReduce execution without being detected. In this chapter, we present an algorithm that masks arbitrary faults by executing each task more than once, comparing the outputs of these executions, and disregarding non-matching outputs. This simple but powerful idea allows our Byzantine fault-tolerant (BFT) MapReduce framework to tolerate any number of faulty task executions at the cost of one re-execution per faulty task. With the aim of guaranteeing acceptable performance, we also designed two novel schedulers that allow our framework to run in two modes: *non-speculative* and *speculative*. We thoroughly evaluate experimentally the performance of these two schedulers in a real testbed, showing that they use around twice more resources than Hadoop MapReduce, instead of the three times more of alternative solutions. We believe this cost to be acceptable for many critical applications.

3.1 Introduction

The fault tolerance mechanisms of the original MapReduce and Hadoop cannot deal with *arbitrary* or *Byzantine faults* (Avizienis *et al.*, 2004), even if considering only accidental faults, not malicious faults, as we do in this work. These faults cannot be detected using file checksums, so they can silently corrupt the output of any map or reduce task. However, it is possible to mask the effect of such faults by *executing each task more than once*, comparing the outputs of these executions, and disregarding the non-matching outputs. This basic idea was proposed by Sarmenta in the context of volunteer computing to tolerate malicious volunteers that returned false results of tasks they were supposed to execute (Sarmenta, 2002). However, he considered only bag-of-tasks applications, which are simpler than MapReduce jobs. A similar but more generic solution consists in using the *state machine replication* approach (Schneider, 1990). This approach is not directly applicable to the replication of MapReduce tasks, only to replicate the jobs, which is expensive. A cheaper and simpler solution, which we call *result comparison scheme*, would be to execute each job twice and re-execute it if the results do not match, but the cost would be high in case there is a fault (the whole job re-execution).

We start by exploring the notion of replication of computation in order to tolerate arbitrary faults and obtain acceptable performance results. We implemented a BFT MapReduce framework [1] that is able to tolerate arbitrary faults by executing each task more than once and comparing the outputs. Since we are especially interested in the performance of the system, we have developed two scheduling algorithms, *speculative* and *non-speculative*, that have the goal to improve performance of the jobs that run on the platform by managing the beginning of the map and reduce tasks. In non-speculative mode, $f+1$ replicas of all map tasks have to complete successfully for reduce tasks to be launched. In speculative execution, reduce tasks start after one replica of all map tasks finish. While the reduce tasks are running, it is necessary to validate the remaining map replicas' outputs. If at some point it is detected that the input used in the reduce tasks was not correct, the tasks will be restarted with the correct input.

The challenge was to create a platform that could execute the tasks *efficiently*, without the need of running $3f+1$ replicas to tolerate at most f faults, which would be the case with state machine replication (e.g., Castro & Liskov (2002); Clement *et al.* (2009a); Veronese *et al.* (2009)). The system uses several techniques to reduce the overhead. It manages to run only two copies of each task when there are no faults plus one replica of a task per faulty replica, instead of a replica of the whole job as in the result comparison scheme.

We thoroughly evaluate experimentally the performance of these two schedulers in a real testbed, showing that they use around twice more resources than Hadoop MapReduce,

[1] This framework is a modified version of Hadoop 1.X.

instead of the three times more of alternative solutions. We believe this cost is acceptable for many critical applications where it is pivotal to obtain valid results .

In summary, the main contributions of this work are:

- an algorithm to execute MapReduce jobs that tolerates arbitrary faults and than can run in two modes, speculative and non-speculative;
- an extensive experimental evaluation of the system using Hadoop's GridMix benchmark in the Grid'5000 testbed.

The successive sections are organized as follows. In Section 3.2 we describe the system model. Then, in Section 3.3, we present the two new type of executions, *non-speculative* and *speculative* executions to launch the minimum number of tasks without the need to run $3f + 1$ replicas to tolerate at most f faults. We have performed a thorough assessment of the new solution in order to understand the cost in comparison to the use of common fault tolerance techniques and we show the results in Section 3.4.

3.2 System model

The system is composed by a set of distributed *processes*: the *clients* that request the execution of jobs composed by map and reduce functions, the *job tracker* that manages the execution of a job, and a set of *task trackers* that launch map and reduce tasks. We do not consider the components of HDFS in the model, as the algorithm is to mostly orthogonal to it (and there is a Byzantine fault-tolerant HDFS in the literature (Clement *et al.*, 2009a)).

We say that a process is *correct* if it follows the algorithm, otherwise we say it is *faulty*. We also use these two words to denominate a task (map or reduce) that, respectively, returns the result that corresponds to an execution in a correct task tracker (correct) or not (faulty). We assume that clients are always correct, because they are not part of the MapReduce execution and if clients were faulty the job output would be necessarily incorrect. We also assume that the job tracker is always correct, which is the same assumption that Hadoop does (White, 2009). It would be possible to remove this assumption by replicating the job tracker, but it would complicate the design considerably and this component does much less work than the task trackers, so we leave this as future work. The task trackers can be correct or faulty, so they can arbitrarily deviate from the algorithm and return corrupted results of the tasks they execute.

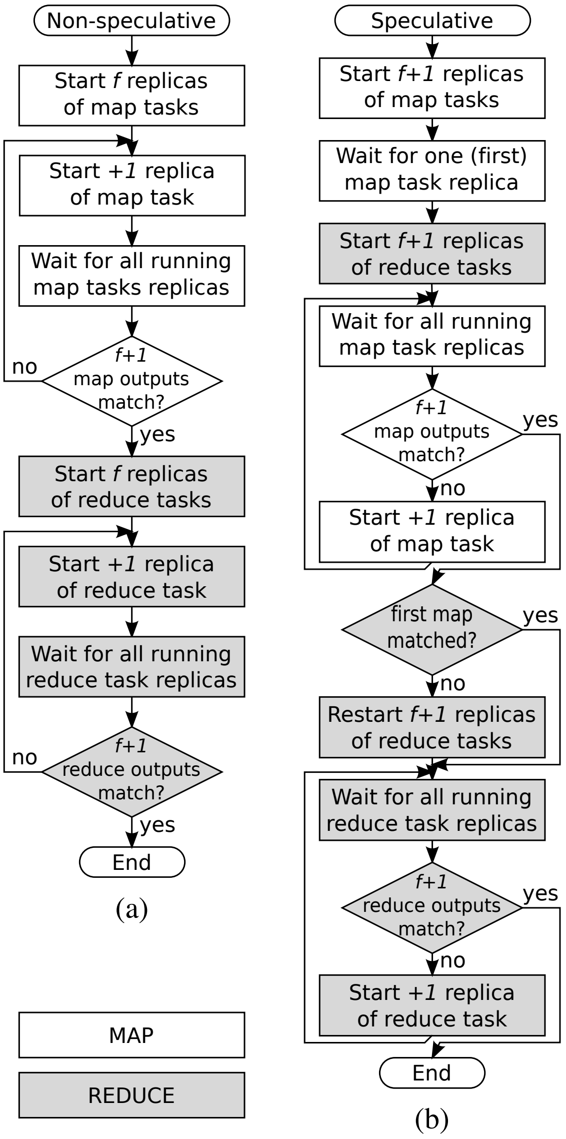
Our algorithm does not rely on assumptions about bounds on processing and communication delays. On the contrary, the original Hadoop mechanisms do make assumptions about such times for termination (e.g., they assume that heartbeat messages from correct task trackers do not take indefinitely to be received). We assume that the processes are connected by reliable channels, so no messages are lost, duplicated or corrupted. In practice this is provided by TCP connections. We assume the existence of a hash function to produce message digests. This function is collision-resistant, i.e., it is infeasible to find two inputs that produce the same output (e.g., SHA-3).

Our algorithm is configured with a parameter f. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas (Bessani *et al.*, 2011; Castro & Liskov, 2002; Clement *et al.*, 2009a; Malkhi & Reiter, 1997; Veronese *et al.*, 2009), but in our case the meaning of f is different: *f is the maximum number of faulty replicas that can return the same output given the same input*. Consider a function $\mathscr{F}$, map or reduce, and that the algorithm executes several replicas of the function with the same input I, so all correct replicas return the same output O. Consider also the worst case in which there are f faulty replicas that execute $\mathscr{F}$ and $\mathscr{F}_1(I) = \mathscr{F}_2(I) = ... = \mathscr{F}_f(I) = O' \neq O$. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is O. If the system selects the correct output by picking the output returned by $f+1$ task replicas, it will never select O' because it is returned by at most f replicas. Similarly to the usual parameter f, our f has a probabilistic meaning (hard to quantify precisely): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible.

3.3 Byzantine fault-tolerant MapReduce algorithm

3.3.1 Overview

A simplistic solution to make MapReduce Byzantine fault-tolerant considering f the maximum number of faulty replicas is the following. First, the job tracker starts $2f+1$ replicas of each map task in different nodes and task trackers. Second, the job tracker starts also


Non-speculative
Start *f* replicas of map tasks
Start +*1* replica of map task
Wait for all running map tasks replicas
f+1 map outputs match?
no
yes
Start *f* replicas of reduce tasks
Start +*1* replica of reduce task
Wait for all running reduce task replicas
f+1 reduce outputs match?
no
yes
End
Speculative
Start *f+1* replicas of map tasks
Wait for one (first) map task replica
Start *f+1* replicas of reduce tasks
Wait for all running map task replicas
f+1 map outputs match?
yes
no
Start +*1* replica of map task
first map matched?
yes
no
Restart *f+1* replicas of reduce tasks
Wait for all running reduce task replicas
f+1 reduce outputs match?
yes
no
Start +*1* replica of reduce task
End
MAP
REDUCE

(a)

(b)

Figure 3.1: Flowcharts of (a) *non-speculative* and (b) *speculative* executions.

$2f+1$ replicas of each reduce task. Each reduce task fetches the output from all map replicas, picks the most voted results, processes them and stores the output in HDFS. In the end, either the client or a special task must vote the outputs to pick the correct result. An even simpler solution would be to run a consensus or Byzantine agreement between each set of map task replicas and reduce task replicas. This would involve even more replicas (typically $3f+1$ (Correia *et al.*, 2006)) and more messages exchanged.

The first simplistic solution is very expensive because it replicates everything $2f + 1$ times: task execution, map task inputs reading, communication of map task outputs, and storage of reduce task outputs. Starting from this solution, we propose a set of techniques to avoid these costs:

Deferred execution. Crash faults, which happen more often, are detected using Hadoop standard heartbeats, while arbitrary faults are dealt using replication and voting. Given the expected low probability of arbitrary faults (Nightingale *et al.*, 2011; Schroeder *et al.*, 2009), there is no point in always executing $2f + 1$ replicas to obtain the same result almost every time. Therefore, our job tracker starts only $f + 1$ replicas of map and reduce tasks. After map tasks finish, the reduce tasks check if all $f + 1$ replicas of every map tasks produced the same output. If some outputs do not match, more replicas are started until there are $f + 1$ matching replies. At the end of execution, the reduce output is also checked to see if it is necessary to launch more reduce replicas. This algorithm is represented as a flowchart in Figure 3.1(a).

Digest outputs. $f+1$ map outputs and $f+1$ reduce outputs must be matched to be considered correct. These outputs tend to be large, so it is useful to fetch only one output from some task replica and compare its digest with those of the remaining replicas. With this solution, we avoid transferring the same data several times causing additional network traffic, and we just transfer data from one replica and the digests from the rest.

Tight storage replication. We write the output of all reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). We are already replicating the tasks, and their outputs will be written on different locations, so we do not need to replicate these outputs even more. A job starts reading replicated data from HDFS, but from this point forward, the data that is saved in the HDFS by each (replicated) task is no longer replicated.

Speculative execution. Waiting for $f + 1$ matching map results before starting a reduce task can worsen the time for the job completion. A way to deal with the problem is for the job tracker to start executing the reduce tasks immediately after receiving the first copy of every map output (see Figure 3.1(b)). Whenever $f + 1$ replicas of a map task finish, if the results do not match, another replica is executed. If $f + 1$ replicas of a map finish with matching results but these results do not match the result of the first copy of the task, then the reduces are stopped and launched again with the correct inputs. For a job to complete, $f + 1$ matching map and reduce results must be found for all tasks, and the reduces must have been executed with matching map outputs.

The difference between the non-speculative and speculative modes of operation (Figure 3.1) is that the latter uses these four techniques, whereas the former excludes speculative execution.

3.3.2 The algorithm in detail

The algorithm is based on the operation of Hadoop MapReduce and follows its terminology. Recall that a client submits a job to the job tracker that distributes the work to the several task trackers. Therefore, the algorithm is divided in the part executed by the job tracker (Algorithms 3.1 and 3.2) and the part executed by the task tracker (Algorithm 3.3). The work itself is performed by the map and reduce tasks, whose code is part of the job specification.

The presentation of the algorithm follows a number of conventions. Function names are capitalized and variable names are in lowercase letters (separated by '_' if composed of multiple words). Comments are inside {...}. The operator |...| returns the number of elements in a set. There are a number of configuration constants and variables. The algorithm is a set of *event handlers* executed in different conditions.

The idea of the algorithm consists essentially in the job tracker inserting tasks in two *queues* – *q_maps*, *q_reduces* – and the task trackers executing these tasks. Several auxiliary functions are used to manipulate these queues:

- *Enqueue(queue, tuple)* – inserts the *tuple* describing a task in *queue*;
- *Dequeue(queue, tuple)* – removes a task described by *tuple* from *queue*;
- *FinishedReplicas(queue, task_id)* – searches in *queue* for replicas of a task identified by *task_id* and returns the number of these replicas that have finished;
- *MatchingReplicas(queue, task_id)* – searches in *queue* for finished replicas of a task identified by *task_id* and returns the maximum number of these replicas that have matching outputs;
- *MaxReplicaId(queue, task_id)* – searches in *queue* for replicas of a task identified by *task_id* and returns the highest *replica_id* among them;

- *ReplicaOutput(queue, task_id, replica_id)* – searches in *queue* for a finished task replica identified by *task_id* and *replica_id* and returns its output;
- *MatchingReplicasOutput(queue, task_id)* – searches in *queue* for finished replicas of a task identified by *task_id* and returns the output of the maximum number of replicas that have matching outputs;

Listing 3.1: Job tracker — common part and non-speculative mode.

```
constants:
    mode            {non-speculative or speculative}
    f               {maximum number of faulty replicas that return the same output}
    nr_reduces      {number of reduce tasks used}
    splits          {split locations}

variables:
    q_maps          {queue of map tasks pending to be executed or being executed; initially empty}
    q_reduces       {queue of reduce tasks pending to be executed or being executed; initially empty}
    reduce_inputs   {identifiers of map replicas that gave inputs to the reduces; initially empty}
    reduces_started {indicates if reduces already started; initialized with false (speculative mode)}

{start execution}
upon job execution being requested do
    for replica_id := 1 to f+1 (
      for map_id := 1 to |splits|
        Enqueue(q_maps, (map_id, replica_id, splits[i], not_running));
    )

{restart a stopped task}
upon task (queue, task_id, replica_id) stopping do
    task := Dequeue(queue, (task_id, replica_id));
    task.running := not_running;
    Enqueue(queue, task);

```

```
{non-speculative mode: start extra map task replica, start reduce tasks}
upon map task (map_id, replica_id, splits) finishing and mode = non-speculative do
  if (FinishedReplicas(q_maps, map_id)$\geq f+1$ and MatchingReplicas(q_maps, map_id)$< f+1$) (
    new_replica_id := maximum replica_id+1;
    Enqueue(q_maps, (map_id, new_replica_id, splits, not_running));
  ) else (
    if ($\forall$map_id : MatchingReplicas(q_maps, map_id)$\geq f+1$) (
      reduce_inputs := {(map_id, urls) tuples with the f+1 matching output urls for each map_id}
      for replica_id := 1 to $f+1$ (
        for reduce_id := 1 to nr_reduces
          Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
      )
      for all map_id, replica_id
        Dequeue(q_maps, (map_id, replica_id));
    )
  )

{non-speculative mode: start extra reduce task replica, finish job}
upon reduce task (reduce_id, replica_id, reduce_inputs) finishing and mode = non-speculative do
  if (FinishedReplicas(q_reduces, reduce_id)$\geq f+1$ and MatchingReplicas(q_reduces, reduce_id)$< f+1$) (
    new_replica_id := maximum replica_id+1;
    Enqueue(q_reduces, (reduce_id, new_replica_id, reduce_inputs, not_running));
  ) else (
    if ($\forall$reduce_id : MatchingReplicas(q_reduces, reduce_id)$\geq f+1$) (
      for all reduce_id, replica_id
        Dequeue(q_reduces, (reduce_id, replica_id));
    )
  )

{send task to task tracker}
upon receiving a TASK_REQUEST (task_type, splits_stored_node) message from task tracker do
```

```
57    queue = undefined;
58    if (task_type = map and ∃(map_id, replica_id, split, not_running) ∈ q_maps) (
59      queue := q_maps;
60      funct := map;
61      replica_id_ := replica_id;
62      inputs := split;
63      if (∃(map_id, replica_id, split, not_running) ∈ q_maps : split ∈ splits_stored_node) (
64        task_id := map_id;
65      ) else (
66        task_id := map_id : (map_id, replica_id, split, not_running) ∈ q_maps;
67      )
68    ) else if (task_type = reduce and ∃(reduce_id, replica_id, split, not_running) ∈ q_reduces) (
69      queue := q_reduces;
70      funct := reduce;
71      task_id := reduce_id;
72      replica_id_ := replica_id;
73      inputs := reduce_inputs;
74    )
75    if (queue = q_maps or queue = q_reduces) (
76      task := Dequeue(queue, (task_id, replica_id_));
77      task.running := running;
78      Enqueue(queue, task);
79      Send EXECUTE_TASK (funct, inputs, task_id, replica_id_) message to the task tracker;
80    ) else (
81      Send NO_TASK_AVAILABLE message to the task tracker;
82    )
```

Let us first present the algorithm executed by the *job tracker* in non-speculative mode (Algorithm 3.1), then the changes to this algorithm when executed in speculative mode (Algorithm 3.2), and finally the algorithm executed by every task tracker (Algorithm 3.3).

Non-speculative job tracker. When the execution of a job is requested, the job tracker inserts $f+1$ replicas of every map task in the *q_maps* queue, which is the minimum number of replicas executed of every task (lines 13-18). Each map task is in charge of processing one input split (line 16). If any task (map or reduce) stops or stalls, it is dequeued and enqueued again to be re-executed (lines 21-24).

In the non-speculative mode, two things may happen when a map task finishes (lines 26-41). If $f+1$ or more replicas of a task have finished but there are no $f+1$ matching outputs, then a Byzantine fault happened and another replica is enqueued for execution (lines 28-30). If there are already $f+1$ matching outputs for every map task, then the map phase ends and the reduces are enqueued to be executed (lines 32-40). To be consistent with Hadoop's nomenclature, we use *urls* to indicate the locations of the map outputs passed to the reduces.

When a reduce task finishes in non-speculative mode, two things can happen (lines 43-53). Similarly to the map tasks, if $f+1$ or more replicas of a task have finished but there are no $f+1$ matching outputs, then there was a Byzantine fault and another replica is enqueued (lines 45-47). Otherwise, if there are already $f+1$ matching outputs for every reduce task, then the job execution finishes (lines 49-52).

The last event handler processes a request for a task coming from a task tracker (lines 55-82). If a map task is being requested, the job tracker gives priority to map tasks for which the input split exists in the node that requested the task. Otherwise, it assigns to the task tracker the next non-running map task in the queue. If a reduce is requested, the job tracker returns the next reduce task in the queue.

Listing 3.2: Job tracker — Speculative mode.

```
{speculative mode: start extra map task replica, start/restart reduce tasks}
upon map task (map_id, replica_id, splits) finishing and mode = speculative do
    if (FinishedReplicas(q_maps, map_id)$\geq f+1$) (
      if (MatchingReplicas(q_maps, map_id)$< f+1$) (
        new_replica_id := MaxReplicaId(q_maps, map_id)+1;
        Enqueue(q_maps, (map_id, new_replica_id, splits, not_running));
      ) else (
        if ($\exists$(map_id, replica_id_, url_) $\in$ reduce_inputs :
            ReplicaOutput(q_maps, map_id, replica_id_)$\neq$ MatchingReplicasOutput(q_maps, map_id)) (
          for all reduce_id, replica_id
            Dequeue(q_reduces, (reduce_id, replica_id));
          reduce_inputs := {(map_id, replica_id, url) tuples with the output url of replica_id of a map_id matching $f$ outputs from other different replicas}
          for replica_id := 1 to $f+1$ (
            for reduce_id := 1 to nr_reduces
              Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
          )
        )
        for all map_id, replica_id
          Dequeue(q_maps, (map_id, replica_id));
      )
    ) else (
      if (not reduces_started and $\forall$map_id : FinishedReplicas(q_maps, map_id)$\geq 1$) (
        reduces_started := true;
        reduce_inputs := {(map_id, replica_id, url) tuples with the output url of replica_id of map_id}
        for replica_id := 1 to $f+1$ (
          for reduce_id := 1 to nr_reduces
            Enqueue(q_reduces, (reduce_id, replica_id, reduce_inputs, not_running));
        )
      )
    )

{speculative mode: start extra reduce task replica}
upon reduce task (reduce_id, replica_id, reduce_inputs) finishing and mode = speculative do
    if (FinishedReplicas(q_reduces, reduce_id)$\geq f+1$ and MatchingReplicas(q_reduces, reduce_id)$< f+1$) (
      new_replica_id := MaxReplicaId(q_reduces, reduce_id)+1;
      Enqueue(q_reduces, (reduce_id, new_replica_id, reduce_inputs, not_running));
    )

{speculative mode: finish job}
upon $\forall$map_id : MatchingReplicas(q_maps, map_id)$\geq f+1$ and
```

```
41     ∀reduce_id : MatchingReplicas(q_reduces, reduce_id)≥f+1 and
42     ∀(map_id, replica_id, url) ∈ reduce_inputs :
43     ReplicaOutput(q_maps, map_id, replica_id) = ↩
    MatchingReplicasOutput(q_maps, map_id) and
44     mode = speculative do
45    for all reduce_id, replica_id
46      Dequeue(q_reduces, (reduce_id, replica_id));
```

Speculative job tracker. Algorithm 3.2 contains the functions that change in the job tracker when the algorithm is executed in speculative mode. Similarly to what happens in non-speculative mode, if $f+1$ or more replicas of a map task have finished but there are no $f+1$ matching outputs, another replica is enqueued (lines 4-6). On the contrary to the other mode, only one replica of each map task must have finished for the reduces to be enqueued for execution (lines 23-30). Finally, in speculative mode there is an extra case: if there are $f+1$ matching outputs of a task but they differ from the one that was used to start executing the reduces, all the reduces have to be aborted and restarted (lines 8-21).

When a reduce task finishes in speculative mode, if there are $f+1$ outputs for that task but not $f+1$ with matching outputs, a new replica is enqueued (lines 33-38).

The event handler in lines 40-47 checks if the job can finish in speculative mode. It tests if there are enough matching map and reduce outputs (lines 41-42) and if the reduces were executed with correct input (line 43-44). If that is the case, the job finishes. This handler is exceptional in the sense that it is activated by the termination of both map and reduce tasks; its code might be part of the two previous handlers, but we made it separate for clarity.

Listing 3.3: Task tracker.

```
constants:
    task_type           {type of task this task tracker executes, map ↩
   or reduce}

variables:
    splits_stored_node  {splits currently stored in this node}
    executing_task      {indicates if a task is being executed; ↩
   initialized with false (not executing)}

```

```
 8  {every T units of time request a task for execution or send heartbeat}
 9  upon timer expiring do
10      if (executing_task = false) (
11        Send TASK_REQUEST (task_type, splits_stored_node) message to the↩
        job tracker;
12      ) else (
13        Send HEARTBEAT message to the job tracker;
14      )
15      launch timer;
16
17  {execute a task}
18  upon receiving EXECUTE_TASK (funct, inputs, task_id, replica_id) ↩
       message from the job tracker do
19      if (executing_task = false) (
20        executing_task := true;
21        execute funct(inputs); {funct may be a map or a reduce}
22        Send TASK_FINISHED (task_id, replica_id) message to the job ↩
       tracker;
23        executing_task := false;
24      ) else (
25        Send ANOTHER_TASK_EXECUTING message to the job tracker;
26      )
```

Task tracker. The presentation of the *task tracker* algorithm (Algorithm 3.3) was simplified by considering that a task tracker does not execute tasks in parallel and that it only executes maps or reduces (defined by the constant in line 2). In practice, what happens are essentially N parallel executions of the algorithm, some for map tasks, others for reduce tasks (the number of each is configurable, see Section 3.4). Periodically every task tracker either requests a task to the job tracker when it is not executing one, or sends a heartbeat message reporting the status of the execution (lines 9-15). If it receives a task from the job tracker, it executes the task and signals termination to the job tracker (lines 18-26). The algorithm does not show the details about inputs/outputs but the idea is: the input (split) for a map task is read from HDFS; the inputs for a reduce task are obtained from the nodes that executed the map tasks; the outputs of a reduce task are stored in HDFS.

Discussion. Algorithms 3.1- 3.3 show the implementation of only two of the four mechanisms used to improve the efficiency of the basic algorithm: deferred execution and speculative execution. The digest outputs mechanism is hidden in functions *MatchingReplicas* and *MatchingReplicasOutput*. The tight storage replication is implemented by modifying HDFS.

In the normal case, Byzantine faults do not occur, so the mechanisms used in the algorithm greatly reduce the overhead introduced by the basic scheme. Specifically, without Byzantine faults, only $f+1$ replicas of each task are executed and the storage overhead is minimal. Notice also that our algorithm tolerates any number of arbitrary faults during the execution of a job, as long as there are no more than f faulty replicas of a task that return the same (incorrect) output.

3.3.3 The prototype

The prototype of the BFT MapReduce runtime was implemented by modifying the original Hadoop 1.x source code. Hadoop is written in Java so we describe the modifications made in key classes. HDFS was almost not modified for two reasons. First, it is used only before the execution of map tasks and after the execution of reduces, therefore it barely interferes with the performance of MapReduce. Second, there is a Byzantine fault-tolerant HDFS in the literature (Clement *et al.*, 2009a) so we did not investigate this issue. The single modification was the setting of the replication factor to 1 to implement the tight storage replication mechanism (Section 3.3).

Most modifications were made in the `JobTracker` class in order to implement Algorithms 3.1- 3.2. The constants of the algorithm (lines 1-5) are read from an XML file. The format of the identifier of tasks (maps and reduces) was modified to include a replica number so that they can be differentiated. A map task takes as input the path to a split of the input file. The job tracker gives each map replica a path to a different replica of the split, stored in a different data node, whenever possible (i.e., as much as there are enough replicas of the split available). It tries to instantiate map tasks in the same server where the data node of the split is, so this usage of different split replicas forces the replicas of a map to be executed in different task trackers (`TaskTracker` class), which improves fault tolerance.

The `JobTracker` class stores data of a running job in an object of the `JobInProgress` class. The `TaskTracker` class sends heartbeat messages to job tracker periodically. We modified this process to include a digest (SHA-1) of the result in the heartbeat that signals the conclusion of a task (map or reduce). The digest is saved in a `JobInProgress` instance, more precisely in an object of the `VotingSystem` class. The `Heartbeat` class, used to represent a heartbeat message, was modified to include the digest and task replica identifier.

3.4 Evaluation

We did an extensive evaluation of our BFT MapReduce. Our purpose was mainly to answer the following questions: Is it possible to run BFT MapReduce without an excessive cost in comparison to the original Hadoop? Is there a considerable benefit in comparison to the use of common fault tolerance techniques such as state machine replication? Is there a considerable benefit in using the speculative mode in scenarios with and without faults? Is it possible to still achieve a high degree of locality in comparison to the original Hadoop?

Our experimental evaluation was based on a benchmark that tests particular parts of the Hadoop framework called GridMix (Apache, 2013). More specifically, we used GridMix2, which is composed by the following jobs: MonsterQuery, WebdataScan, WebdataSort, Combiner, StreamingSort and JavaSort. The experiments were executed in the Grid'5000 environment, a French geographically distributed infrastructure used to study large-scale parallel and distributed systems, during several months.

3.4.1 Analytical evaluation

This section models analytically the performance of the BFT MapReduce, the original MapReduce, and hypothetical BFT MapReduce systems based on state machine replication and the result comparison scheme. This performance is analyzed in terms of a single metric: *the total execution time* or *makespan*. Our objective is twofold: to do a comparison with systems that do not exist, so cannot be evaluated experimentally; to provide an expression that helps understanding the experimental results presented in the following sections.

The execution of a job is composed by serial and parallel phases. The job initialization, the shuffle (sending the map task outputs to the reduce tasks), and the finishing phase belong to the serial phase. We model these times as a single value Ts. The two parallel phases are the execution of map and reduce tasks. The time to execute the map (respectively reduce) tasks depends on maximum the number of map (respectively reduce) tasks that can be executed in parallel.

The total execution time of a job (makespan) without faults is obtained using Equation 3.1 for all considered versions of MapReduce. The versions are differentiated by the value of α, that corresponds to the number of replicas of map and reduce tasks executed (without faults). For the *(i)* original MapReduce $\alpha = 1$ and for the *(ii)* BFT MapReduce $\alpha = f + 1$. If we execute *(iii)* the original MapReduce sequentially — sequential BFT MapReduce — $\alpha = f + 1$. In the *(ii)* hypothetical scheme, the client issues the job to a set of replicas that process the job in parallel and return the results, which are compared by the client. In *(iii)*, the client issues the job sequentially to a single cloud and compares the outputs. Byzantine fault-tolerant state machine replication typically requires $3f + 1$ replicas, but in (Yin *et al.*, 2003) have shown that only $2f + 1$ have to execute the service code, therefore, for this version $\alpha = 2f + 1$. The sequential BFT MapReduce scheme consists in executing the whole job $f + 1$ times sequentially; if all executions return the same result, it is considered the correct result, otherwise the job has to be re-executed one or more times until there are $f + 1$ matching results. Therefore, similar to BFT MapReduce but with a different twist, for this scheme $\alpha = f + 1$.

$$Tj = Ts + \alpha \cdot \left\lceil \frac{Nm}{Pm \cdot N} \right\rceil \cdot Tm + \alpha \cdot \left\lceil \frac{Nr}{Pr \cdot N} \right\rceil \cdot Tr - \Omega \tag{3.1}$$

In relation to the rest of the parameters, Nm and Nr are the number of map and reduce tasks executed. Pm and Pr are the number of tasks that can be executed in parallel per task tracker (in Hadoop by default $Pm = Pr = 2$) and N is the number of nodes (or task trackers). Tj is the time of a job execution, Tm is the average time that it takes to execute a map task, and Tr is the same for a reduce task. These values change from job to job and have to be obtained experimentally. Ω expresses an overlap that may exist in the execution of map and reduce tasks. For the original Hadoop we observed that Ω is essentially 0 because reduce tasks start by default after 95% of map tasks finish. So we use this value for the state machine replication and result comparison versions. For the BFT MapReduce we have two cases. In non-speculative mode, again $\Omega = 0$. In speculative mode, the reduces start processing data after the first replica of every map is executed, so there is an overlap and $\Omega > 0$.

To assist in the comparison among MapReduce systems, we introduce a parameter called the *non-replicated task processing time*, Tn. This parameter measures the time to process map and reduce tasks without replication, i.e., in the original Hadoop. It is obtained from Equation 3.1 by setting $Ts = 0$ (it considers only task processing time), $\alpha = 1$ (no replication), and $\Omega = 0$ (no overlap). The Tn parameter is defined in Equation 3.2.

$$Tn = \left\lceil \frac{Nm}{Pm \cdot N} \right\rceil \cdot Tm + \left\lceil \frac{Nr}{Pr \cdot N} \right\rceil \cdot Tr \tag{3.2}$$

Table 3.1 compares the MapReduce systems when there are no faults: the original Hadoop, sequential BFT MapReduce (*BFT-MR-seq*), our BFT MapReduce in non-speculative (*BFT-MR-ns*) and speculative (*BFT-MR-s*) modes, the hypothetical BFT MapReduce based on state machine replication (*BFT-MR-smr*) and the BFT MapReduce based on the result comparison scheme (*BFT-MR-cmp*). The comparison is made in terms of Tj/Tn and assumes $Ts \ll Tn$, $Nm \gg Pm \cdot N$ and $Nr \gg Pr \cdot N$ (consider the contrary: if there are much more resources than tasks, executing some extra tasks does not impact the total execution time). It is shown in terms of a formula and by instantiating it with the first values of f.

In a scenario without faults, *BFT-MR-smr* needs to launch much more tasks than our solution. In contrast, the non-speculative execution needs to launch the same number of tasks as the comparison scheme solution. This shows the benefit of our BFT MapReduce opposed to the state machine replication version.

System	Tj/Tn	$f = 1$	$f = 2$	$f = 3$
Hadoop	1	1	1	1
BFT-MR-seq	$f + 1$	2	3	4
BFT-MR-ns	$f + 1$	2	3	4
BFT-MR-s	$f + 1 - \frac{\Omega}{Tn}$	$2 - \frac{\Omega}{Tn}$	$3 - \frac{\Omega}{Tn}$	$4 - \frac{\Omega}{Tn}$
BFT-MR-smr	$2f + 1$	3	5	7
BFT-MR-cmp	$f + 1$	2	3	4

Table 3.1: Analytical comparison between MapReduce systems without faults.

Table 3.2 shows the impact of a map or reduce task affected by a Byzantine fault in the makespan of a job. The table shows that the impact of a fault in our BFT MapReduce is quite small, whereas for the result comparison scheme the makespan doubles. This happens because it is necessary to launch a new job in case of any fault. In *BFT-MR-smr* there is no need to relaunch any task because this solution can tolerates up to f faults.

Clearly the answer to one of the questions — if there was a clear benefit in using our scheme instead of the result comparison scheme — is positive. Moreover, our solution is just better than the state machine replication solution in case of any faults.

System	Extra time if there is a faulty map task
Hadoop	cannot tolerate
BFT-MR-seq	Tn
BFT-MR-ns	$Tm/(Pm \cdot N)$ on average
BFT-MR-s	$Tm/(Pm \cdot N)$ on average
BFT-MR-smr	0
BFT-MR-cmp	Tj

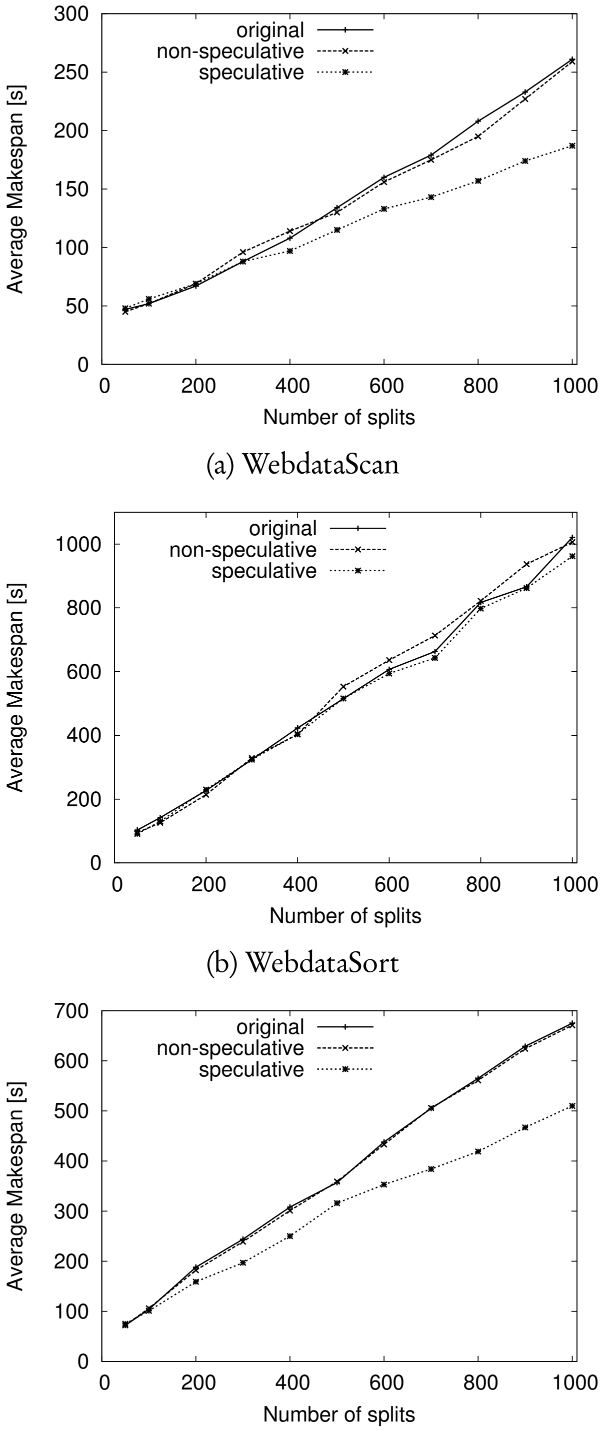
Table 3.2: Effect of a single faulty map task in the makespan for all MapReduce systems.

3.4.2 Experimental evaluation

Makespan vs. number of input splits

Figure 3.3 shows the makespan of the six GridMix2 benchmark applications for the original Hadoop and our BFT MapReduce in the non-speculative and speculative modes. We consider only the case of $f = 1$. We recall that the meaning of f in our system is not the usual one and that the probability of the corresponding assumption being violated is even lower than in other BFT replication algorithms. The values we present are averages of around 100 executions of each experiment. The average does not include outliers, which represent less than 1% of the executions. The standard deviation is low, showing that most of the results are close to the average, and for that reason we do not include this information in any of the following graphs. Each job processed from 50 to 1000 input splits of 64 MB stored in HDFS data nodes. We use the default data-block size of 64 MB to minimize the cost of seeks (White, 2009). To allow results to be comparable, we used a standard configuration for all tests. The times reported were obtained from the logs produced by GridMix. We choose to run the experiments with the original Hadoop in 100 cores and those with the BFT MapReduce in 200 cores, as our framework uses twice as many resources as the first with $f = 1$ and no faults, which is the case we are considering. This allows a fair apples-to-apples comparison.

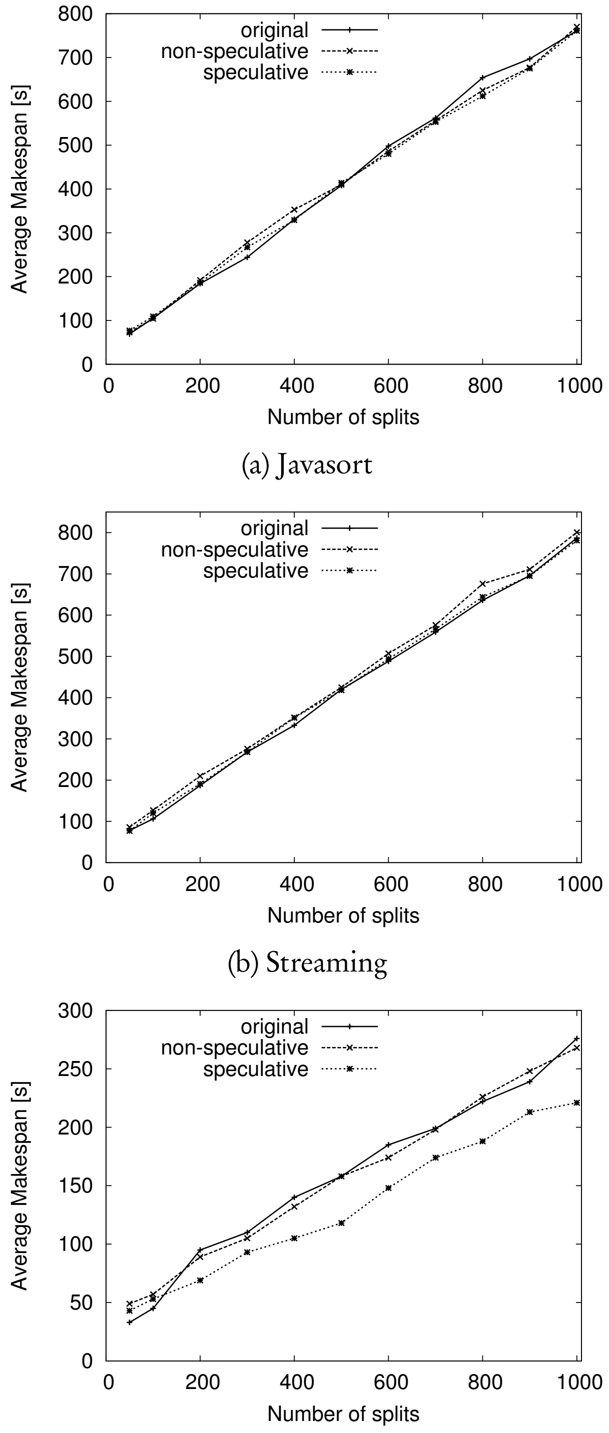
Overall, the original Hadoop and the BFT MapReduce in non-speculative mode had similar makespan in all experiments (see Figure 3.3). This may seem counter-intuitive but



(a) WebdataScan

(b) WebdataSort

(c) Combiner



(a) Javasort

(b) Streaming

(c) Monsterquery

Figure 3.3: Makespan of total execution time of the six GridMix2 benchmark applications varying the number of splits.

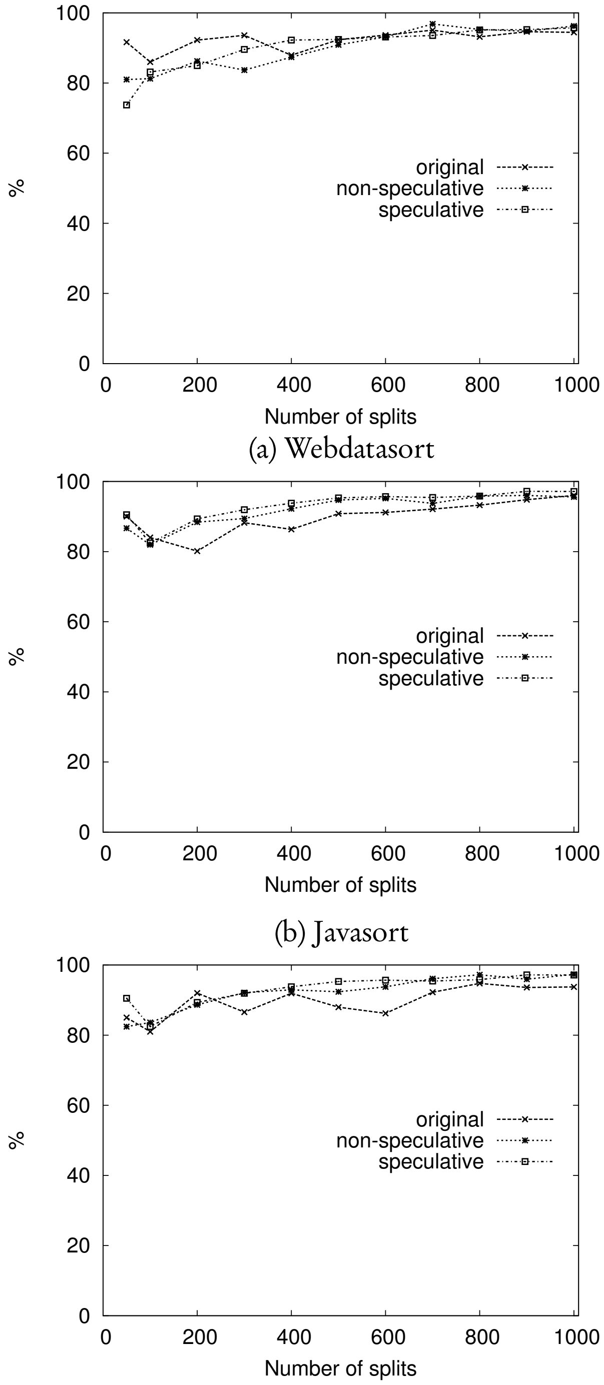
recall that we provided the BFT MapReduce with twice the number of cores of Hadoop. This similarity of makespans shows something interesting: the additional communication contention in the network and in the nodes (caused by the comparisons of all map replica's outputs) did not impact significantly the performance of the BFT MapReduce (twice as much computation was done using twice as many resources in the same time).

The objective of the speculative mode is to improve the makespan by starting reduces earlier, when there are results from at least one replica of each map. The speculative mode improved the makespan in three of the benchmarks — webdatascan, combiner, and monsterquery. We can observe that the improvement gets larger as there are more splits to process, reaching an improvement of 30-40%, which is to be expected as more splits mean more map tasks to process them. Interestingly, the speculative mode had almost no impact in the other three benchmarks with up to 1000 splits. An analysis of the logs of these experiments has shown that the reduce tasks were launched when around three quarters of the maps finished, instead of one half as we would expected with $f+1=2$. This late start of the reduce tasks led to a very small benefit in using the speculative execution. Moreover, it is preferable to execute webdatascan, combiner, and monsterquery with our solution, instead of running sequentially a couple of jobs like we would do in the BFT MapReduce sequential hypothetical system.

In summary, there is a cost associated to running BFT MapReduce in comparison to the original Hadoop, as approximately twice as many resources are used ($\alpha = 2$ in Equation 3.1). Given twice the number of cores, the time to run BFT MapReduce is essentially the same as Hadoop's, somewhat better if the speculative mode is used. With the same number of cores the makespan is approximately the double (Costa *et al.*, 2011).

Locality

The Hadoop MapReduce default scheduler tries to run each map task in the location where its input split is stored. This objective is called locality and a map task for which this is achieved is called a *data-local* task. When this is not possible, the task will be preferably processed in a node in the rack where the input split resides — *rack-local* tasks. If not even that is possible, the task is executed in a node in a different rack. Locality is an important property in MapReduce as moving large amounts of data (splits of 64 MB in our experiments) for a different node takes time and occupies resources, with a negative impact in the makespan and the load in the systems where MapReduce is executed.



(a) Webdatasort

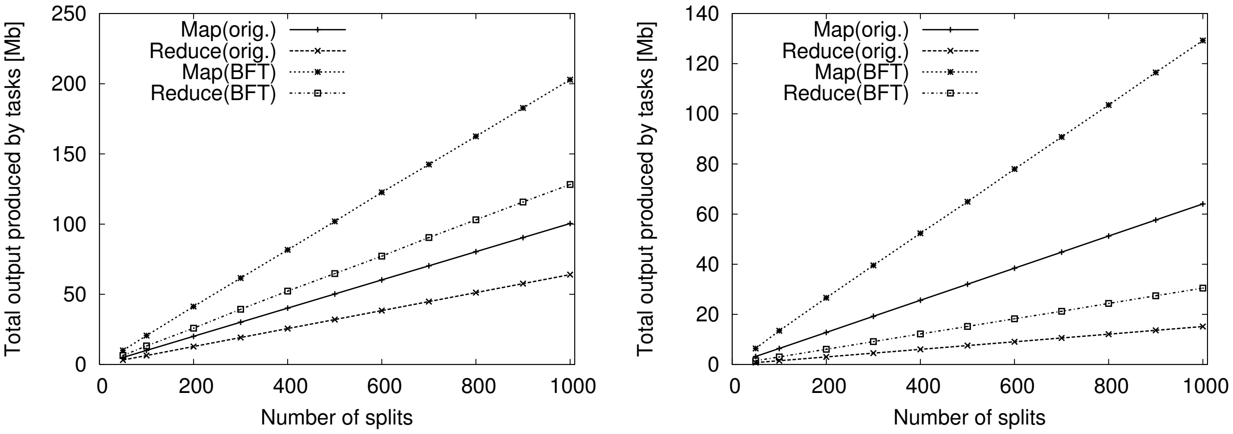
(b) Javasort

(c) Streaming

Figure 3.4: Percentage of data-local tasks in three of the GridMix2 benchmarks varying the number of splits.

Figure 3.4 shows the percentage of data local tasks in the Webdatasort, Javasort, and Streaming benchmarks in the same experiments that were reported in the previous section. The results of the others are similar so we do not show them in the interest of space. The first conclusion is that the locality of the BFT MapReduce in both modes is around 90% and similar to the one achieved in the original Hadoop with half of the nodes. A second conclusion is that although the absolute number of non-data-local tasks increases considerably, the percentage of non-data-local tasks stays reasonably stable when the number of tasks increases.

Data volume



(a) Webdatasort

(b) Streaming

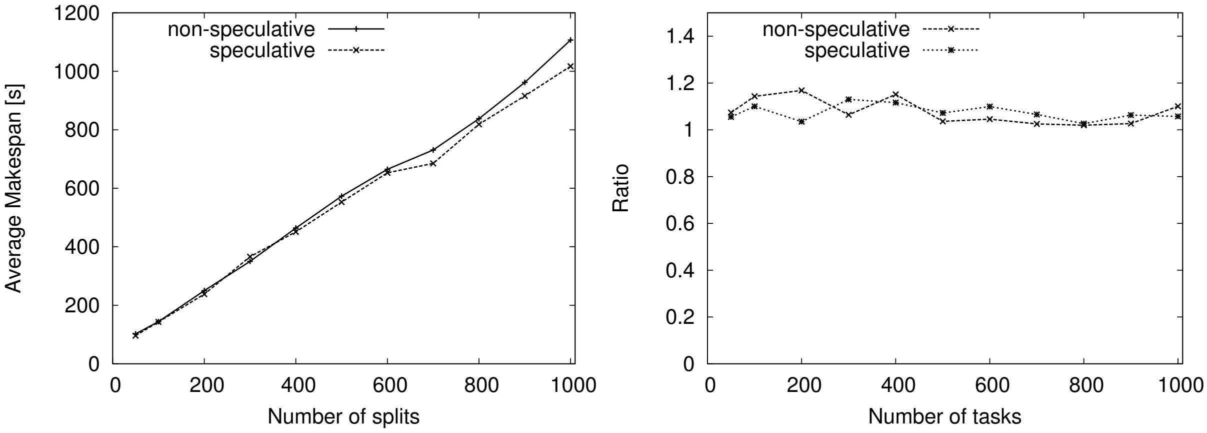
Figure 3.5: Total size of map and reduce outputs in two of the GridMix2 applications with different number of splits.

This section compares the quantity of data processed in the original and in the BFT Hadoop. Figure 3.5 shows the total size of the data produced by the maps and reduces in each experiment, i.e., the sum of the size of the outputs of all maps and the sum of the size of the outputs of all reduces. Recall that each input split has 64 MB. For the webdatasort application, the output of each map task had approximately the same size of its input (64 MB), whereas for streaming the output of a map had an average of 17 MB. The main conclusion is that the total output data of the BFT MapReduce with $f = 1$ and two replicas executed without faults is twice the value for Hadoop, which is the expected result.

Makespan with faults

The experiments of the previous sections were executed in a scenario without faults. We created a simple *fault injector* that tampers outputs and digests of outputs of map and reduce tasks. The component injects random bits leading the job tracker to detect differences in the outputs of replicas, forcing the system to run additional tasks. The percentage of tasks affected by faults is configurable.

We set the percentage of faults to 10%. Figure 3.6(a) shows the makespan of webdatasort with different numbers of splits. The graph shows two lines with a slope similar to those in Figure 3.3. To better compare the makespan with and without faults Figure 3.6(b) shows the ratio between them. We can see that the makespan was roughly 10% longer, which is to be expected after executing 10% more tasks (see Equation 3.1 and consider that Ts and Ω are negligible in comparison to the time taken to execute the tasks).

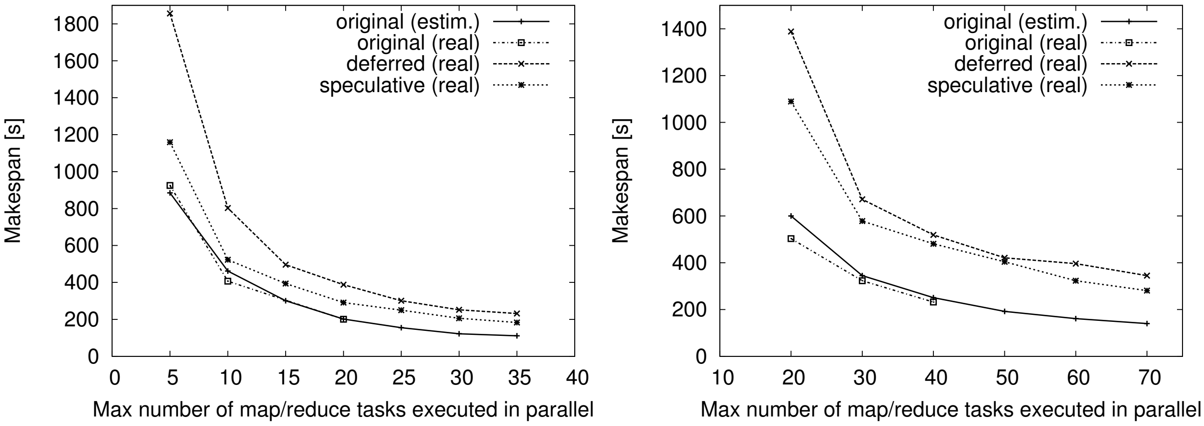


(a) Makespan of both modes of execution. (b) Faulty over faultless makespan ratio.

Figure 3.6: Makespan of the webdatasort benchmark with fault injector enabled.

Makespan vs. parallelism

In the tests presented so far we used a fixed number of nodes. That configuration allowed running the experiments as quickly as possible, within the resource constraints imposed by Grid'5000. This section presents experiments in which we fixed the input data size and varied the number of nodes (without faults). As we increase the number of nodes, we allow more tasks to run in parallel.



(a) Combiner.

(b) WebdataSort.

Figure 3.7: Makespan varying the parallelism without faults.

The experimental results are presented in Figure 3.7. We also plotted values from the original Hadoop estimated using Equation 3.1 (with $\Omega = 0$ and values of Ts, Tm and Tr obtained experimentally). The horizontal axis is the number of map/reduce tasks that can be executed in parallel (i.e., $Pm \cdot N = Pr \cdot N$) and the vertical axis is the makespan. Interestingly, the equation provides a good approximation of the curves (also for the BFT versions, although we do not plot those curves).

The graphs show an exponential drop as the number of map/reduce tasks executed in parallel increases. From the equation, it becomes clear that the curves converge asymptotically to $Ts - \Omega$.

The experimental evaluation confirmed what might be intuited from the algorithm: with $f = 1$ the resources used and the makespan essentially double. Although this is a considerable cost, we have shown that it is much better than alternative solutions: state machine replication and result comparison scheme. This cost may be acceptable for a large number of applications that handle critical data. We also believe that setting f to 1 is a reasonable option as this parameter is the maximum number of faulty replicas that return the same output.

3.5 Summary

Hadoop MapReduce is prepared to tolerate crash faults by re-executing tasks in case of failure. Yet, the framework is not prepared to deal with Byzantine faults that can silently corrupt the output of any map or reduce task. Evidence shows that arbitrary faults are often observed in commodity-hardware datacenters and have disrupted large services. In this chapter, we have presented a Byzantine fault-tolerant MapReduce algorithm that can mask arbitrary faults by executing each task more than once, comparing the outputs of these executions, and disregarding non-matching outputs. This simple but powerful idea allows our solution to tolerate any number of faulty task executions at the cost of one re-execution per faulty task. Always keeping in mind the performance cost, we developed two scheduling algorithms (speculative and non-speculative) that have the goal to improve performance of the jobs that run on the platform.

We performed a thorough experimental evaluation in a real testbed, and the results have confirmed what might be intuited from the algorithm: with $f = 1$ the resources used and the makespan essentially double. Although this is a considerable cost, we have shown that it is much better than alternative solutions: state machine replication and result comparison scheme. In conclusion, we believe that this cost is acceptable for critical applications that require a high degree of certainty of the correctness of the results.

4
Cloud-of-Clouds Dependable MapReduce

Several applications that we use daily have been moving from centralized to decentralized cloud architectures to improve their scalability. MapReduce, a programming framework for processing large amounts of data using thousands of machines in a single cloud, also needs to be scaled out to multiple clouds to adapt to this evolution. The challenge of building a multi-cloud distributed architecture is substantial. Notwithstanding, the ability to deal with the new types of faults introduced by such setting, such as the outage of a whole datacenter or an arbitrary fault caused by a malicious cloud insider, increases the endeavor considerably.

In this chapter we propose Medusa, a platform that allows MapReduce computations to scale out to multiple clouds and tolerate several types of faults. Our solution fulfills four objectives. First, it is transparent to the user, who writes her typical MapReduce application without modification. Secondly, it does not require any modification to the widely used Hadoop framework [1]. Thirdly, the proposed system goes well beyond the fault tolerance offered by MapReduce to tolerate arbitrary faults, cloud outages, and even malicious faults caused by corrupt cloud insiders. Fourth, it achieves this increased level of fault tolerance at reasonable cost. We performed an extensive experimental evaluation in the ExoGENI testbed, demonstrating that our solution significantly reduces execution time when compared to traditional methods that achieve the same level of resilience.

[1] For this chapter, we have used Hadoop MapReduce 2.X version.

4.1 Introduction

Spreading data across multiple servers in different administrative domains can provide availability and fault tolerance to the users. For a growing number of applications, the data is gathered and stored in different datacenters but the analysis required are global. A well-known example includes the analysis of scientific data such as those originated from the Large Hadron Collider (LHC). The tens of petabytes of data produced every year by this particle accelerator are stored in more than 140 datacenters distributed across 34 countries. Moreover, to cope with the unprecedented data growth, applications such as web search, social networking and bioinformatic applications can be distributed in geographically-distant datacenters to leverage data locality and improve data processing efficiency (Baeza-Yates *et al.*, 2009; Matsunaga *et al.*, 2008). In these cases, multiple MapReduce clusters are set up in distributed datacenters. Each cluster collects and processes the data that is close to its datacenter. The final data processing results are obtained by aggregating their respective outputs in an aggregation job.

Acknowledging this trend, the research community has recently proposed MapReduce-based platforms that scale out to multiple clouds, like G-Hadoop (Wang *et al.*, 2013) and G-MR (Jayalath *et al.*, 2014). These examples are demonstrative of the challenges to build multi-cloud distributed architectures. Notwithstanding, the ability to deal with – and to tolerate – the new types of faults that are introduced by this setting makes building such system significantly harder.

Unfortunately, accidental faults that may affect the *correctness of the results* have been known to happen, *corrupting the processing* and leading to wrong values (Meza *et al.*, 2015; Nightingale *et al.*, 2011; Schroeder *et al.*, 2009). This issue is intensified when calculations are scaled out to multiple datacenters, and may indeed become more frequent. Secondly, *malicious attacks* perpetrated by cloud insiders or external hackers can also cause *corruption of the processing and of its results*. For example, a malicious insider in a cloud that hosts an epidemiological surveillance system can tamper the diagnosis of patients with tragic consequences. A recent report mentions malicious insiders as one of the top threats in cloud computing (Cloud Security Alliance, 2013), and alarming instances of this problem have occurred in companies such as Google (Chen, 2010; Kandias *et al.*, 2013). Thirdly, cloud outages may lead to the *unavailability of MapReduce instances and data loss*. Experience shows that these events are also frequent, with cases of unavailability of minutes to days in services like Google Drive or Amazon EC2, to name just a few (CloudSquare, 2015). Several cases have been reported, including the disruption of one Amazon EC2 datacenter for almost five hours in 2015 (Smolaks, 2015), and the disruption of the Google Cloud Engine service for some periods in 2016, affecting customers in all regions (Bort, 2016). Cloud outages can *interrupt the execution of MapReduce jobs*, and the original framework cannot deal with this type of fault as it is restricted to work in a single datacenter. To deal with such faults, one needs to add redundancy to the computation. As both malicious faults and cloud outages can impair a complete cloud, handling them involves resorting to more than one cloud.

The terms *cloud federation* (Kurze *et al.*, 2011) and *cloud-of-clouds* (Bessani *et al.*, 2011) have been used to denote such virtual environments composed of multiple clouds. We explore this idea to replicate MapReduce jobs in different clouds to avoid their incorrectness or unavailability of computation due to arbitrary and malicious faults, and cloud outages.

We propose a novel approach that allows MapReduce to scale out to multiple clouds to tolerate arbitrary and malicious faults, as well as cloud outages, for critical applications. As per above, the use of multiple clouds for MapReduce is not in itself new. The novelty of this work arises from the use of a multi-cloud environment to not only parallelize computation, but also to transparently tolerate different types of faults at the minimum cost. Our solution addresses several non-trivial challenges for this purpose. First, it aims to be a transparent solution for the user. The MapReduce API is not changed, and the user simply writes her typical MapReduce application without modification. Secondly, it does not require any modification to the Hadoop framework. Thirdly, it tolerates not only crash faults, as the original MapReduce, but also arbitrary faults, cloud outages, and malicious faults caused by corrupt cloud insiders. Fourth, it achieves this level of fault tolerance at the minimum replication cost and guaranteeing acceptable performance.

Our approach relies on a *proxy*, Medusa, that runs in the client and that interacts with (unchanged) MapReduce runtimes in different clouds to tolerate the three kinds of faults above. The basic idea is to replicate each MapReduce job in more than one cloud and to compare the outputs of the replicated jobs to tolerate faults. The challenge is to perform this efficiently, and doing so without changing the framework and keeping the whole process transparent to the user. Achieving efficiency requires *(i)* replicating each job the minimum number of times; and *(ii)* assigning each replicated job to the cloud that ensures the best performance. To this end, instead of replicating each job at least $2f+1$ times to ensure a majority of correct results and tolerate f faults, as is common (Schneider, 1990; Veronese *et al.*, 2013), our approach is crafted to run only $f+1$ replicas for each job when there is no fault, and $2f+1$ replicas when there are f faults.

Importantly, our approach can tolerate not only f but any number of faulty replicas or clouds as long as no more than f faulty replicas return the same wrong output. This includes the possibility that up to f clouds maliciously collude and the system remains able to reach a correct output. Moreover, we introduce a novel scheduling algorithm that takes into account the heterogeneity of the individual clouds to schedule the replicated jobs among them in order to reduce data communication and job completion time.

We performed an extensive experimental evaluation of our approach in a real testbed (ExoGENI). The results demonstrate that our solution significantly reduces the execution time when compared to traditional methods that achieve the same level of resilience. As an example, in certain scenarios we achieve a gain of up to 3 in efficiency when compared with a conventional round-robin approach that tolerates cloud faults.

In summary, our proposal is practical and transparent to the user by only involving a new software module running in the client (Medusa), not requiring any modification to the MapReduce framework or to user applications. Simultaneously, it tolerates arbitrary and malicious faults, and cloud outages, efficiently.

The remainder sections are organized as follows. In the next section we describe the system model and define the problem. Then, in Section 4.3 we present the detailed design of Medusa, the cloud fault-tolerant MapReduce system we propose, as well as a new scheduling algorithm that distribute replicated jobs among the clouds to ensure efficient completion of the entire job. Section 4.4 reports the experimental evaluation that we have performed in a testbed to show the advantages of our solution contrasted with customary methods that accomplish a similar level of resilience.

4.2 System model

In a multi-cloud system, the MapReduce job runs in a federation of *clouds*. Each cloud has an HDFS instance to store the initial inputs and final outputs of the jobs running in that cloud. The entire data to be processed by the job is distributed across the clouds in the system, *i.e.*, each cloud has a subset of the data stored in its HDFS. The data can be either

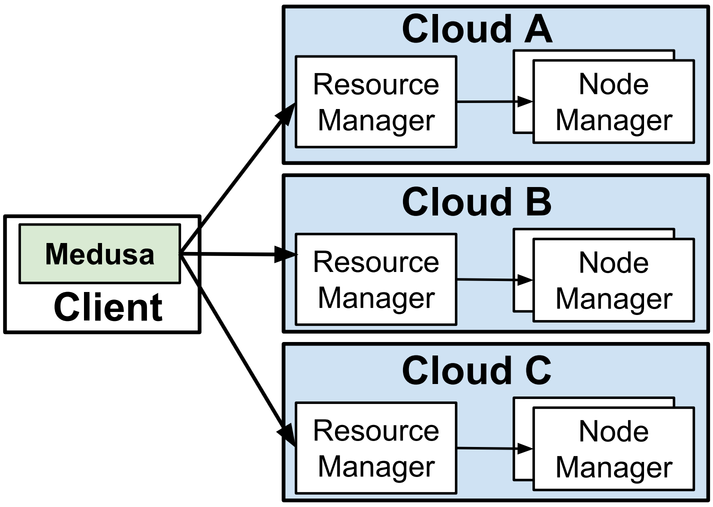


Figure 4.1: Medusa interacting with several MapReduce runtime in a multi-cloud system.

collected by the cloud itself or assigned by some external processes, but we ignore this detail as it is application dependent and is orthogonal to the MapReduce execution.

The system is composed by a set of distributed *processes* (see Figure 4.1). The *client* that requests the execution of a job, submits it through the *proxy* (also called Medusa) to the *resource manager*. The goal of the proxy is to manage the execution of the job in all the clouds. Each cloud first runs its own MapReduce instance to process the data it has: the *resource manager* controls the execution of the part of job assigned to that cloud; and the *node manager* in each server runs the map and reduce tasks assigned to that server. Finally, the proxy is in charge of assigning the outputs from these clouds to an aggregation MapReduce job in one of the clouds to obtain the final output for this job.

The messages between the proxy and the clouds are mediated by a message queuing service (MQ), which uses reliable channels so that no messages are lost, duplicated or corrupted. In practice, this is provided by establishing TCP/IP connections. A message is only lost if the cloud is unreachable.

The system is asynchronous, *i.e.*, we make no assumptions about bounds on processing and communication delays in each stage of job execution (Section 4.3.2). We assume the use of authenticated reliable channels for communication.

We assume the existence of one correct dataset in each cloud at the beginning of job execution. We further assume the existence of a collision-resistant hash function, *i.e.*, a hash function for which it is impossible to let two different inputs produce the same output (*e.g.*, SHA-256). We use the digests produced by this function to verify the integrity of the data replicated between clouds, and to validate the correctness of the job outputs.

The proxy is the key component in the multi-cloud system for tolerating cloud faults. We focus, in this work, on the design of the proxy as well as on a scheduling mechanism that guarantees good system performance.

4.2.1 Fault model

We say that a process is *correct* if it follows the algorithm, otherwise we say it is *faulty*. We also use these two words to denominate a task (map or reduce) that, respectively, returns the result of applying the map/reduce function to the input (correct) or some other result (faulty). We assume that clients are always correct, because they are not part of the MapReduce execution. If clients were faulty, the job output would be necessarily incorrect. We also assume that the proxy is always correct because it runs at the client side, e.g., in the same host as the client or in a host under the same administration. Resource managers and node managers can fail arbitrarily: they can return wrong results (e.g., processing corruption, or malicious insider) or even stop executing (e.g., due to a cloud outage). A cloud is faulty if it becomes partitioned from the rest of the processes, it is compromised by a malicious attacker, or suffers an outage, a disconnection from the Internet or another severe communications problem.

Our algorithm is configured with two parameters f and t. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas, but in our case the meaning of f is different and weaker: f is the maximum number of faulty replicas that can return the same wrong output given the same input. t is the number of faulty clouds that the system tolerates before the service becomes unavailable. Moreover, O is the output of a MapReduce computation. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is O. If the system selects the correct output by picking the output returned by $f+1$ task replicas, it will never select O' because it is returned by at most f replicas. Similarly to the usual parameter f, our f has a probabilistic meaning (hard to quantify): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible.

The other parameter, t, is the maximum number of clouds that may fail arbitrarily (including outages and malicious faults). We assume there are at least $2t + 1$ clouds, to ensure that there are always enough clouds to execute the job.

4.2.2 Problem formulation

We aim at tolerating *(i)* arbitrary and malicious faults, and *(ii)* cloud outages, when running MapReduce jobs in multi-cloud systems. To tolerate f faults, a basic approach is to create $2f + 1$ replicas of each job (*i.e.*, the job running in each cloud and the aggregation job), spread them in $2t + 1$ different clouds, and compare the $2f + 1$ outputs of each job. If at least $f + 1$ outputs are identical, their corresponding MapReduce jobs are correct and the identical output is the correct output of this job.

This basic approach has two major problems. First, it is expensive in terms of computation, communication, and storage. Even if there is no fault, each job is executed $2f + 1$ times. This requires replicating the data initially hosted by each cloud to $t+1$ other clouds, which can be expensive in geographically distributed clouds. The same data also have to be stored in the HDFS of $2t + 1$ clouds. In fact, if there is no fault, executing each job $f + 1$ times is enough (there is no need for additional computations). Second, the basic approach does not take into account the difference among clouds for data replication and data processing. Intuitively, the data initially hosted by a cloud should be replicated to other clouds with which the original cloud has a high pair-wise bandwidth and, simultaneously, has high computational power. This would ensure the efficiency of the entire job execution in the multi-cloud system.

Therefore, our objective in this work is to design a MapReduce proxy that ensures the MapReduce job running in multiple distributed clouds to tolerate cloud faults while *(i)* minimizing the amount of data replication and processing; and *(ii)* ensuring efficient completion of the entire MapReduce job.

4.3 Medusa: a cloud fault-tolerant MapReduce

4.3.1 Overview

As mentioned in Section 4.2, a full job execution in a multi-cloud system is comprised of two phases. The first phase runs a *vanilla MapReduce job* in each cloud that holds a subset of data initially owned by that cloud. The second phase runs a *global MapReduce job* that aggregates the outputs from all clouds to generate the final results. To tolerate arbitrary, malicious faults, and cloud outages, the MapReduce jobs in each phase need to be replicated to other clouds for ensuring the existence of $f+1$ identical outputs and thus the correctness of the results.

We propose in this work a MapReduce proxy that works as a middleware in a multi-cloud system (*i.e.*, a federation of clouds). We refer to this proxy as *Medusa*[1].

As explained, we assume that no more than f replicas of a job return to the proxy identical wrong outputs from their executions. This assumption allows the proxy to know that a result is correct by getting $f+1$ identical results from different clouds / resource managers. Given the expected low probability of arbitrary and malicious faults, it is too expensive to always execute $2f+1$ replicas of a job as is done in typical approaches. Therefore, instead of replicating each job $2f+1$ times, Medusa first replicates each job $f+1$ times in $t+1$ different clouds (*i.e.*, once in the cloud where the data initially are and f times in other clouds that do not have the corresponding data initially). If the executions of these replicas do not produce identical outputs, one more replicated job is launched in a different cloud until the proxy gets $f+1$ identical outputs. This *deferred job execution* avoids the redundant data transmission, storage and processing when no fault happens. As per the point above, each subset of data initially available in one cloud is replicated (at least) $f+1$ times, instead of $2f+1$ times.

We detail in Section 4.3.2 how Medusa works and explain in Section 4.3.3 how the replicated jobs are scheduled among the clouds in the system to ensure efficient completion of the entire job.

4.3.2 Medusa proxy in a nutshell

Each of the two phases of a full job execution is composed of three stages: replicating the data from the local cloud that initially holds them to other clouds; running the replicated jobs in all the clouds having the same data; and agreeing on the outputs of the replicated jobs. In each stage, all processes wait until every other process finishes, to move to the next stage. Fig. 4.2 and Fig. 4.3 depicts the two-phase execution by means of an example, where the full job is running in 3 clouds and is set to tolerate 1 fault ($f=1$).

[1] Medusa is the mythological figure that has living snakes in place of hair. Metaphorically, the connections of our proxy to the clouds are the snakes, and these follow orders given by Medusa's brain (our proxy).

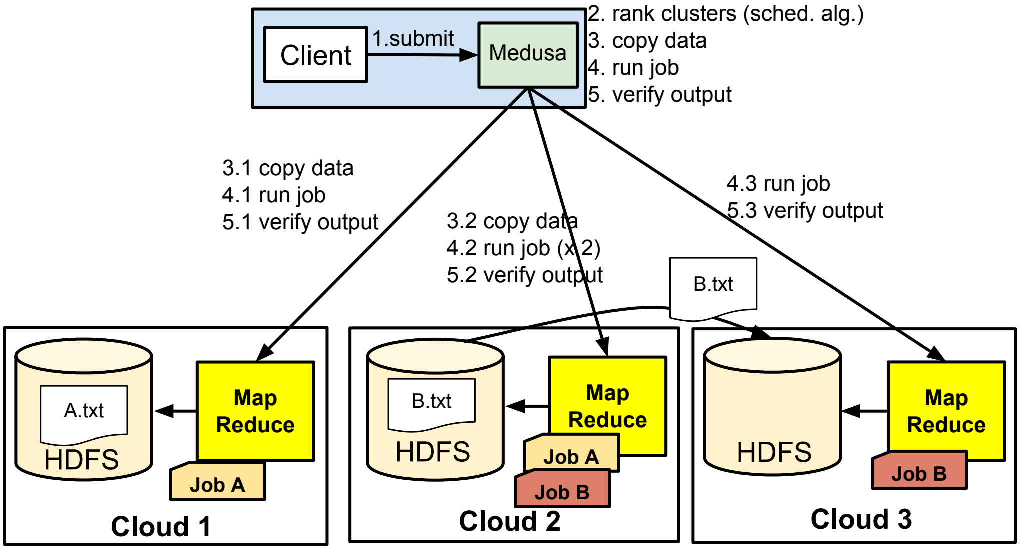


Figure 4.2: Phase 1: Vanilla MapReduce execution.

The proxy interacts with the clouds with the help of a message queuing service (MQ). In Fig. 4.2 and Fig. 4.3, MQ is tightly coupled with the proxy, but in practice it can be running in a different host.

In the first phase (Fig. 4.2), the client submits a job (Step 1) with input data *A.txt* and *B.txt* (representing the two subsets of the entire data to be processed). As mentioned earlier, we assume that the cloud collected or generated the data itself as we anticipate this to be the most common scenario. Hence, the input data is already stored in *cloud 1* and *cloud 2*, respectively. After receiving the request from the client, Medusa runs a scheduling algorithm (Section 4.3.3) to select the best cloud(s) to run the replicated jobs (Step 2). Medusa selects the clouds that offer the best performance in terms of data transmission and data processing to ensure low job makespan (*i.e.*, the time it takes for the whole job to finish). In this example, Medusa chooses to copy *A.txt* to *cloud 2* and *B.txt* to *cloud 3*. Note that to guarantee data integrity, a digest of each data is computed using a collision-resistant hash function (*e.g.*, SHA-256) and is sent with the original data for validation at reception.

If the data transmission is successful and the communication is not tampered, the execution of the job will start (Step 4). After the executions of the $f + 1$ job replicas finish, digests of their outputs are computed using the same collision-resistant hash function, and the correctness of these outputs are verified (Step 5). If all the digests are identical, the

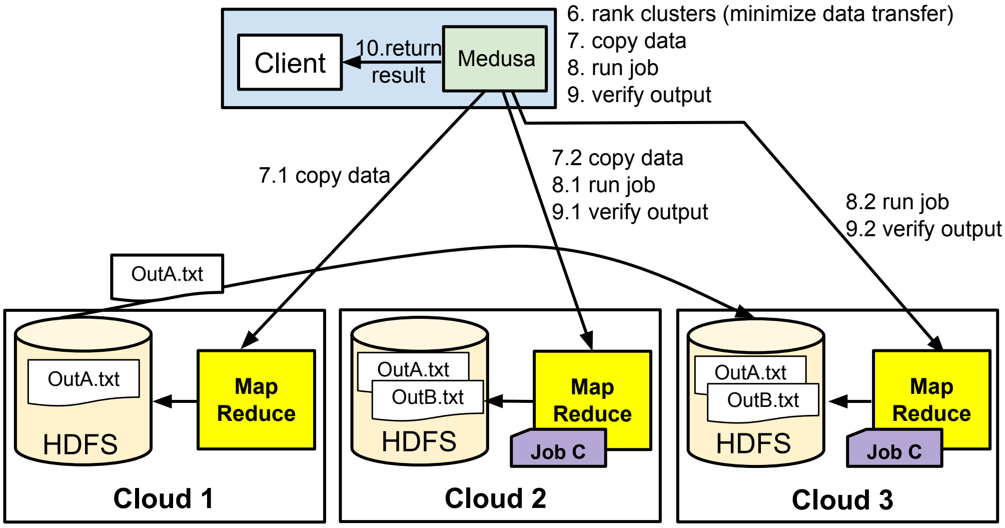


Figure 4.3: Phase 2: Global MapReduce execution.

vanilla MapReduce job finishes successfully. Otherwise, the data is copied to a different cloud, selected by Medusa's scheduling algorithm, and the job is executed again to obtain $f+1$ identical outputs. Once a majority of correct MapReduce jobs have finished, the second phase can start.

In the second phase (Fig. 4.3), Medusa chooses the $t+1$ clouds that minimize the data transmission time to gather the outputs of the first phase and to run the global MapReduce jobs to produce the final output. In this example, only *OutA.txt* is replicated to *cloud 3* (Step 7). Then, after verifying the integrity of the data, the $f+1$ replicated jobs are launched (Step 8). Once the $f+1$ replicated jobs finish, the digests of their outputs are computed and compared (Step 9). As in the first phase, if the digests are identical, the job finishes successfully, and the correct result is returned to the client. Otherwise, another replica of the global MapReduce job is launched until $f+1$ equal results from different clouds are obtained.

4.3.3 The Medusa scheduler

When a client submits a job, Medusa needs to instantiate $f+1$ replicas of the job. These replicas will be launched in different clouds to tolerate cloud faults as explained in the previous section. Whenever there is a disagreement on the output of a job, Medusa needs to launch an extra execution of that job.

Deciding in which cloud a replicated job should be executed is crucial for the performance of the system. Specifically, the client is interested in minimizing the time it takes for the submitted job to complete: the *makespan* of the job. Intuitively, if a job, as well as the data it needs to process, is replicated to a particular cloud with high computational power and to which it is connected by high-bandwidth links, it should take relatively shorter time for the job to complete, when compared to the available alternatives. On the contrary, if the data has to be copied to a cloud using low bandwidth links, or to a cloud that is already overloaded, it may take a very long time for the job to complete. Therefore, to ensure a small makespan of the entire job, it is important for Medusa to choose the appropriate clouds to replicate each job.

To this end, we propose a scheduler to distributes the replicated jobs across different clouds based on the predicted data transmission time and data processing time in each cloud. The prediction takes into account the historical performance as well as the current status of each cloud, allowing us to incorporate the *heterogeneity* of the clouds into the scheduling decision.

The overall makespan is determined by the longest time to complete the vanilla MapReduce jobs running in all the clouds and the time to complete the global MapReduce job. Each job is replicated and executed between $f+1$ times and $2f+1$ times. The Medusa scheduler follows a greedy approach and selects, for each job, the clouds that minimize the time to complete the executions of all its replicas. Specifically, the Medusa scheduler estimates, for each job, the time to replicate and process the corresponding data in each cloud that does not initially holds the data, and selects the $t+1$ clouds in increasing order of the estimated time. In case of a fault, the Medusa scheduler makes another estimation and chooses the cloud with the shortest estimated time to run the extra replica. Note that the scheduler will deal in accordance with the type of fault. When the system is set (by the client) to assume any fault can be malicious, it will not consider the cloud where the fault occurred as an option to run the extra replica. That cloud cannot be trusted again since the malicious insider may corrupt the results once more. When the system is set to only tolerate arbitrary faults and cloud outages, any alive cloud can be an option to launch the extra replica.

Formally, the estimated time $t_1(i)$ for completing a (replicated) vanilla MapReduce job (*i.e.*, phase 1) in cloud i can be written as

$$t_1(i) = t_{trans}(j,i) + t_{1,proc}(i), \tag{4.1}$$

where $t_{trans}(j,i)$ is the estimated time to transfer the data to be processed by the job from cloud j to cloud i, and $t_{1,proc}(i)$ is the estimated time to execute the vanilla MapReduce job in cloud i.

The estimated time $t_2(i)$ for completing a (replicated) global MapReduce job (*i.e.*, phase 2) in cloud i can be written as

$$t_2(i) = max_{j \in C \wedge j \neq i}\{t_{trans}(j,i)\} + t_{2,proc}(i). \tag{4.2}$$

This equation requires further explanation. For the global MapReduce job, all the outputs produced by the vanilla MapReduce jobs need to be copied to cloud i except the ones that already exist in cloud i. As the data transmission can take place in parallel, the time for transmitting the outputs of the vanilla MapReduce jobs is bounded by the maximum time for transmitting the outputs from any cloud j to cloud i ($t_{trans}(j,i)$). $t_{2,proc}(i)$ is the estimated time to execute the global MapReduce job in cloud i.

Therefore, for each job the Medusa scheduler selects the cloud i that has the shortest estimated time $t_1(i)$ or $t_2(i)$ to run a replica among the clouds that have not been selected until the correct output is obtained. We explain in the following how the data transmission time $t_{trans}(j,i)$ and the data processing time $t_{1,proc}(i)$ and $t_{2,proc}(i)$ are estimated, respectively.

Estimating data transmission time The data transmission time between two clouds depends on *(i)* the network distance between them, *(ii)* the network throughput between them, and *(iii)* the size of the data to transfer. Hence, we estimate the time to transfer data of size S between cloud j and cloud i as

$$t_{trans}(j,i) = l(j,i)/2 + S/T(j,i), \tag{4.3}$$

where $l(j,i)$ is the round-trip time between cloud j and i, and $T(j,i)$ is the estimated throughput between them.

The value of $l(j,i)$ depends on the geographical distance between clouds j and i, the speed of transmission in the different propagation media connecting them, among other variables, and is usually stable. Since this value is small (typically in the milliseconds range), accurate estimation of the data transmission time depends largely on the estimation of the network throughput $T(j,i)$ between cloud j and cloud i. Nevertheless, we decide to include both variables in Equation 4.3 for the sake of model completeness.

Considering that the throughput varies depending on the traffic load of other connections, Medusa keeps track of the throughput for each pair of clouds in the system. The measures are taken sequentially and periodically by running the network tool *Iperf*, a tool that measures maximum TCP bandwidth. Specifically, a script is running in each cloud to measure its throughput to other clouds. Before scheduling a job, Medusa will access this information to estimate the throughput from one cloud to the others. The throughput between two clouds is estimated as the average throughput between them over a window of size k, where k is the number of the most recent throughput measurements.

Estimating data processing time The time for completing a given MapReduce job mainly depends on *(i)* the capacity of the cloud running this job and *(ii)* the configuration of the job. Obviously, if a cloud has high computational power and large amounts of free resources, the job running on it will finish within a short time. In addition, a high level of parallelization (*i.e.*, large number of map and reduce tasks) for the same job in the same cloud implies shorter data processing times. Considering this, the Medusa scheduler relies on a linear regression model to predict the data processing time for a MapReduce job to complete in a cloud.

Specifically, the Medusa scheduler trains one linear regression model for each cloud in the system to predict its time to complete a MapReduce job, in the form

$$\hat{y} = \beta_1 x_1 + \cdots + \beta_n x_n + \beta_0, \tag{4.4}$$

where $\hat{y}$ is the data processing time to predict (*i.e.*, $t_{1,proc}(i)$ or $t_{2,proc}(i)$ in cloud i) and x_1, ..., x_n are the n features we use to make the prediction. We estimate the parameters β_0, ..., β_n using the least squares approach (Björck, 1996).

The Medusa scheduler relies on three types of features to make the prediction: *(i)* job configuration; *(ii)* cloud capacity; and *(iii)* cloud overhead. We describe the representative features for each type in the following.

Job configuration features. We consider the size of the input data, the number of map tasks and the number of reduce tasks as features in this type. Clearly, large input data and a small number of map and reduce tasks imply long job completion times. The values of these features are always known to the Medusa scheduler.

Cloud capacity features. We consider the clock speed (MHz) and the number of cores of the CPU and the total memory capacity (MBs) as features in this type. These are variables that define the capacity of cloud, but they do not tell the load of the cloud in a specific time.

Cloud overhead features. In addition to the computational capacity, the overhead in the cloud also has an impact on the completion time of the job to schedule. For instance, if a cloud is overloaded and there are already a number of jobs queued to be launched, the scheduled job will not finish in a short time even if very small. In contrast, if a cloud has more free resources the scheduled job can finish early even if its capacity is relatively low. We use the number of MapReduce jobs that are currently running in the cloud, the percentage of completion of the running MapReduce jobs, the number of MapReduce jobs that are queued to run, and the size of the input data of the running jobs as features in this type. These are part of the filesystem information that MapReduce can provide.

For each vanilla MapReduce job to schedule, the Medusa scheduler estimates the data processing time $t_{1,proc}(i)$ in cloud i using this linear regression model.

4.4 Evaluation

This section evaluates the performance of our system. Section 4.4.1 describes the experimental setup as well as the implementation and configuration of Medusa. Section 4.4.2 reports on its performance, considering both the presence and absence of faults during job execution.

4.4.1 Experimental setup

We evaluate the performance of Medusa using the *WordCount*, *WebdataScan*, and *Monsterquery* benchmarks from Hadoop's Gridmix benchmark (Apache, 2013), as examples of applications commonly used in real-world scenarios.

Running *WordCount* in a multi-cloud system can be considered as building the inverted indexes of a multi-site web search engine for each search site (*i.e.*, cloud). Specifically, in a multi-cloud web search engine each site is in charge of collecting and indexing a subset of the entire document collection. To build the search index (that supports the term frequency-inverse document frequency style ranking functions, TF-IDF), each search site runs a local MapReduce job to parse the documents it has and to count the occurrences of each term in a document (*i.e.*, TF) and the number of documents in that search site containing each term (*i.e.*, partial IDF). This can be achieved by running WordCount as a vanilla MapReduce job in each cloud. To ensure the same search results can be retrieved as in a single-cloud search engine, the local outputs from previous executions need to be aggregated to obtain the number of documents in the search engine that contain each term (*i.e.*, global IDF). To this end, we implemented a *WordCountAggregator*. This corresponds to the global MapReduce job described in Section 4.3. *WebdataScan* is a benchmarking application that extracts samples from a large data set, which is a common form of processing in many systems. *Monsterquery* is another benchmarking application that queries part of the data from a large data set. The MapReduce framework divides a query into steps and the dataset into chunks, and then runs those step/chunk pairs in separate physical hosts. The mappers perform the data collection phase and the reducers take care of data processing.

For evaluating these applications we used up to 6GB of data generated by Gridmix, equally partitioned and stored in all clouds. For a small subset of the experiments we tested larger files. The results obtained using these larger datasets confirm the general trend we report next.

We evaluated the system in the ExoGENI testbed (Baldine *et al.*, 2012), a distributed networked infrastructure-as-a-service spread across the USA that allows setting up virtual topologies across sites and servers in each site.

We set up four clouds located in different sites for each experiment. For the WordCount benchmark, we used clouds located in the East and West coasts of the USA: California, Chicago, and West-Virginia. In the WebdataScan and MonsterQuery benchmarks the clouds were geographically closer: Pittsburgh, Massachusetts, and Texas. For the sake of heterogeneity we set for each experiment two clouds in the same state, with the other two in different states (*e.g.*, in the WordCount experiment we set two clouds in Chicago). WordCount was executed in clouds different from WebdataScan and MonsterQuery due to frequent maintenance operations in ExoGENI, which prevented us from continuing to use the same clouds. However, the hardware used with all applications was heterogeneous in terms of CPU characteristics and RAM size. We have set one specific cloud in each experiment with better resources than the remaining ones in order to maximize heterogeneity and to demonstrate the benefits of the Medusa scheduler. In the WordCount experiments the best cloud was in Chicago, whereas in the WebdataScan case it was in Pittsburgh.

Each cloud is composed of 4 hosts with a MapReduce runtime: one *resource manager* (master) and 3 *node managers* (slaves). The framework is not modified, which leads to each Resource Manager being a single point of failure *in its cloud*. However, Medusa has the capability to detect which MapReduce runtime is running in a cloud. If not, that cloud is considered faulty (a type of fault our system tolerates). Medusa is installed in the client machine because we assume that the client is always correct. If a client was faulty, the job output and the proxy results would be compromised.

Medusa is implemented in Python 2.7. Medusa is the key component to schedule the jobs and tolerate cloud faults. It submits and coordinates the execution of jobs, makes scheduling decisions, and verifies the integrity of the replicated data and job outputs (checking if $f+1$ replicas are identical and launching new replicas accordingly). These operations require the proxy to communicate with the Resource Manager in each cloud. The proxy is logically located outside the system (in the sense that the clouds are oblivious to it). In our experiments, it runs in the same machine as the client, an Amazon AWS host located in Oregon. The message queuing service used is RabbitMQ (Videla & Williams, 2012). The RabbitMQ server runs in the same machine as the proxy.

Each experiment is configured to first execute the WordCount, WebdataScan, or Monsterquery job and then aggregate the results to obtain the final output. Each experiment is repeated 40 times. The Medusa scheduler relies on a history of 30 executions to train the linear regression model and estimate the throughput between clouds. We use Hadoop's DistCp3 tool[1] to copy data between the HDFS of two clouds. We inject random workloads by running random numbers of extra jobs in each cloud, to simulate background overhead added by other users.

We compare the performance of the Medusa scheduler in terms of job makespan – the time it takes to complete the entire job – and system workload against a baseline that also

[1] Distributed Copy, or DistCp, is a tool that uses MapReduce for large inter/intra-cluster data copying.

tolerates cloud faults but uses a simple strategy to schedule the replicated jobs – a *Round-robin* scheduler. This scheduler selects the clouds to run the replicas of a job in a circular order, assuming the clouds are numbered sequentially.

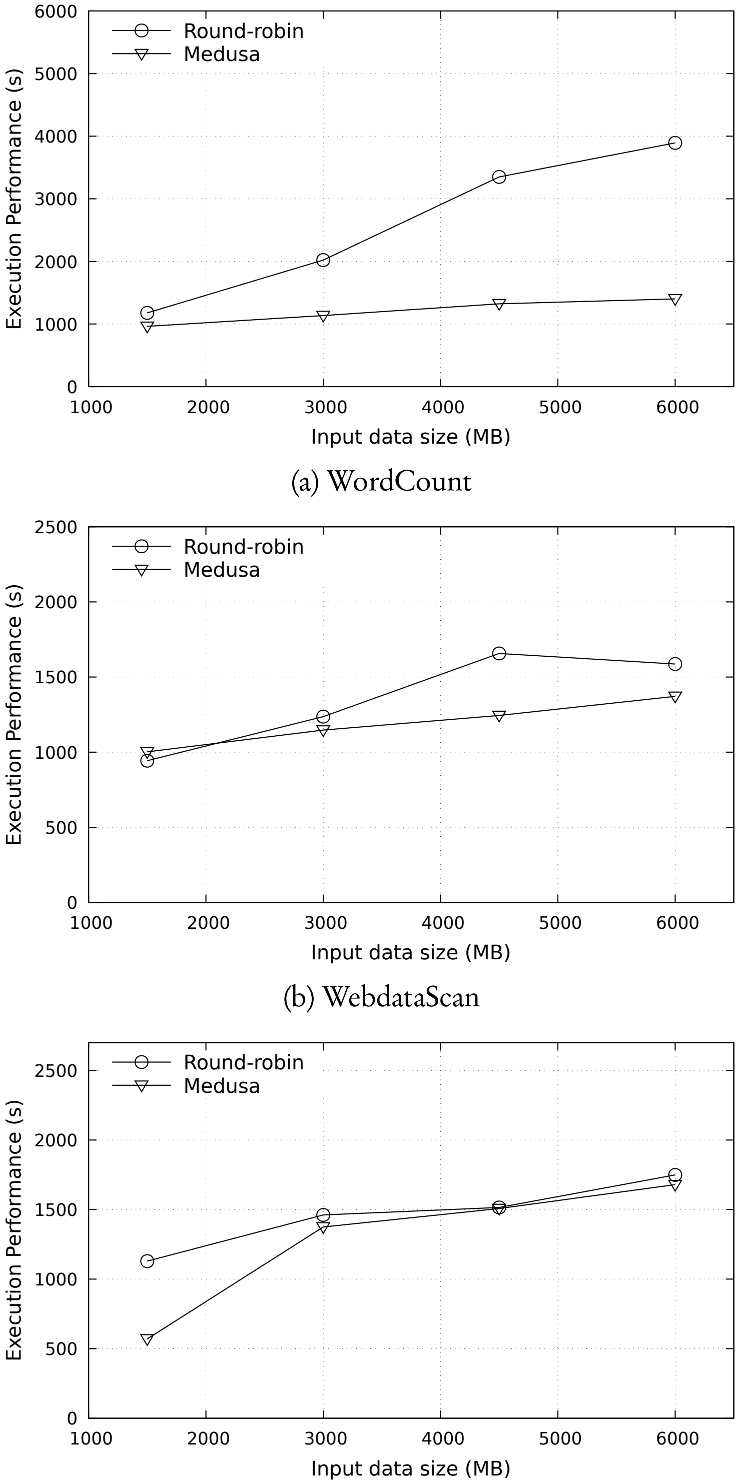
4.4.2 Experimental performance

Performance without faults We first evaluate the performance of Medusa when no fault occurs during the entire job execution. We consider $f = 1$ in our experiments, so each job will be executed twice, *i.e.*, $f + 1$ times. This choice of $f = 1$ is based on the observation that the faults we consider in this work, despite potentially having devastating consequences, are assumed not to be frequent.

Job efficiency. Fig. 4.4 compares the job makespan of the Medusa scheduler against the baseline with different job sizes. We observe that Medusa outperforms Round-robin in all cases. In particular, notice that for the WordCount experiment (Fig. 4.4a) *the Medusa scheduler is up to 3 times faster to complete the job when compared with Round-robin for the larger input sizes*. Moreover, from the bars depicting the first and third quartiles of the makespan in both figures, we conclude that the *Medusa scheduler offers more stable performance* as the variance of the Round-robin scheduler is much more evident (in fact, the variance of the others is so low that the bars are barely perceptible).

The performance gain of Medusa in the other two experiments exists but is not so pronounced because the throughput between clouds is similar. However, in terms of stability Medusa still offers an advantage: the variation in performance of the Round-robin is perceptible in all plots, contrary to our scheme.

Discussion. We now focus on trying to understand the results a bit further. For this purpose we measured the load distribution among clouds with the WordCount application to understand the improved makespan of the Medusa scheduler in more detail. Fig. 4.5 illustrates the usage of each cloud for executing an entire job when different schedulers are used. The Round-robin scheduler distributes the workload evenly across the clouds, ignoring cloud and network performance. In contrast, with the Medusa scheduler the jobs are launched more often in the Chicago clouds, followed by West-Virginia, and only rarely in California. The reason is twofold. First, as can be seen in Fig. 4.6a, which shows the throughput measured between each pair of clouds in the system, the Chicago clouds have better network connections between them when compared to the other clouds. Thus, the



(a) WordCount

(b) WebdataScan

(c) MonsterQuery

Figure 4.4: Execution time of WordCount, WebdataScan, and MonsterQuery (no faults).

transfer time to these clouds is, on average, smaller, and this advantage results in the Chicago clouds being chosen more frequently to run replicated jobs. We observe, however – and that is the second reason –, that although West-Virginia has slightly lower network throughput when compared to California, more jobs are still scheduled to that cloud. This is because the Medusa scheduler considers *both* network transfer time and available computational capacity of the cloud when making its decision. So, in this case, as the relative difference of network throughput is not very high, the cloud with higher computational power (West-Virginia in this case) is capable of offering a lower makespan of the entire job.

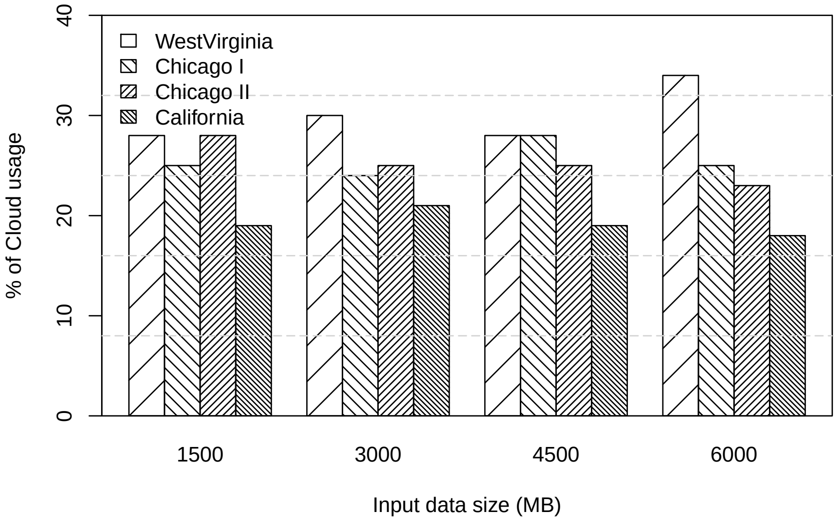
Why does Medusa outperforms Round-robin so considerably in the WordCount experiment, in particular, when compared with the other two? Recall from Section 4.4.1 that we ran the two sets of experiments in different sets of clouds. Contrary to Fig. 4.6a, the throughput measured between each pair of clouds in the WebdataScan and Monsterquery (Figure 4.6b) experiments was very stable. The average throughput was around 1 Gbps.

As is clear, this scenario is more homogeneous in terms of network throughput between clouds, so a Round-robin scheduler performs reasonably well in this setting. The reason why Medusa shows a very significant improvement for the WordCount experiment and less so for the others is therefore related to the heterogeneity of the scenario.

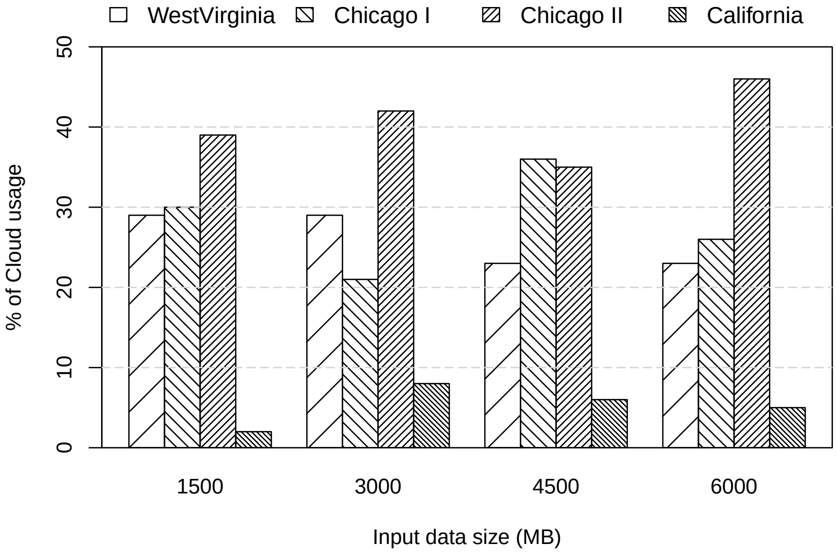
In summary, as we have shown, the combination of the cloud characteristics and network throughput both influence the scheduling decision explaining the best performance of Medusa over a traditional scheduler.

Performance with faults In this section, we evaluate the performance of the Medusa scheduler when there are faults. We consider $f = 1$ in this experiment, as before.

We first evaluate the performance of the Medusa scheduler when a malicious fault occurs. To this end, we inject a fault that corrupts the digest of a job output at the end of the vanilla WordCount execution, forcing the scheduler to launch an extra replica of this job in a different cloud. Fig. 4.7 shows the performance of the Medusa scheduler when one such fault occurs (Medusa w/ malicious faults). In this case, the job makespan doubles when compared to the case with no faults (Medusa w/o faults) as one extra job has to be executed in a different cloud. This is due to the fact that the scheduler can not consider as options the cloud where the job has just run. As explained in Section 4.3.3, that cloud cannot be trusted again.

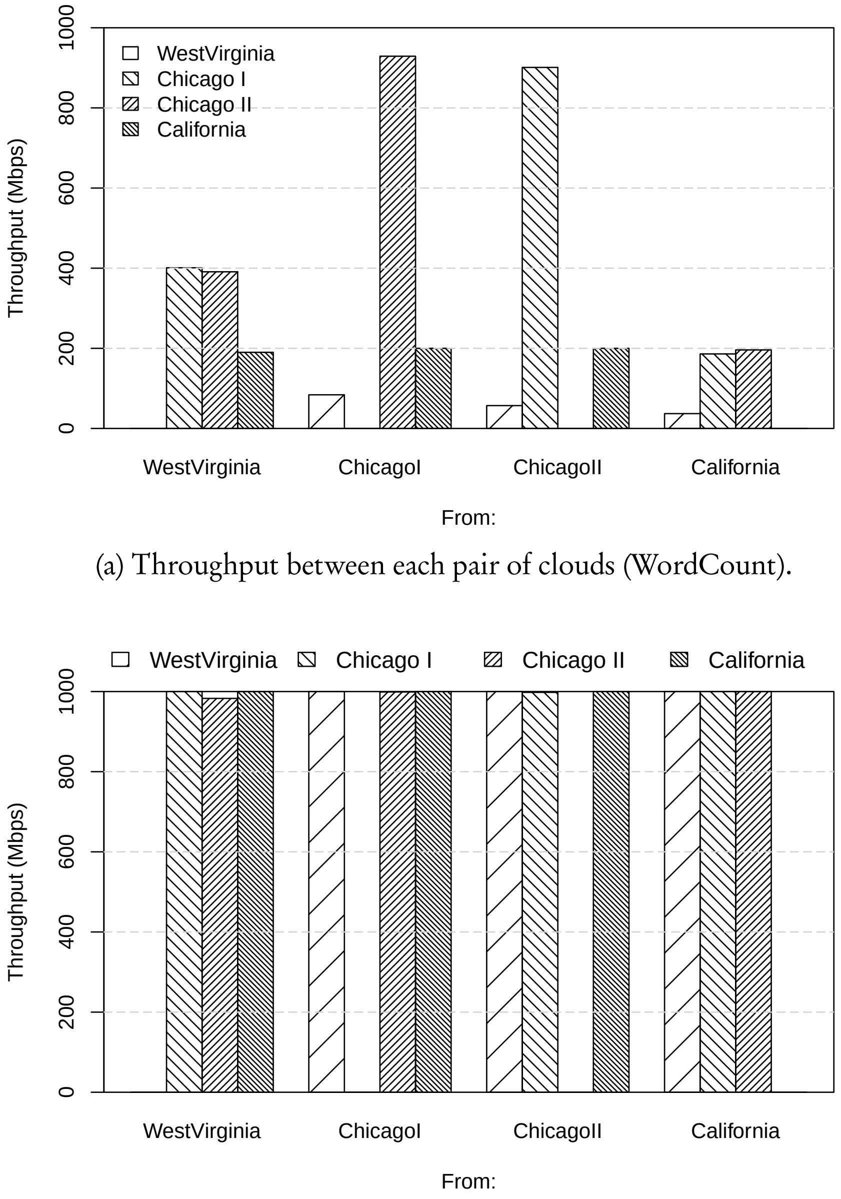


(a) Round-robin



(b) Medusa

Figure 4.5: Percentage of cloud usage with both schedulers.



(a) Throughput between each pair of clouds (WordCount).

(b) Throughput between each pair of clouds (WebdataScan and Monsterquery).

Figure 4.6: Throughput measured between multiple cloud instances.

We also evaluate the performance of the Medusa scheduler by assuming that the fault is arbitrary but not malicious (Medusa w/ arbitrary faults). In this scenario, we also inject a fault that tampers the digest of one job output at the end of the vanilla WordCount execution. The difference, in this case, is that the Medusa scheduler has the possibility to launch the extra replica of this job in the same cloud where the fault occurred (as we assume it is not malicious). As we observe from Fig.4.7, only tolerating arbitrary faults allows reducing the job makespan. This is mainly because the extra replica of the faulty job will with high probability be scheduled to the same cloud, so there is no need to copy the input of this job to another cloud.

Finally, we evaluate the performance of the Medusa scheduler when one cloud outage happens (Medusa w/ cloud outage). To this end, we simulate cloud outages by crashing the resource manager of a random cloud. This forces the scheduler to launch an extra replica of this job to a different cloud. As shown in Fig. 4.7, tolerating cloud outages requires higher job makespan than tolerating arbitrary or malicious faults. This is due to the scheduler taking some time to detect that the cloud has failed, looking for a copy of the data in another cloud, making the necessary data transfers and running the job.

Interestingly, when the input data is large, *the Medusa scheduler in the presence of faults performs better than Round-robin with no faults occurring*. We find it therefore unnecessary to report the performance of the Round-robin scheduler when a fault occurs.

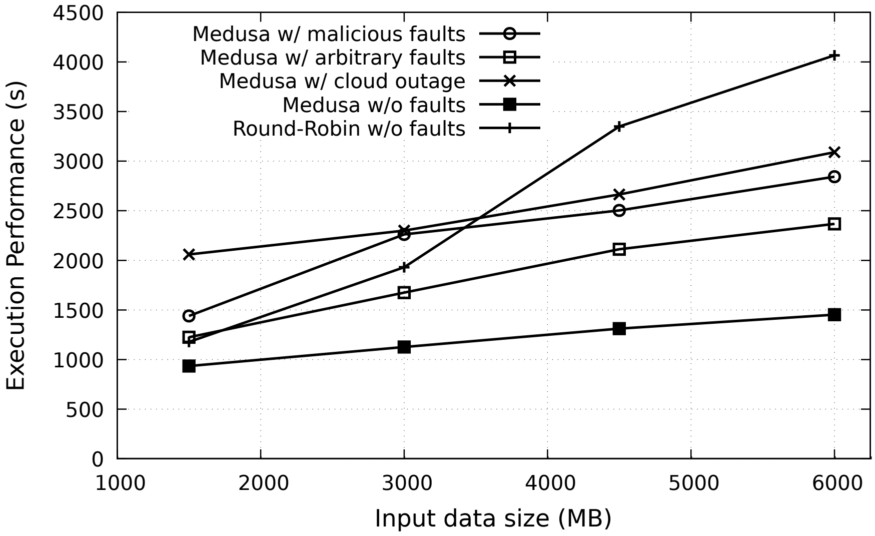


Figure 4.7: Job makespan with one fault injected (WordCount).

To summarize, our experimental evaluation shows that the Medusa scheduler is more efficient than the conventional round-robin alternative, and achieves a significant gain in realistic scenarios of high cloud heterogeneity.

4.5 Summary

In this chapter we proposed a solution, Medusa, for scaling out MapReduce to multiple clouds and, simultaneously, tolerating the new faults introduced by such multi-cloud environment.

This proposal fulfills the objectives that we set forth. First, Medusa scales MapReduce computations to multiple clouds. Second, it extends the fault tolerance offered by MapReduce to tolerate arbitrary and malicious faults, as well as cloud outages. Third, it does so transparently to the user. The Hadoop API is not touched and the existing Hadoop MapReduce programs run without modification. Forth, Medusa is a proxy in the client, and thus the system does not require any modification to the Hadoop framework. The clouds just need to run "vanilla" Hadoop (*e.g.*, Amazon Elastic MapReduce (Amazon, 2015)). Finally, as demonstrated by our extensive experimental evaluation in the ExoGENI testbed, it achieves this increased level of fault tolerance at a reasonable cost when compared with common alternatives. It does so by minimizing replication and by running a novel scheduling algorithm that judiciously chooses the best clouds to perform the necessary replicated jobs.

As future work, we plan to investigate techniques to improve the performance of our system by replicating at a more fine-grained level than at a job level.

5 Fine-Grained Cloud-of-Clouds Dependable MapReduce

In this chapter, we propose a novel execution system that allows to scale out MapReduce computations to a cloud-of-clouds, and tolerate arbitrary faults, malicious faults, and cloud outages. Our system, Chrysaor, is based on a fine-grained replication scheme that tolerates faults at the task level, contrary to Medusa. Our solution has three important properties: it tolerates arbitrary faults, malicious faults, and cloud outages at reasonable cost; it requires minimal modifications to the users' applications; and it does not involve changes to the Hadoop source code. [1] We performed an extensive evaluation of our system in Amazon EC2, showing that our fine-grained solution is efficient in terms of computation by recovering only faulty tasks. This is achieved without incurring a significant penalty for the baseline case (i.e., without faults) in most workloads.

5.1 Introduction

Using multiple clouds to store and compute data can bring many benefits. First, it increases resilience by avoiding single points of failure. A user can be made immune to any single cloud availability zone outage by spreading its services across providers. Second, it can improve performance – for example by taking advantage of data locality (bringing the data to a cloud closer to the user), or by leveraging from the computing and network diversity

[1] We have used Hadoop MapReduce 2.X version.

of multiple infrastructures. Third, it can improve security, for instance by exploring the interaction between public and private clouds. A tenant that needs to comply with privacy legislation may demand certain data to be stored in a specific location (e.g., in a private facility). Finally, it may help in reducing costs, by taking advantage of dynamic pricing plans from multiple cloud providers (Zheng *et al.*, 2015). In this work we provide a solution for doing MapReduce computation on such multi-cloud – or *cloud-of-clouds* – environment, for *fault tolerance*.

We propose a novel MapReduce runtime environment, *Chrysaor*, that scales out MapReduce computations to clouds-of-clouds in order to tolerate arbitrary faults, malicious faults, and cloud outages. Chrysaor is based on a *fine-grained replication* scheme that tolerates faults at the task level. This scheme allows recovering from faults by re-executing only the tasks that were affected. In previous work we have proposed a system, Medusa (see Chapter 4), which shared the goal of allowing MapReduce computations to tolerate the same type of faults. However, Medusa worked at the granularity of MapReduce jobs, resulting in a high cost for fault recovery. For a realistic workload composed of several map and reduce tasks, the single fault in a task would require the whole job to be recomputed in Medusa.

The challenge of achieving the form of fine-grained replication we target in Chrysaor – of tasks, not full jobs – is exponentiated by one of our requirements: not changing the Hadoop source code. Our goal is for our system to use unmodified Hadoop runtimes running in the clouds, including commercial offerings, such as Amazon Elastic MapReduce (Amazon, 2015). As a result, Chrysaor employs a more sophisticated approach that involves the creation of logical jobs to enable the re-execution of MapReduce tasks, in order to reduce the cost of recovery from a fault.

Tolerating cloud faults through replication may be considered expensive as one expects these faults to be rare. The reality contradicts this expectation: cloud outages are becoming very common (Cerin *et al.*, 2013; Clarke, 2015). Moreover, Hadoop MapReduce is increasingly being used for critical applications such as medical research and finance, where any incorrectness or unavailability issues may be intolerable and it is necessary to protect execution against devastating consequences. Cloud providers seem to share this concern: Amazon recently launched Cross-Region Replication to automatically replicate data in different geographical locations (Amazon Web Services, 2015). Motivated by these facts, we believe the cost of replication to be acceptable for such critical applications, in order to guarantee that rare faults with devastating consequences do not occur. Nevertheless, limiting the cost of replication does not cease to be an important goal in our design.

We have performed an extensive experimental evaluation of our system on Amazon EC2. The main conclusion is that Chrysaor is more efficient in terms of computation (number and size of replicas executed) by recovering only faulty tasks, instead of the whole job. This is achieved without incurring a significant penalty for the baseline case (i.e., without faults) in most workloads.

In summary, the main contribution of this chapter, Chrysaor[1], is a system that leverages from several Hadoop MapReduce runtimes spread in different clouds to provide fault tolerance against arbitrary and malicious faults, and cloud outages. Chrysaor fulfills three additional requirements. First, it requires only minimal modifications to the users' applications. Second, it is based on Hadoop but does not involve modifications to the Hadoop source code. Third, it achieves its goals at a reasonable cost, as our experimental evaluation shows. As a result, with Chrysaor users can outsource their critical computations while being assured that the result is trustworthy.

The following sections are organized as follows. In Section 5.2 we describe the system model and define the problem. We present in Section 5.3 Chrysaor by giving a detailed view of the execution during several use cases, and a thorough explanation about the details of implementation. Finally, in Section 5.4, we performed an exhaustive evaluation of our solution in a real test bed in order to understand the gain of the new fine-grained fault tolerance mechanism.

5.2 System model

Chrysaor aims to solve the same problem as Medusa. Chrysaor is distinct than Medusa because it handles faults at task level, so it can react quickly to faults and avoid the overhead of re-executing a whole job.

The system is composed by a set of distributed *processes* (see Figure 5.1): the *client* that request the execution of job, the *proxy* (also called Chrysaor) that submits the job to the *resource manager*, the *resource manager* that governs the execution of jobs and tasks in a

[1] Medusa was a Greek mythology figure that had snakes in place of hair. Chrysaor was one of the sons of Medusa and his name meant "he who bears a golden sword".

cloud, and a set of *node managers* that execute map and reduce tasks. We do not consider the components of HDFS in the model, as the algorithm is mostly orthogonal to that service. For simplicity we consider that each cloud contains exactly one MapReduce runtime.

We say that a process is *correct* if it follows the algorithm, otherwise we say it is *faulty*. We also use these two words to denominate a task (map or reduce) that, respectively, returns the result of applying the map/reduce function to the input (correct) or some other result (faulty). We assume that clients are always correct, because they are not part of the MapReduce execution. If clients were faulty, the job output would be necessarily incorrect. We also assume that the proxy is always correct because it runs at the client side, e.g., in the same host as the client or in a host under the same administration. Resource managers and node managers can fail arbitrarily: they can return wrong results (e.g., processing corruption, or malicious insider) or even stop executing (e.g., due to a cloud outage). A cloud is faulty if it becomes partitioned from the rest of the processes, it is compromised by a malicious attacker, or suffers an outage, a disconnection from the Internet or another severe communications problem.

Our algorithm is configured with two parameters f and t. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas, but in our case the meaning of f is different and weaker: f is the maximum number of faulty replicas that can return the same wrong output given the same input. t is the number of faulty clouds that the system tolerates before the service becomes unavailable. Moreover, O is the output of a MapReduce computation. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is O. If the system selects the correct output by picking the output returned by $f+1$ task replicas, it will never select O' because it is returned by at most f replicas. Similarly to the usual parameter f, our f has a probabilistic meaning (hard to quantify): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible.

The other parameter, t, is the maximum number of clouds that may fail arbitrarily (including outages and malicious faults). We assume there are at least $2t+1$ clouds, to ensure that there are always enough clouds to execute the job.

Chrysaor does not rely on assumptions about processing and communication delays in each stage of job execution, making the system asynchronous. On the contrary, the original MapReduce makes assumptions about such times for termination (e.g., they assume that

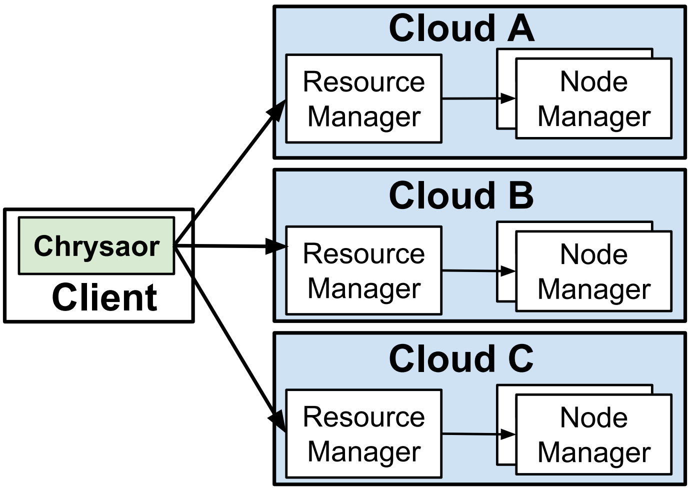


Figure 5.1: Chrysaor interacting with several MapReduce runtime in a multi-cloud system.

heartbeat messages from correct node managers do not take indefinitely to be received). Therefore, Chrysaor also makes these assumptions implicitly (cf. Section 4.2).

We assume that processes are connected by reliable channels, so no messages are lost, duplicated or corrupted. In practice this is provided by TCP connections.

We assume the existence of a hash function that is collision-resistant, i.e., it is unfeasible to find two inputs that produce the same output (e.g., SHA-256).

5.2.1 Problem formulation

We aim at tolerating *(i)* arbitrary and malicious faults, and *(ii)* cloud outages, when running MapReduce jobs in multi-cloud systems. To tolerate f faults, a basic approach is to create $2f+1$ replicas of each task (i.e., the task running in each cloud), spread them in $2t+1$ clouds, and compare the $2f+1$ outputs of each task. If at least $f+1$ outputs are identical, the tasks that produced them must be correct and this must be the correct output, due to the definition of f. The proxy can replicate all tasks, verify all outputs, and obtain the correct output of the job.

This basic approach is expensive in terms of computation. Even if there is no fault, each job is executed $2f+1$ times. Therefore, our objective in this work is to design a proxy that ensures the MapReduce job running in multiple distributed clouds to tolerate cloud faults while *(i)* minimizing the amount of processing; *(ii)* ensuring efficient completion of the entire MapReduce job; *(iii)* and tolerate faults at the task level.

5.3 Chrysaor

To tolerate arbitrary/malicious faults and cloud outages, tasks need to be replicated in a few clouds for ensuring the existence of $f+1$ identical outputs, thus the correctness of the result. Let us first describe briefly how our previous system works (see Chapter 4). Medusa has a proxy that works similarly to a middleware node in a multi-cloud environment. In that solution, a full job execution is comprised of two phases. The first phase runs a *vanilla MapReduce job* in each cloud, which holds a copy of the data to be processed. The second phase runs a *global MapReduce job* that aggregates the outputs from all clouds to generate the final job result. If insufficient identical results are obtained, additional vanilla MapReduce jobs are executed.

In Chrysaor, the proxy still works as a middleware node, but now it has the advantage of just relaunching failed tasks, instead of having to relaunch a whole job in case of no majority.

5.3.1 Chrysaor overview and the logical job abstraction

Our solution involves defining the concept of *logical job*. To perform fine-grained replication, more specifically to re-execute faulty tasks, we would like Hadoop MapReduce to provide an API to control the execution of a job (e.g., to pause it), but such an API is not available. There would be two ways to solve this limitation: *(i)* by modifying the Hadoop source code to provide such control; *(ii)* or to split a job into parts that from the point of view of Hadoop are still jobs (as Hadoop can only execute jobs). We opted for the latter solution.

Chrysaor executes jobs that are divided in two *logical jobs* that are launched and managed by the Hadoop framework in each cloud. From the Hadoop viewpoint, each logical job is a complete MapReduce job. From the Chrysaor viewpoint, the first logical job is dedicated to execute solely map tasks, whilst the second logical job that is dedicated to the execution of reduce tasks is preceded by the execution of identity map tasks. These logical jobs are replicated in different clouds. In case a task fails, Chrysaor creates yet another logical job for that failed task and gives it to Hadoop for execution in one or more clouds.

The end of the Chrysaor map phase corresponds to the end of the first logical job, so by default Hadoop would write the output in HDFS. This has a penalty in terms of performance, so in Chrysaor this output is written in a RAM disk (a virtual disk in RAM memory). The second *logical job* will read the stored data and will execute the reduce tasks. Since it is not possible to run a job in Hadoop starting from the reduce tasks (another API limitation), we need to use *identity map tasks* that output the same input data that reduce tasks consume.

During the execution of map and reduce tasks, each task will generate one digest of the output that our application will use to validate the result. In case there are enough equal results, the proxy will consider that the task has ended successfully. In contrast, if there are not enough equal results, Chrysaor creates a new logical job to run just the faulty task(s).

5.3.2 Chrysaor operation

Chrysaor has the capability to deal with accidental and malicious faults in the map and reduce tasks. In this section we explain the operation of Chrysaor in steps: first without faults, second with arbitrary faults, and finally with malicious faults. We consider $f = 1$ and $t = 1$ in the examples.

Chrysaor without faults Figure 5.2 depicts a successful job execution in Chrysaor without faulty tasks. The scenario contains two MapReduce runtimes in two clouds. It assumes that input data is replicated in both clouds before the execution begins.

When the client (not represented in the figure) requests Chrysaor to execute a job, $t + 1 = 2$ clouds (clouds A and B) are selected by the proxy (Chrysaor in the figure) to run the first logical job (step 1). Chrysaor executes $max(f,t)+1$ replicas of each logical job, which in this case means 2 replicas, one per cloud as $f = t = 1$ (*max* returns the maximum of two numbers). During execution, each map task creates a digest of the map output. The digests are fetched (step 2) and compared by Chrysaor to check if all map task replicas produced the same results (step 3). This is the case (we are considering no faults), and so the second logical job is launched (step 4).

The second logical job will read the data that was stored previously, run the identity map tasks, and do the shuffle&sort phase before the reduce tasks start. The reduce tasks will produce the final output and the corresponding digest that will be fetched by our system

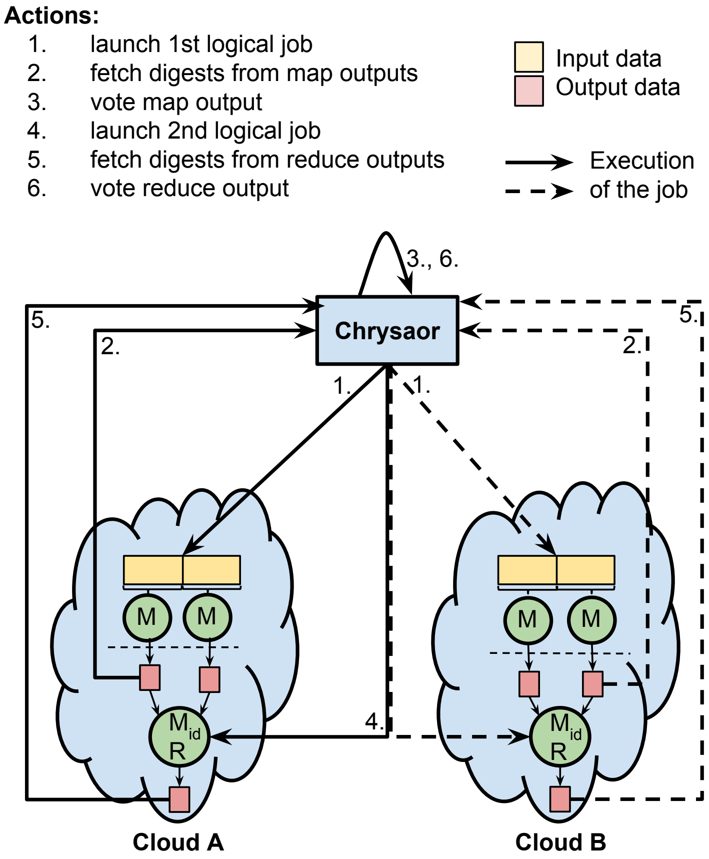


Figure 5.2: Chrysaor executing a job in two clouds without faults

(step 5). Again, Chrysaor will compare the results (step 6). As there are no faults, the results are the same and the job execution terminates successfully.

Chrysaor with arbitrary faults This section explains the case when a map or reduce task returns a wrong output. The alternative case of a task not returning a result at all is not as interesting because it is handled autonomously by the Hadoop runtime: the resource manager of the cloud where the task is being executed simply re-executes the task in the already selected clouds.

Chrysaor detects that the result of a task is wrong when it observes that replicas of the task return different results, i.e., that there are no $max(f,t)+1$ equal results (2 in the example). Notice that it does not know which of the replicas is faulty, only that one of them is faulty as there is disagreement on the result. In that situation, Chrysaor creates a logical job with that task in both clouds and re-executes it.

Chrysaor cannot differentiate if a fault that affected the task was accidental or malicious. If it was accidental, the re-executions may be done in the same clouds: if the faults are intermittent, they will eventually no longer affect the same tasks; if they are permanent, the Hadoop runtime will eventually choose other node managers (and other nodes). If the fault is intentional – malicious – it may affect a whole cloud so it is advisable to use another cloud.

Chrysaor allows configuring how to deal with such faults, but the key idea is that it can try a few times to re-execute the tasks in the same clouds, then pick additional clouds if no agreement is reached. Specifically, there is a threshold T_{faults} that is defined by the user. When T_{faults} occur, Chrysaor considers that there may be malicious faults and it picks an additional cloud to execute tasks (next section).

If a faulty result is detected at the end of the second logical job, it is necessary to relaunch the faulty reduce task. This involves re-running not only the reduce task but also the identity map tasks that precede it. The job ends successfully when all reduce task replicas produce the same result.

Chrysaor with malicious faults When Chrysaor is dealing with malicious faults or cloud outages, it has to execute tasks in another cloud until it obtains $max(f,t)+1$ equal results. As already explained, Chrysaor is not able to detect that there is malicious behavior; it simply starts using a new cloud when the threshold T_{faults} is exceeded.

Figure 5.3 depicts the re-execution of a job when map tasks suffer a malicious fault and T_{faults} is exceeded (e.g., because $T_{faults} = 0$). Something similar would happen if there was an outage. Cloud C is going to be used to execute the extra replica. At the end of the first logical job (step 3), there is a task that did not return equal results forcing the system to re-launch the same job in cloud C (step 4). The validation of the digests in cloud C (steps 5, 6) will allow the system to obtain the correct result and show that A may be malicious as it did not provide the correct result. Again, at the end of the execution of the second logical job, Chrysaor reads the digests (step 7). If the execution has ended correctly, the solution as the capability to validate the results and find which cloud is compromised.

In case of a malicious fault at the end of the second logical job, i.e., in the reduce tasks, it is necessary to execute a new full job in the new cloud, and wait to validate the output result. This is the worst case scenario in terms of performance. If more clouds were necessary and were not available, Chrysaor would abort the execution and inform the client.

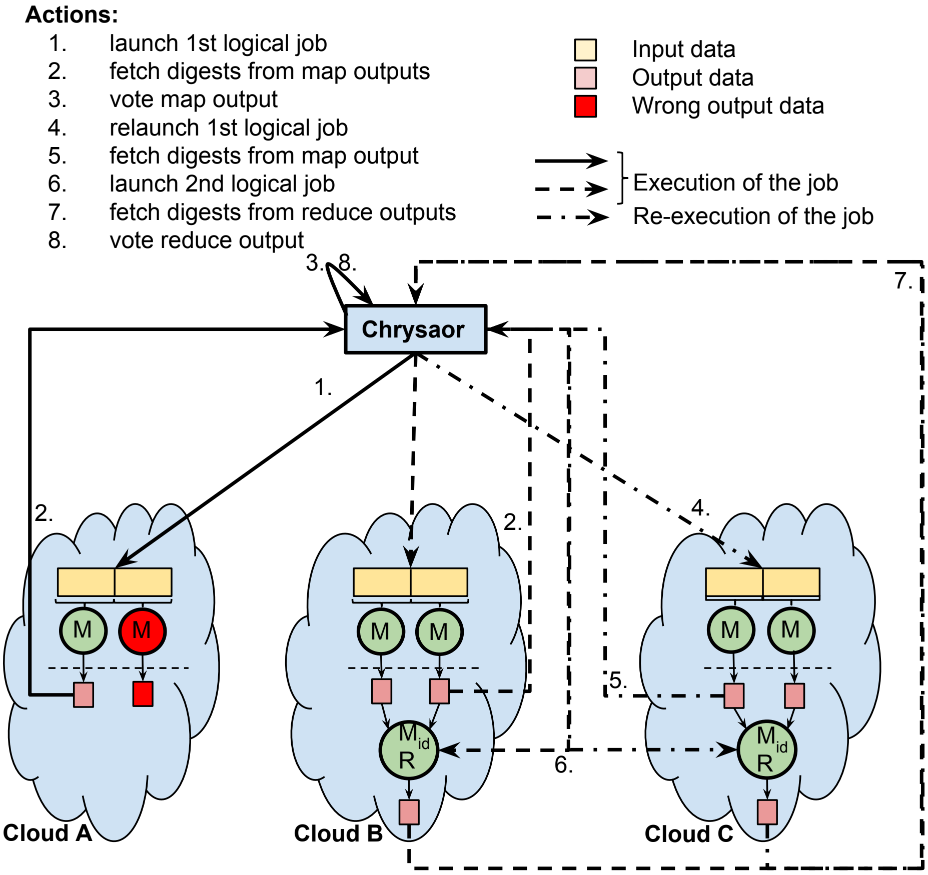


Figure 5.3: Chrysaor executing a job in two clouds with a map task re-execution in another cloud due to a fault

5.3.3 Chrysaor implementation

So far we have explained Chrysaor's design. Now we detail its implementation.

The proxy (Chrysaor) is installed in the client machine. It interacts with the resource manager in each cloud using the RabbitMQ message broker (Videla & Williams, 2012). The proxy has the capability to tolerate cloud outages by triggering a timeout when a connection to a resource manager is detected.

Besides the proxy, Chrysaor provides a Java library that is installed in each resource manager. Although Hadoop supports map and reduce functions written in a few languages, the current implementation of Chrysaor supports only Java. Hadoop is not modified, which leads to each resource manager still being a single point of failure *in its cloud*. The framework is configured to store temporary outputs in RAM disk.

The server-side Java library has the goal of intercepting certain calls done by the executing jobs to the Hadoop API in order to execute Chrysaor server-side code. The calls are intercepted using AspectJ during the execution of the MapReduce job. AspectJ is an aspect-oriented extension to Java that allows, essentially, adding hooks that force calls to external methods in certain conditions, without requiring modifications to the original source code (Kiczales *et al.*, 2001).

For an user to take advantage of Chrysaor, she has to do minor modifications to her application. The user does not need to modify the Java code of the map and reduce functions to use Chrysaor. The updates are related to the definition of the identity map function. The user can create his own identity map code, or use the template that is available in the Chrysaor library. The identity map code has to take as inputs a key and values with the same types (classes in Java) as those returned by the job's map function.

Listing 5.1 shows an example identity map function for the WordCount application (Apache, 2016). Specifically, in Line 4 we define a simple identity map method that reads a key and values with expected type (respectively Object and Text) and produces identical output. In the WordCount application, the key and the value of the map output come in the *value* parameter. The value of the variable is split and set in the *word* attribute.

During the creation of the job in the main class, it is necessary to set the identity map class. In Line 14, we set our own identity map class in the job configuration using Hadoop *JobConf* variable. *JobConf* is the interface for a user to describe a map-reduce job to the Hadoop framework for execution.

Chrysaor intercepts *write* calls that are made inside the map and reduce functions to update the digest for each key and value produced by the task (Apache, 2016). When the task ends, it invokes the *cleanup* method. The *cleanup* call is intercepted by the Chrysaor server code to save the digest locally. The set of digests produced by the replicated tasks will be used to detect incorrect results.

The job is launched using the *run* method from the *JobClient* interface provided by the Hadoop API. *JobClient* provides facilities to submit jobs and track their progress.

Listing 5.1: Definition of the identity mapper.

```
 public static class MyFullyIndentityMapper
  extends Mapper<Object, Text, Text, IntWritable>{
    private Text word = new Text();
    public void map(Object key, Text value, Context context) throws ↩
   IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
        }
    }
}

public static void main(String[] args) throws Exception {
    (...)
    conf.setClass("mapreduce.job.map.identity.class", ↩
   MyFullyIndentityMapper.class, Mapper.class);
    JobClient.runJob(job);
    (...)
}
```

5.4 Evaluation

The objective of our experimental evaluation is to answer the following questions: (1) How does the performance of Chrysaor compares with its nearest system, Medusa, in a baseline scenario without faults? How does Chrysaor behaves compared with the vanilla MapReduce? (Section 5.4.2); (2) What is the gain of the fine-grained fault tolerance introduced in Chrysaor? (Sections 5.4.2 and 5.4.2); (3) How does the type of job affect Chrysaor's performance?

Section 5.4.1 describes the experimental setup as well as the configuration of our solution. Section 5.4.2 reports on the performance of Chrysaor, considering both the presence and absence of faults during job execution. Finally, in Section 5.4.3, we perform an analytical evaluation to understand in more detail the costs introduced by our system.

5.4.1 Experimental setup

To answer the aforementioned questions we evaluate our solution in a real-world scenario. We have configured a testbed in Amazon EC2 to run all experiments. Amazon EC2 provides a distributed networked infrastructure-as-a-service for computation and storage in the cloud. We set up four clouds located in different sites to evaluate our solution. We used clouds located in Oregon, North Virgina, California and Ireland. The hardware used in all applications was diversified in terms of CPU characteristics and RAM size. Each cloud was composed of 4 hosts with a MapReduce runtime: one resource manager (master) and 3 node managers (slaves). The framework was not modified, which leads to each Resource Manager being a single point of failure in its cloud. However, Chrysaor has the capability to detect MapReduce failures.

We considered three common applications provided by Hadoop's Gridmix benchmark (Apache, 2013): *WordCount*, *WebdataScan*, and *Sort*. This choice aims to ensure application diversity in terms of communication and computation requirements. We have applied to all the applications the slight modifications required to run our system (cf. Section 5.3.3).

We run each experiment 10 times, reporting in the figures the average results, and the 5th and 95th percentiles.

WordCount. The first application we evaluated is related to Web indexing. Running *WordCount* in a multi-cloud system can be considered as building the inverted indexes of a multi-site web search engine for each search site (i.e., cloud). Each cloud runs a local MapReduce job to parse the documents, and to build a search index that supports frequency-inverse document frequency style ranking functions (TF-IDF). This can be achieved by running WordCount as a vanilla MapReduce job in each cloud. To ensure the same search results can be retrieved as in a single-cloud search engine, the input data need to be replicated in each cloud.

WebdataScan. This application has the goal to extract value from big data, an increasingly important tool for decision-making. *WebdataScan* extracts a small amount of relevant data from a large data set, which is a common form of processing and data analysis in many systems. The map tasks keep just a small fraction of the data (0.2%) and the reduce tasks again return just a small part of their input (5%).

Sort. The last application we evaluated does sorting of big data, another typical use of MapReduce. *Sort* is an example of a benchmark that is computationally-intensive (rather than communication-bound). In this application the intermediate key/value pairs are processed in increasing key order. This ordering makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, for instance.

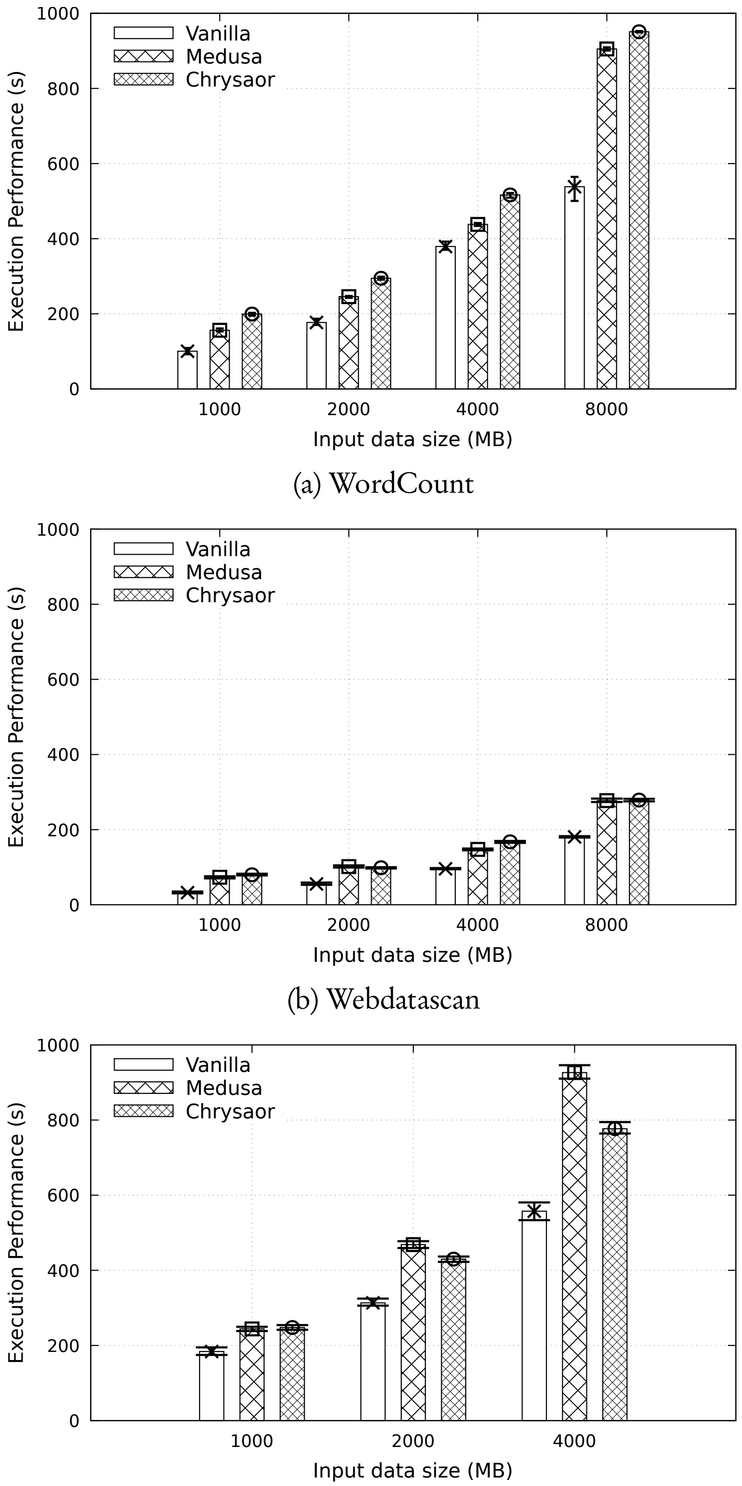
To simulate a real-world scenario, we have set up three clouds located in different Amazon EC2 sites (Oregon, North Virginia, and California). Each cloud is composed of one *resource manager* (master) and four *node managers* (slaves). The hosts are general purpose instances that provide a balance of compute, memory, and network resources. Each instance contains a 2.3 GHz Intel Xeon E5-2686 processor for a total of 8 vCPUS per server. Each server has 12GB of memory, and 150GB of Elastic Block Store (EBS) space. The clouds are protected from the outside world using firewalls, so it is not possible to access them without proper credentials or specific configurations.

We compare the performance of Chrysaor and Medusa in terms of the time it takes to complete the entire job (makespan). Medusa is the only system to tolerate arbitrary and malicious faults, and cloud outages (at the job level). So we use it as a baseline.

Based on the observation that the faults we consider in this work, despite potentially having devastating consequences, are rare, we consider $f = t = 1$ in our experiments.

5.4.2 Experimental performance

Performance without faults We start by analyzing the performance of Chrysaor without faults. In Figure 5.4a, we check the performance of the WordCount application with several sizes of input data, ranging from 1GB to 8GB. The choice of these input sizes is based on the fact that the MapReduce jobs that run in Microsoft or Yahoo! production clusters typically operate over input sizes under 14GB (Rowstron *et al.*, 2012). By comparing Chrysaor with vanilla MapReduce, we see that Chrysaor was 27% to 50% slower. As we use the same amount of resources in the evaluation, and Chrysaor needs to execute double of the tasks, we have in the best case less than the double of the performance. When compared with Medusa, we see that Chrysaor got slightly worse results. The reason is that in WordCount the map tasks produce a map output larger than the input data, so the identity map tasks will compute large data, with a considerable overhead. As result, Chrysaor is 5% to 27% slower than Medusa in the reported cases.



(a) WordCount

(b) Webdatascan

(c) Sort

Figure 5.4: Detail of job execution without faults

The WebdataScan application is mostly centered in the map side. As such, less time was spent in the second logical job. As consequence, a full execution in Chrysaor spends similar amount of time in the map and reduce tasks as in Medusa. In other words, the identity map tasks and the digests produced while the tasks run did not introduce visible delay as in the WordCount application. Thus, we can see in Figure 5.4b similar results between both solutions. When we compare Chrysaor with vanilla MapReduce, we see that our solution was 35%-56% slower. Most interesting, as we increase the input size, the difference between both solutions is decreasing.

In the Sort application (see Figure 5.4c), Medusa got worse results in comparison to Chrysaor, due to the fact that generating digests after job execution in Medusa delays the entire execution (it involves invoking an HDFS command to access a file). In contrast, the digests are generated while the output is being produced in our new solution. Due to the large output that is produced, the characteristics of Chrysaor wins over Medusa. For instance, in the case of 4GB, Chrysaor was 16% faster. It was not possible to run a use case with 8GB of input data due to lack of memory. Anyway, the trend is clear in showing the advantage of Chrysaor for this type of application. In comparison with vanilla MapReduce, Chrysaor was 25% to 30% slower. Overall, generating a digest while the output is being produced is shown to be better than generating a digest after the reduce tasks finish. Importantly, this gain would not be made possible in Medusa. It is the new architecture introduced in Chrysaor of a fine-grained approach that allows this optimization. When we compare Chrysaor with vanilla MapReduce, we see that our solution is slower 25%-30%. Although Chrysaor has performed twice as much computation, it takes only a quarter of the time more. This result shows how the framework architecture is adequate to this type of job.

In summary, the main cost of Chrysaor are the identity maps. When we compare Chrysaor with Medusa, we notice that for applications that require identity map tasks that handle large amounts of data, as in the WordCount example, the penalty introduced by our system is non-negligible. In contrast, applications such as WebdataScan that spend less time executing identity map tasks, do not suffer and behave similarly to the baseline case. Finally, the Sort application is an example of a class of MapReduce jobs where Chrysaor improves over Medusa even in the baseline case without faults. In particular, when the cost of generating digests is high, the advantage introduced by our architecture of enabling this cost to be amortized as the system runs results in an effective gain. When we compare with vanilla MapReduce, we realize that despite Chrysaor executing the double of tasks, our solution can take just 25% more time in some examples.

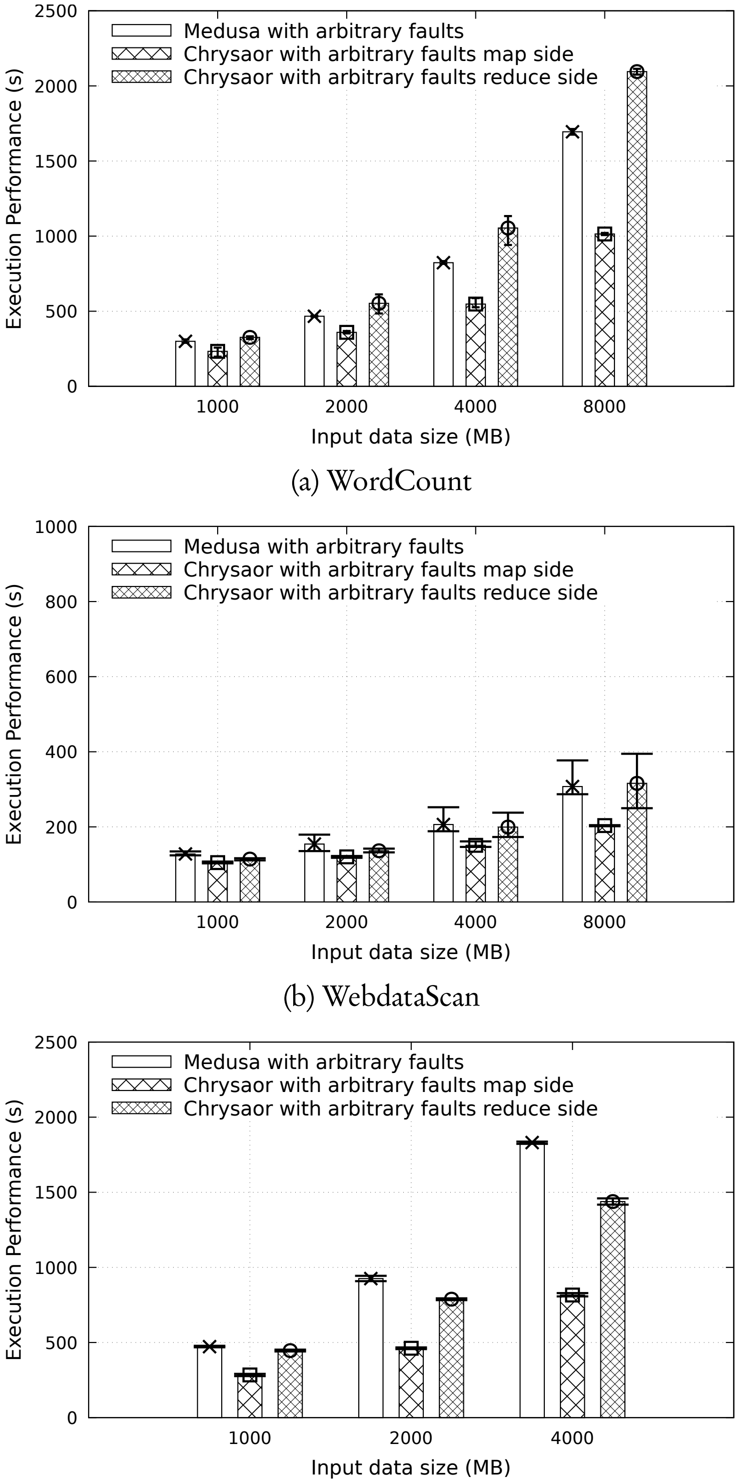
Performance with arbitrary faults In this section, we want to understand the behavior of the system when arbitrary faults occur in the map and reduce tasks. Medusa and Chrysaor behave differently when they are dealing with arbitrary faults. The faults were injected using a configuration setup that tampers randomly the digests of the map or reduce tasks. In the case of Chrysaor we leveraged AspectJ to inject the faults.

Chrysaor is the only system that responds immediately to a fault at the task level, so its behavior is different if the fault affects a map or a reduce task. When an arbitrary fault happens in a map task, the corrupted task will be relaunched in the same clouds. When it happens in a reduce task, it is necessary to relaunch again all the identity map tasks to re-execute the faulty task(s). When dealing with arbitrary faults, we consider the threshold T_{faults} to be greater than 1 in order to re-execute the faulty task in the same cloud. Medusa always re-executes the full job when there is a fault, so its performance is the same if the fault compromises a map or a reduce.

Figure 5.5 depicts the execution time in the case of accidental faults for the three applications. When a fault happens in the map side, *Chrysaor has always the best performance*, up to 56% better than Medusa. Dealing with faults at the task level brings this important benefit.

When we analyze the reduce tasks, we have different results depending on the application. In the case of WordCount (see Figure 5.5a), Chrysaor was slower when a fault happened in the reduce tasks, whereas in the case of WebdataScan (Figure 5.5b) we see that both systems reached similar results. In this application, re-executing our solution with the identity tasks took the same time as re-executing the full job. Finally, in the Sort application (Figure 5.5c), Chrysaor was always faster. Again, in the case of an arbitrary fault generating the digests whilst the output is being produced continues to be a better solution than generating the digest at the end of the job.

One important result from this analysis is the fact that Chrysaor was always faster when the faults affected map tasks. In most MapReduce jobs the number of map tasks (one per input slice) is much larger than the number of reduce tasks, which means that in the common case our solution will outperform Medusa in the presence of arbitrary faults.

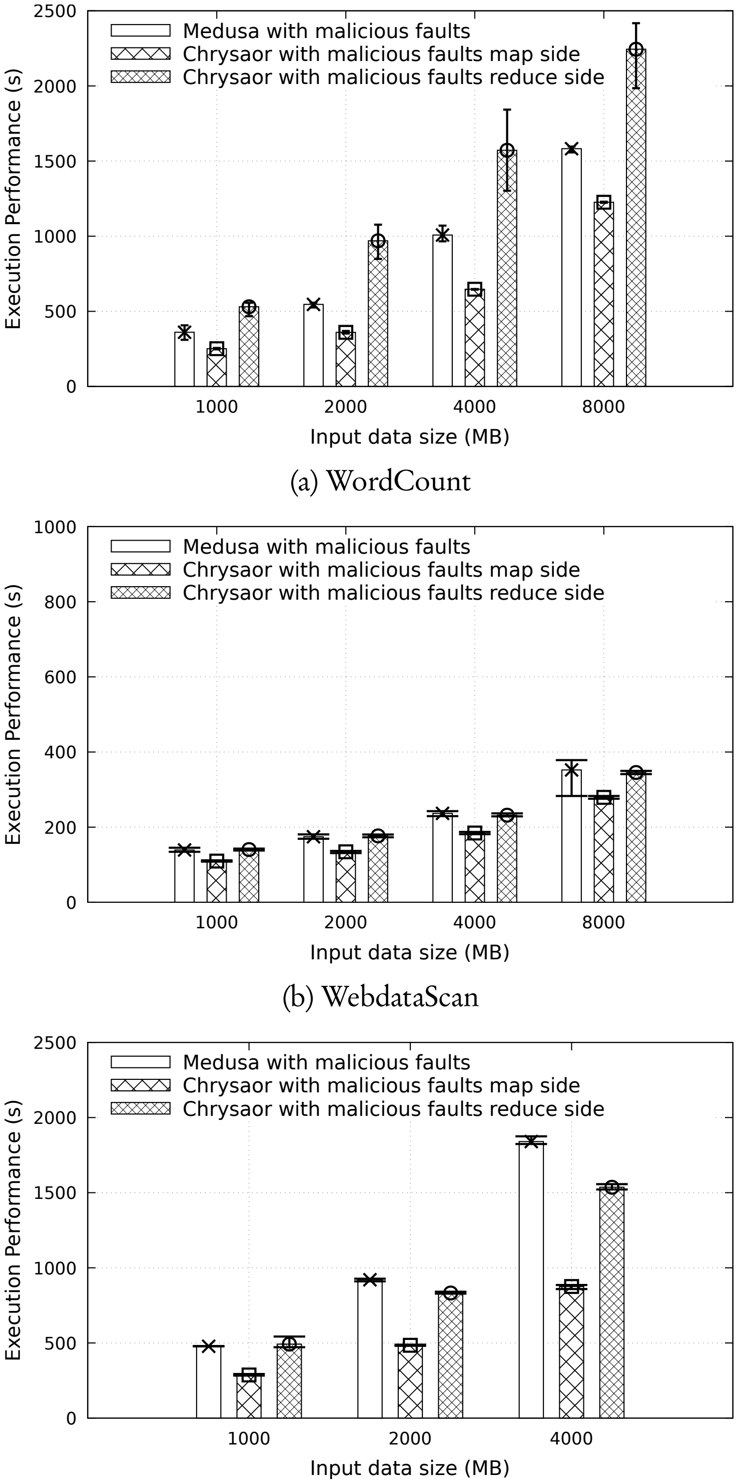


(a) WordCount

(b) WebdataScan

(c) Sort

Figure 5.5: Detail of job execution with arbitrary faults



(a) WordCount

(b) WebdataScan

(c) Sort

Figure 5.6: Detail of job execution with malicious faults

Performance with malicious faults When the system deals with a malicious fault, it re-executes the faulty tasks in a new cloud (as at least one cloud is deemed malicious). In this section, the system used $T_{faults} = 0$ in order to use immediately a new cloud without trying first re-execution in the same clouds.

There are different scenarios of execution when we test Chrysaor with malicious faults. In case of a malicious fault in the map side, the algorithm chooses to execute all the map tasks in a new cloud. It is only after this process that the system can check which cloud is compromised, and decide to continue with the correct ones. When a malicious fault is detected in the reduce tasks, a new cloud is chosen to execute the whole job before finding the compromised cloud.

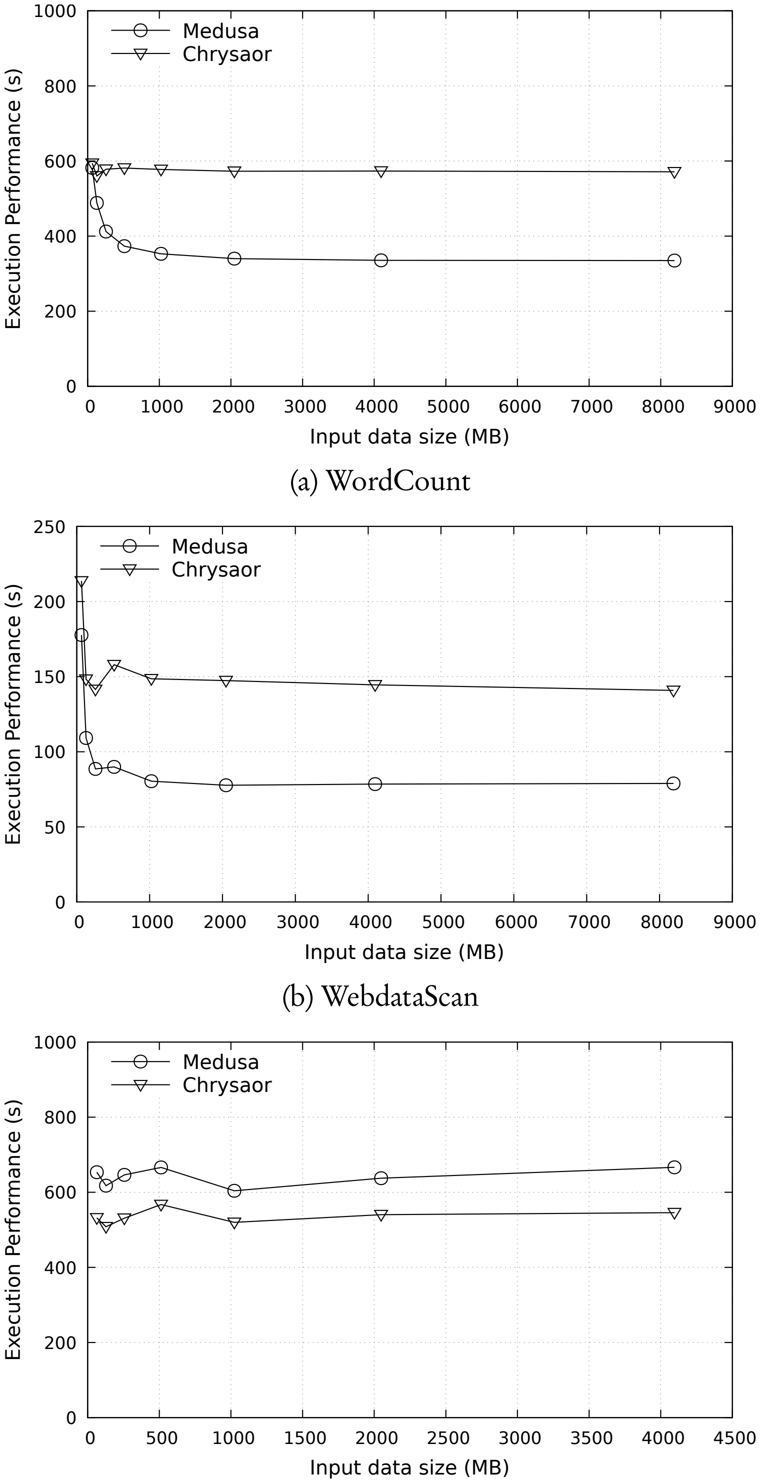
The results are presented in Figure 5.6. As in the previous section, Chrysaor got the best results when dealing with faults that occurred in the map tasks. Again, our solution can be up to 60% better when compared with Medusa. As before, the results are not as positive when a fault happens in the reduce tasks. Observing Figure 5.6a, we see that Chrysaor was slower than Medusa. The reason is the need to execute a large identity map task in the new job that runs in the additional cloud.

In the WebdataScan application (see Figure 5.6b), we confirm the similarity of results between Medusa and Chrysaor. This is due to the identity map tasks not being the most important component of the overall execution time.

Finally, in Figure 5.6c we see that the Sort application had better performance in Chrysaor than in Medusa for all cases. In this case, the Sort application was 2% to 27% faster in Chrysaor.

The conclusions to draw in the experiments with malicious faults is similar to those of the arbitrary case. Namely, Chrysaor is always the best solution when dealing with faults that occur at the map tasks. However, in case of faults in the reduce tasks, Chrysaor is only favorable for workloads that do not involve large execution in the identity step.

Performance with different input size We wanted to evaluate the system as we vary the size of the split, i.e., of the input for every map task. For the WordCount and WebdataScan, we have set the split size from 64MB to 8GB, and for the Sort application from 64MB to 4GB. The size of the input file for the WordCount and WebdataScan is 8GB, and for the Sort is 4GB. We have run these applications without faults.



(a) WordCount

(b) WebdataScan

(c) Sort

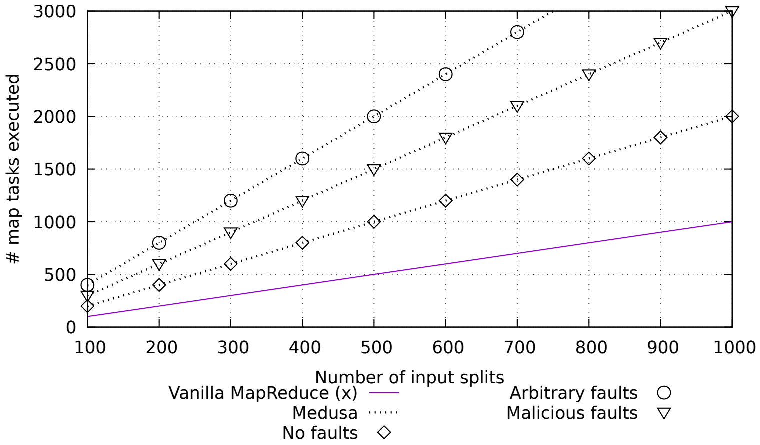
Figure 5.7: Detail of job execution with different input split sizes (without faults)

In the WordCount application (see Figure 5.7a), Medusa tends to improve performance as the size of each split increases until 1024MB. After that, the performance tends to stabilize. In contrast, Chrysaor always presented similar performance results despite the input size. By looking into the details of Chrysaor jobs, we observed that the time to execute the second part of the job remained the same despite the input size. What has decreased in terms of execution performance was the first logical job. Since the first logical job is 20% time faster than the second part, the difference of results between each split is negligible in Chrysaor. As result, Medusa starts to present similar results to Chrysaor when a split has the size of 64MB and ends being 60% faster than Chrysaor when dealing with splits of 8GB (there are no faults). We have claimed that having identity map tasks in Chrysaor was the main reason for Chrysaor to be slower than Medusa, and we can add that the performance difference remains the same despite the number of splits.

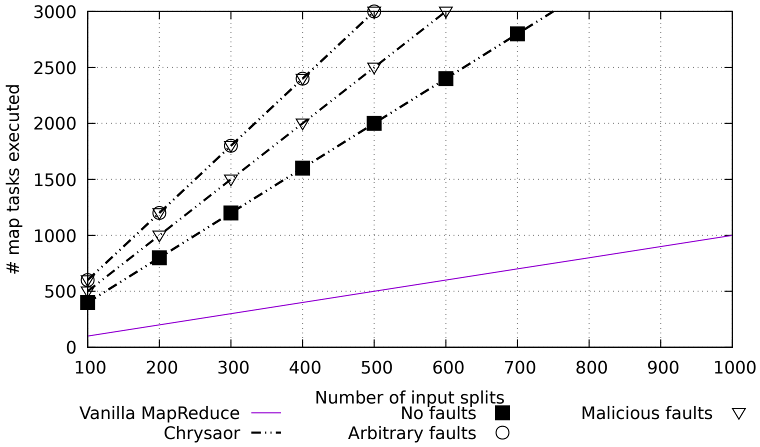
In the WebdataScan application shown in Figure 5.7b, Chrysaor starts to be 17% slower than Medusa when we are dealing with splits of 64MB. Chrysaor is 27% slower than Medusa in the case of 128MB splits. Finally, Chrysaor tends to be twice more slower as we increase the split size. The main reason for Chrysaor cost is the need of having the second logical job. Since most of the computation in this application is in the first part, launching a second job to run the reduce tasks is not the best strategy for this application. Again, the use of identity map tasks delayed the performance.

In the case of the Sort (see Figure 5.7c), Chrysaor was always 16%-19% faster than Medusa. This difference in the execution between both solutions remained the same independently of the split size. We realized that generating digests in Medusa at the end of the job got an impact of 55% in the overall execution. As result, Medusa got much worse performance than with Chrysaor.

Overall, Chrysaor obtained better performance in the Sort application and lost to Medusa in the WordCount and WebdataScan application. As conclusion, we can say that the split size did not affected Chrysaor's performance, because of the existence of the identity map tasks. What affected the performance of Chrysaor is the existence of identity map tasks. If we removed this type of task by updating the MapReduce source code, we would have much better results but we would be constrained to the MapReduce version. This is a trade off that we must take into consideration when analyzing these results. Since most of jobs use a split size from 64MB to 128MB as the default block of HDFS has 64MB (White, 2009), the difference between performance between both solutions is much smaller than when we



(a) Number of map tasks executed versus number of splits ($f = t = 1$) with Medusa



(b) Number of map tasks executed versus number of splits ($f = t = 1$) with Chrysaor

are dealing with splits sizes with bigger size. Therefore, we consider Chrysaor achieved acceptable results in these scenarios.

5.4.3 Analytical evaluation

Chrysaor needs more computation in most of the cases when dealing with faults in comparison with Medusa. This is the result of the solution architecture and the introduction of identity map tasks. In this section, we evaluate the computation costs analytically by calculating the number of tasks executed versus the number of input splits.

In the Figures 5.8a and 5.8b we present a simulation of the number of map tasks needed to process a certain number of input splits. We consider the fact that a map task processes one split. We calculate the number of map tasks from 100 to 1000 splits. We also set $f = t = 1$, like we did in the experiments, to calculate the number of map tasks needed to run in the Medusa platform.

No faults. The vanilla MapReduce executes every task only once when there are no faults. For instance, if we have a job that is composed by 100 splits, the framework will run 100 map tasks to process the job. Since the number of map tasks is proportional to the number of splits, it is represented in the graph as a positive line with slope $m = 1$ (bottom line). This runtime does not have any mechanism to tolerate other types of runtime faults besides crash faults.

With Medusa, it is necessary to execute $f+1$ job replicas in the case of no faults, making a total of 200 tasks. This solution requires to launch the double of tasks than in the original Hadoop, which represents a slope $m = 2$ (second line from bottom).

In Chrysaor, it is necessary to launch $f + 1$ map tasks and $f + 1$ identity map tasks to run the reduce tasks. Therefore, 400 tasks must be launched to execute a job with the same number of splits. As result, we have a slope of $m = 4$. Notice however that the identity maps do not process the data, only return what they receive as input, therefore their processing time tends to be lower than the rest of the maps. In fact, if we did not consider these maps, we would have the same number of maps as in Medusa (200 maps for 100 splits).

Arbitrary faults. In case of arbitrary faults, and assuming that these faults only happen once, it is necessary to execute $2f+2$ jobs in Medusa. Because Medusa cannot differentiate if a fault has happened in a map or reduce task, it is necessary to re-launch a full job. This produces a line with slope $m = 4$ in the figure.

In Chrysaor, when an arbitrary fault happens in the map tasks, it is necessary to launch just the failed map task $f + 1$ times. As result, the execution time is similar to the case of no faults (slope $m = 4$). In fact, Chrysaor is the solution that offers better results when dealing with faults at the map side. When there is a faulty task at the reduce side, it is necessary to re-run the identity map tasks to just execute the faulty reduce tasks. In the end, Chrysaor needs to execute $3f + 3$ tasks, which is $f + 1$ more tasks than in Medusa. This makes a slope of $m = 6$. Again, this is counting with the identity maps. Without these less expensive maps we would fall on the $m = 2$ line.

Malicious faults. In case of malicious faults, Medusa needs to execute at least $2f + 1$ jobs to terminate the computation. In this case, it is necessary to launch 3× more tasks than the vanilla MapReduce, which makes a slope of $m = 3$. In contrast, Chrysaor behaves differently.

If a malicious fault happens in the map tasks, i.e. after the execution of the first logical job, it is necessary to launch a new job in a new cloud. Therefore, $2f+3$ tasks need to run. Notice that the second logical job in the new cloud also needs to execute the identity map tasks. That is the reason that we have $2f+3$ and not $2f+1$. If a malicious fault happens in the reduce tasks, it is necessary to execute a whole new job when a fault is detected. In the end, we have a total of $2f+4$ map tasks to run, which means that it is necessary to execute 3× more tasks with our solution than with Medusa. Again, this is counting with the identity map tasks.

Fault / Solution	No	Arbitrary		Malicious	
		Map	Reduce	Map	Reduce
Hadoop	1	-	-	-	-
Medusa	$f+1$	$2f+2$		$2f+1$	
Chrysaor	$f+1$	$f+1$	$3f+3$	$2f+3$	$2f+4$

Table 5.1: Number of jobs to launch depending on f

In Table 5.1, we sum up the number of jobs that each solution needs to launch according to the use case. Analyzing the values in the table, we can see that Chrysaor needs to do more computation when dealing with certain type of faults. If we compare the values of Chrysaor in Table 5.1 with our evaluation, we notice that the relation between the performance and the analytical evaluation is not proportional. For instance, Chrysaor needs to execute $f+1$ more tasks when dealing with arbitrary faults in the reduce side in comparison with Medusa. But our evaluation showed that both solutions presented similar results with the WebdataScan application. This contradicts the expectation that Chrysaor were $f+1$× slower than Medusa. This analysis is extended to all use cases.

The need to use identity map tasks affected Chrysaor performance. If we removed this type of task by updating the MapReduce source code, we would have much better results but we would be constrained to the MapReduce version. Chrysaor interacts with MapReduce runtime using the framework available commands, making it suitable to work with any MapReduce version. This is a trade off that we must take into consideration when analyzing these results.

5.5 Summary

We presented a second solution, Chrysaor, that allows to scale out MapReduce computation to a cloud-of-clouds (or multi-cloud) environment. The motivation is twofold: to tolerate arbitrary and malicious faults that may corrupt the result of MapReduce jobs at the fine granularity of a task, and to tolerate cloud outages.

Our solution involved the development of a new abstraction – the logical job – to obviate the need to modify the Hadoop source code. As such, Chrysaor requires minimal modifications to the users' applications and does not involve changes to Hadoop. As result, Chrysaor is capable of tolerating faults in MapReduce jobs at the task level in a cloud-of-clouds environment.

We compared Chrysaor with the closest alternative – Medusa – and with vanilla MapReduce to understand the impact of our new architecture. The results from experiments in Amazon EC2 have shown that our fine-grained solution is always better in the most common fault case (a fault in a map task). Moreover, despite Chrysaor requiring the double of tasks, when we compare our solution with vanilla MapReduce, we realize that in some use cases the extra-work that our solution introduces is reasonable. In addition, despite the unavoidable penalty introduced by not changing Hadoop, our novel design allows performance improvements even in the baseline case, for particular workloads.

6
Conclusions and Future Work

This chapter summarizes the main contributions of the work, concerning improving the dependability limitations of MapReduce, focusing on the several techniques that we have proposed to tolerate arbitrary faults, malicious attacks, and cloud outages. We divided the chapter into two sections. First, we present the key findings taken from our work, and then the future work that can derive from this thesis, namely to improve the performance of our solutions.

6.1 Conclusions

The popularity of cloud computing enabled the establishment of large-scale computing services from numerous resources, and the computation of massive volumes of data that traditional software had difficulty in processing in acceptable time bounds. Following this trend, Hadoop MapReduce has appeared as a popular framework that allows processing large quantities of data in parallel and distributed ways. At the scale of thousand of servers that are spread in datacenters, however, failures are frequent. Current datacenters are built with commodity servers that are prone to soft and hard errors in the hardware that will propagate to the software running atop, causing the crash of the application, or causing subtle failures. These uncorrectable errors can affect the execution of MapReduce procedures causing the framework to carry on mistakenly. Although it is crucial to tolerate crashes of tasks and data corruptions in the disk, like MapReduce does, other faults that can affect the *correctness of results* should be tolerated. MapReduce does not tolerate such arbitrary faults, affecting its dependability. Another limitation of MapReduce dependability is its design based on a single datacenter (i.e., a single cloud), which makes such framework vulnerable to an outage in the datacenter (cloud outage). In this thesis, we addressed the referred MapReduce dependability limitations — inability to deal with arbitrary faults and cloud outages — following different directions. Table 6.1 summarizes the frameworks that we have built during the Ph.D. to give MapReduce the ability to tolerate arbitrary faults, malicious faults, and cloud outages.

Application	Description	Chapter
BFT-MapReduce	MapReduce solution that masks arbitrary faults to tolerate a number of faulty task executions at the cost of one re-execution.	3
Medusa	Platform that allows MapReduce computations to scale out to multiple clouds and tolerate several types of faults.	4
Chrysaor	Platform that scales out MapReduce computations to multiple clouds and has a fine-grained replication scheme to tolerate faults at the task level.	5

Table 6.1: Summary of the contributions of the thesis

As first contribution, we designed and implemented a BFT MapReduce framework that is able to tolerate arbitrary faults. The solution has two modes of job execution: non-speculative and speculative. The goal of supporting these two modes is to improve the overall performance, by launching the minimal number of replicas as soon as possible. What differentiates them is the starting point of the reduce tasks. In non-speculative mode, $f+1$ replicas of all map tasks have to complete successfully for reduce tasks to be launched. In speculative execution, reduce tasks start after only one replica of all map tasks finishes. We evaluated our system extensively in a real testbed, Grid'5000, and concluded that our solution is indeed more efficient than the alternatives, using only twice as many resources as the original Hadoop. Also, the speculative mode considerably accelerates termination in a scenario without faults. We also concluded that the impact of faults in the makespan is low, even in a harsh fault scenario.

As second contribution, we explored the idea of replicating MapReduce jobs in multiple clouds to enhance the dependability of the framework by avoiding the unavailability of computation due to arbitrary and malicious faults, and cloud outages. We proposed

Medusa, a platform that scales out MapReduce computations to multiple clouds and tolerates these types of faults. The innovation of this work emerges from the utilization of a multi-cloud environment to parallelize computing, as well as to transparently tolerate different types of faults at a minimum cost. Our solution addresses several non-trivial challenges for this purpose. First, it is a transparent for the user. Second, it does not require any modification to the Hadoop framework. Third, it tolerates not only crash faults as the original MapReduce, but also the other faults mentioned above that can corrupt the execution, and also cloud outages. Fourth, it achieves this level of fault tolerance at the minimum replication cost and guaranteeing acceptable performance results.

As part of this work, we proposed a scheduler to distribute the replicated jobs across different clouds, based on two metrics: predicted data transmission time and data processing time. The scheduling algorithm uses several attributes related to the current status of the environment and a historic of executions to estimate the execution time of the next job. We have shown that deciding in which cloud a replicated job should be executed is crucial for gaining performance in the overall execution, by evaluating Medusa thoroughly in a real testbed, ExoGENI. The developed scheduler does so by minimizing replication and judiciously choosing the best clouds to perform the replicated jobs.

Medusa works at the granularity of MapReduce jobs, leading in some scenarios to a high cost for fault recovery. The reason is that to recover from a single fault in a task requires the whole job to be recomputed. This limitation issues the necessity of finding a new fault tolerance mechanism with less impact on performance, while keeping the goal of leaving MapReduce unchanged.

The next contribution addresses this problem. Chrysaor, the system we propose, is based on a fine-grained replication scheme to tolerate faults at the task level. Chrysaor is similar to Medusa in the sense that it replicates the execution in different clouds, but with a different system architecture and a fine-grained fault tolerance mechanism. This solution tolerates the classes of faults mentioned above at a more reasonable cost, requiring minimal modifications to the users' applications. Again, this solution does not involve changing the Hadoop source code. The creation of a fine-grained fault-tolerant mechanism was possible with the development of a new abstraction — the *logical job*. We evaluated our solution thoroughly in a real testbed, Amazon EC2, and compared it with different schemes, including Medusa and MapReduce. As a conclusion, our fine-grained solution is efficient regarding computation by recovering only faulty tasks, without incurring a noteworthy penalty for the baseline case (i.e., without faults) in most workloads.

As an overall conclusion, our solutions allow scaling out MapReduce to multiple clouds for tolerating several types of faults not covered before, and by introducing a set of core techniques that made this possible at reasonable cost. With this work, we significantly improve the dependability of MapReduce, increasing the confidence in its use even in very critical application areas.

6.2 Future Work

This thesis proposed several techniques to improve the dependability of MapReduce frameworks. Despite their advantages, there is room for improvement. We thus hope this thesis can open new avenues for research. For instance, future works can investigate different methodologies to improve the performance of MapReduce and still guarantee the same level of fault tolerance. Other may further enhance the dependability of the solution concerning security.

In this final section, we leave a few ideas. The first idea that we describe in Section 6.2.1 consists of new techniques that could be explored to improve performance in the context of Medusa and Chrysaor. The second idea, outlined in Section 6.2.2, consists in applying network programmability (e.g., SDN), to accelerate MapReduce computations. Finally, in Section 6.2.3, we consider improving security. Namely, we propose using trusted computing to guarantee data confidentiality, which is not assured in the solutions presented in this thesis.

6.2.1 Improvements on Medusa and Chrysaor

As explained in Chapter 5, the logical abstraction introduced in Chrysaor adds a performance penalty due to the fact that an extra-step is added in a job execution. Two solutions could be investigated in this respect to improving performance: the usage of hooks, and the enhancement of the MapReduce API.

Hooking is a technique used to modify the behavior of an operating system or application by intercepting function calls, messages or events passed between software components (Wynne & Hellesoy, 2012). The code that handles such events is known as a hook. In

Chrysaor (Chapter 5), we could use hooks to intercept the writes of the map tasks to disk, preventing the tasks from closing the files. This way, Chrysaor would be able to pause the execution and validate the writes to disk before resuming the execution. With this technique, we no longer need to use the identity map tasks, which would immediately improve performance.

A second solution could be to extend the MapReduce API. The idea would be to introduce new MapReduce commands that would give more control to job execution. By adding these novel commands — *pause*, *resume*, and *kill* — to be part of the MapReduce API, Chrysaor would be able to control each replicated job remotely in each cloud, adding a fine-grained control of the job execution to the system that would bring better performance to the overall execution.

Lets describe the functioning of the system using this more elegant solution in three scenarios: *(i)* without faults, *(ii)* with arbitrary faults, *(iii)* and with malicious fault or cloud outages.

1. In a no fault scenario, after all map or reduce tasks finish, the job could be *paused* by Chrysaor, to validate the output. After a successful validation, Chrysaor could resume it. Consequently, a job could be resumed after the validation of the map output, to start the reduce tasks, or after the validation of the reduce output, to just terminate the job.

2. In the case of an arbitrary fault in a task, Chrysaor could *kill* the failed task and relaunch it. An eventual correct result would replace the incorrect one, and the job could be resumed.

3. In the case of a malicious fault or cloud outages, Chrysaor could *pause* the running job to execute another replica in a new cloud. In this use case, this solution would behave similarly to Chrysaor (Section 5.3.2), but excluding the identity step. After the validation of the output, the correct jobs could be resumed from the point where they have been suspended.

A different topic of research that shares the same goal of improving performance is scheduling. In Medusa, we proposed a scheduling algorithm that replicates jobs across different clouds based on the predicted data transmission time and processing time in each cloud. Despite this solution allowing to incorporate the *heterogeneity* of the clouds into the scheduling decision, it could be interesting to explore Machine Learning techniques to achieve more accurate models and predictions for network and cloud conditions (Abadi *et al.*, 2016; Dean *et al.*, 2012). Using a machine learning solution like TensorFlow to create new scheduling algorithms with a focus on training and inference on deep neural networks, would result in an algorithm that could use innumerous parameters to select clouds more accurately.

These three solutions focus mostly on the MapReduce application. As the network and the computing system can influence the framework's performance significantly, in the next sections, we investigate the impact of these factors to enhance our solutions.

6.2.2 Exploring network programmability

Our experience with Medusa and Chrysaor led us to the conclusion that in several situations, the network is the bottleneck of the system. As such, one could explore Software-Defined Networking (Farhady *et al.*, 2015) to improve network control, enabling us to distribute traffic more precisely.

Software-defined networks (SDN) materialize the decoupling between the control and data forwarding logic of network elements (switches/routers), moving the control-plane of the network elements and on to a centralized controller (Kreutz *et al.*, 2015). Software-defined networks provide the ability to program the network at runtime in a manner such that the data flow is optimized for faster, service-aware, and more resilient application execution. The dynamic fine-grained control of the underlying datacenter network that Chrysaor or Medusa would have with SDN would allow distributing tasks taking into account the location, the performance of the node, the network throughput and latency between nodes. The common goal behind these three factors would be rearranging the network and the task distribution to avoid low network throughput during the shuffle & sort phase. One could also explore recent advances in switch programmability (Bosshart *et al.*, 2014) to improve network visibility (Shahbaz *et al.*, 2016), and thus enhance network control. For instance, using P4 (Shahbaz *et al.*, 2016) as a high-level language to program packet switching behavior in conjunction with SDN control protocols like OpenFlow, one could envisage solutions where parts of MapReduce processing could be offloaded to the network.

6.2.3 Improving security properties

Security and privacy in MapReduce computations are essential concerns when it is necessary to execute sensitive data in public or hybrid clouds. As such, one could explore SGX (Chakrabarti *et al.*, 2017) or Blockchain (Swan, 2015) technology to add confidentiality and auditability to Hadoop MapReduce.

Intel's Software Guard Extensions (SGX) is a set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to computation performed on a computer where all the privileged software is potentially malicious (Chakrabarti *et al.*, 2017; McKeen *et al.*, 2016). Allocating private regions of memory, called enclaves, to the map and reduce tasks, would force the map and reduce functions to be executed in a trusted environment (Pires *et al.*, 2017). This would add confidentiality (or privacy) to the map and reduce operations, something that it is not assured in the three solutions presented in this thesis. Specifically, the data could be provided encrypted to the enclave, decrypted inside it, and the result encrypted before leaving the enclave. This would prevent an adversary with access to the servers that process data from reading it.

Another technology that could be used to make MapReduce computations more secure is Blockchain. Blockchain is a decentralized database shared among a network of computers that are used to maintain a continuous growing list of blocks (Swan, 2015). Each peer in the network must approve a transaction before it can be recorded. This technology excludes the need of a trusted intermediary to arbiter all transactions. The information would be held securely and transparently by a digital ledger for all users on the network to see. Applying Blockchain to MapReduce is a way to make each step of a job execution auditable. For instance, after each map and reduce tasks finish, this information could be added to the database. As result, Medusa and Chrysaor could audit which tasks were executed and validate the output with the help of every party in the network. The challenge of scalability of the Blockchain would be particularly interesting in this context.

To conclude this thesis, we hope these ideas offer motivation to improve the dependability of MapReduce system further and may become part of its environment in the near future.

Bibliography

ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D.G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y. & ZHENG, X. (2016). TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283. 122

ALEXANDROV, A., BERGMANN, R., EWEN, S., FREYTAG, J.C., HUESKE, F., HEISE, A., KAO, O., LEICH, M., LESER, U., MARKL, V., NAUMANN, F., PETERS, M., RHEINLÄNDER, A., SAX, M.J., SCHELTER, S., HÖGER, M., TZOUMAS, K. & WARNEKE, D. (2014). The stratosphere platform for big data analytics. *The VLDB Journal*, **23**, 939–964. 22

ALVISI, L., MALKHI, D., PIERCE, E. & REITER, M. (2001). Fault Detection for Byzantine Quorum Systems. *IEEE Transactions on Parallel and Distributed Systems*, **12**, 996–1007. 30

AMAZON (2015). Amazon EMR Documentation. https://aws.amazon.com/documentation/emr/. 2, 14, 89, 92

AMAZON (2017). AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting. http://aws.amazon.com/ec2/. 32

AMAZON S3 (2011). Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html. 28

AMAZON WEB SERVICES (2015). Amazon S3 introduces cross-region replication. https://aws.amazon.com/pt/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/. 92

AMIR, Y., DANILOV, C., DOLEV, D., KIRSCH, J., LANE, J., NITA-ROTARU, C., OLSEN, J. & ZAGE, D. (2006). Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. In *Proceedings of the IEEE/IFIP 36th International Conference on Dependable Systems and Networks*, 105–114. 29

ANDERSON, M. (2017). Amazon Cloud Storage Failure Causes Widespread Disruption. http://www.nbcnewyork.com/news/tech/Amazons-Web-Services-Down-Causes-Massive-Outages-Online-415003823.html. 28

APACHE (2013). Fair Scheduler. https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html. 32, 37

APACHE (2013). MapReduce 0.22 Documentation – GridMix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html. 54, 80, 103

APACHE (2016). MapReduce Tutorial. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. 101

APPRIVER GUEST BLOG (2014). Malicious insiders pose biggest threat to security? http://talkincloud.com/blog/malicious-insiders-pose-biggest-threat-security. 25

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A.D., KATZ, R.H., KONWINSKI, A., LEE, G., PATTERSON, D.A., RABKIN, A. & ZAHARIA, M. (2009). Above the Clouds: A Berkeley View of Cloud Computing. Tech. rep., . v, 1

AVIZIENIS, A., LAPRIE, J.C., RANDELL, B. & LANDWEHR, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, **1**, 11–33. vii, 3, 39

BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F., PLACHOURAS, V. & TELLOLI, L. (2009). On the Feasibility of Multi-site Web Search Engines. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 425–434. 6, 68

BAHMANI, B., CHAKRABARTI, K. & XIN, D. (2011). Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 14

Baldine, I., Xin, Y., Mandal, A., Ruth, P., Heerman, C. & Chase, J. (2012). *ExoGENI: A multi-domain infrastructure-as-a-service testbed*, vol. 44 LNICST of *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, 97–113. Springer International Publishing. 8, 81

Barga, R. (2011). Daytona – Iterative MapReduce on Windows Azure. http://research.microsoft.com/en-us/projects/daytona/. 2, 14

Barroso, L.A. & Hoelzle, U. (2009). *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edn. 28

Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V. & Warneke, D. (2010). Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 119–130. 22, 23

Bessani, A., Correia, M., Quaresma, B., André, F. & Sousa, P. (2011). DepSky: Dependable and Secure Storage in a Cloud-of-clouds. In *Proceedings of the 6th Conference on Computer Systems*, EuroSys '11, 31–46. 25, 30, 42, 69

Bessani, A.N., Sousa, J. & Alchieri, E.A.P. (2014). State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, 355–362. 29

Björck, A. (1996). *Numerical Methods for Least Squares Problems*. Siam Philadelphia. 79

Blelloch, G.E. (1989). Scans as primitive parallel operations. *IEEE Transactions on Computers*, **38**, 1526–1538. 13

Blodget, H. (2017). Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data. http://www.businessinsider.com/amazon-lost-data-2011-4. 4, 5

Borkar, S. (2005). Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, **25**, 10–16. 28

BORT, J. (2016). Google apologizes for cloud outage that one person describes as a 'comedy of errors'. http://www.businessinsider.com/google-apologizes-for-cloud-outage-2016-4. 69

BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G. & WALKER, D. (2014). P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review*, **44**, 87–95. 122

BU, Y., HOWE, B., BALAZINSKA, M. & ERNST, M.D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proceeding VLDB Endowment*, **3**, 285–296. 22

BUYYA, R., RANJAN, R. & CALHEIROS, R.N. (2010). Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In C.H. Hsu, L. Yang, J. Park & S.S. Yeo, eds., *Algorithms and Architectures for Parallel Processing*, vol. 6081 of *Lecture Notes in Computer Science*, 13–31, Springer Berlin Heidelberg. 34

CASTRO, M. & LISKOV, B. (1999). Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, 173–186. 29

CASTRO, M. & LISKOV, B. (2002). Practical Byzantine Fault-Tolerance and Proactive Recovery. *ACM Transactions Computer Systems*, **20**, 398–461. viii, 7, 40, 42

CELAYA, J. & ARRONATEGUI, U. (2011). A Highly Scalable Decentralized Scheduler of Tasks with Deadlines. In *GRID*, 58–65. 35, 37

CERIN, C., COTI, C., DELORT, P. *et al.* (2013). Downtime statistics of current cloud solutions. *The International Working Group on Cloud Computing Resiliency*. vii, 5, 92

CHAKRABARTI, S., LESLIE-HURD, R., VIJ, M., MCKEEN, F., ROZAS, C., CASPI, D., ALEXANDROVICH, I. & ANATI, I. (2017). Intel Software Guard Extensions (Intel SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. In *ACM Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP'17, 7:1–7:8. 123

CHEN, A. (2010). GCreep: Google Engineer Stalked Teens, Spied on Chats. http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats. 68

CHRISTODOULOPOULOS, K., SOURLAS, V., MPAKOLAS, I. & VARVARIGOS, E. (2009). A comparison of centralized and distributed meta-scheduling architectures for computation and communication tasks in grid networks. *Computer Communications*, **32**, 1172–1184. 34

CHU, C.T., KIM, S.K., LIN, Y.A., YU, Y., BRADSKI, G., NG, A.Y. & OLUKOTUN, K. (2006). MapReduce for Machine Learning on Multicore. In *Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS)*, 281–288. 21

CLARKE, G. (2015). Microsoft Azure was most FAIL-FILLED cloud of 2014. http://www.theregister.co.uk/2015/01/16/microsoft_worst_cloud_uptime_2014. vii, 5, 92

CLAYCOMB, W.R. & NICOLL, A. (2012). Insider threats to cloud computing: Directions for new research challenges. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC)*, 387–394. 5, 24

CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M. & RICH, T. (2009a). UpRight Cluster Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. viii, 7, 40, 41, 42, 53

CLEMENT, A., WONG, E., ALVISI, L., DAHLIN, M. & MARCHETTI, M. (2009b). Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*. 29

CLEMENTE-CASTELLO, F., NICOLAE, B., RAFIQUE, M.M., MAYO, R. & FERNANDEZ, J.C. (2017). Evaluation of Data Locality Strategies for Hybrid Cloud Bursting of Iterative MapReduce. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 21

CLOUD SECURITY ALLIANCE (2013). The notorious nine: Cloud computing top threats in 2013. 24, 68

CLOUDSQUARE (2015). Cloudsquare Service Status. https://cloudharmony.com/status-1year-of-storage-and-compute-group-by-regions-and-provider. 4, 69

CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J.M., ELMELEEGY, K. & SEARS, R. (2010). MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 21–21. 20

CORREIA, M., NEVES, N.F. & VERSSIMO, P. (2006). From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal*, **49**, 82–96. 28, 43

COSTA, P., PASIN, M., BESSANI, A.N. & CORREIA, M. (2011). Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, 32–39. 60

COSTA, P., PASIN, M., BESSANI, A.N. & CORREIA, M.P. (2013). On the Performance of Byzantine Fault-Tolerant MapReduce. *IEEE Transactions on Dependable and Secure Computing*, **10**, 301–313. 8

CUI, X., ZHU, P., YANG, X., LI, K. & JI, C. (2014). Optimized big data K-means clustering using MapReduce. *The Journal of Supercomputing*, 1249–1259. 21

DAI, L., GAO, X., GUO, Y., XIAO, J. & ZHANG, Z. (2012). Bioinformatics clouds for big data manipulation. *Biology Direct*, **7**. 5

DAWN-HISCOX, T. (2017). Fujitsu's sydney data center suffers five hour outage. http://www.datacenterdynamics.com/content-tracks/security-risk/fujitsus-sydney-data-center-suffers-five-hour-outage/98826.fullarticle. 4

DEAN, J. (2009). Large-Scale Distributed Systems at Google: Current Systems and Future Directions. Keynote speech at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS). 2

DEAN, J. & GHEMAWAT, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, 10–10. v, vi, 1, 2, 13, 28

Dean, J., Corrado, G.S., Monga, R., Chen, K., Devin, M., Le, Q.V., Mao, M.Z., Ranzato, M., Senior, A., Tucker, P., Yang, K. & Ng, A.Y. (2012). Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, 1223–1231. 122

Dede, E., Fadika, Z., Govindaraju, M. & Ramakrishnan, L. (2014). Benchmarking mapreduce implementations under different application scenarios. *Future Generation Computer Systems*, **36**, 389 – 399. 21

Deshmukh, R.V. & Devadkar, K.K. (2015). Understanding DDoS Attack & its Effect in Cloud Environment. *Procedia Computer Science*, **49**, 202–210, proceedings of 4th International Conference on Advances in Computing, Communication and Control (ICAC3'15). 3

Driscoll, K., Hall, B., Sivencrona, H. & Zumsteg, P. (2003). Byzantine fault tolerance, from theory to reality. In S. Anderson, M. Felici & B. Littlewood, eds., *Computer Safety, Reliability, and Security*, vol. 2788 of *Lecture Notes in Computer Science*, 235–248, Springer Berlin Heidelberg. 29

Dubey, A.K., Jain, V. & A.P.Mittal (2015). Stock market prediction using hadoop mapreduce ecosystem. *Computing for Sustainable Global Development (INDIACom)*. 5

Dvorak, J. (2011). The Cloud: Risky, Unreliable, and Dumb. `http://www.pcmag.com/article2/0,2817,2385463,00.asp`. vii

Ekanayake, J., Pallickara, S. & Fox, G. (2008). MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, 277–284. 22

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J. & Fox, G. (2010). Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 810–818. 22

Faccio, F. (2011). *Radiation Effects and Hardening by Design in CMOS Technologies*, 69–87. Springer Netherlands. 28

FADIKA, Z. & GOVINDARAJU, M. (2010). LEMO-MR: Low overhead and elastic MapReduce implementation optimized for memory and cpu-intensive applications. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 1–8. 21

FARHADY, H., LEE, H. & NAKAO, A. (2015). Software-defined networking. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, **81**, 79–95. 122

FERRARO PETRILLO, U., ROSCIGNO, G., CATTANEO, G. & GIANCARLO, R. (2017). Fastdoop: a versatile and efficient library for the input of fasta and fastq files for mapreduce hadoop bioinformatics applications. *Bioinformatics*, **33**, 1575–1577. 5

GAO, G.R., STERLING, T., STEVENS, R.L., HERELD, M. & ZHU, W. (2007). ParalleX: A Study of a New Parallel Computation Model. In *IEEE International Parallel and Distributed Processing Symposium*. 13

GARCIA, M., BESSANI, A., GASHI, I., NEVES, N. & OBELHEIRO, R. (2011). OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*. vii, 4

GATES, A. (2011). *Programming Pig*. O'Reilly Media, Inc., 1st edn. 21

GHEMAWAT, S., GOBIOFF, H. & LEUNG, S.T. (2003). The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 29–43. vi, 2

GIFFORD, D. (1979). Weighted Voting for Replicated Data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, 150–162, ACM. 29

GONZALEZ, J.E., XIN, R.S., DAVE, A., CRANKSHAW, D., FRANKLIN, M.J. & STOICA, I. (2014). GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 599–613. 22

GROLINGER, K., HAYES, M., HIGASHINO, W.A., L'HEUREUX, A., ALLISON, D.S. & CAPRETZ, M.A.M. (2014). Challenges for mapreduce in big data. In *Proceedings of the 2014 IEEE World Congress on Services*, 182–189. 21

GROUP, I.W.I.W. (2011). P2302 - standard for intercloud interoperability and federation. http://standards.ieee.org/develop/project/2302.html. 26

GUERRAOUI, R. & RODRIGUES, L. (2006). *Introduction to Reliable Distributed Programming*. Springer-Verlag. 30

GUERRAOUI, R. & YABANDEH, M. (2010). Independent Faults in the Cloud. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS '10, 12–17. 3

GUNARATHNE, T., WU, T.L., QIU, J. & FOX, G. (2010). MapReduce in the Clouds for Science. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 565–572. 22

HADZILACOS, V. & TOUEG, S. (1994). A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts. Tech. Rep. TR 94-1425, Department of Computer Science, Cornell University, New York - USA. 28

HANLEY, M., DEAN, T., SCHROEDER, W., HOUY, M., TRZECIAK, R. & MONTELIBANO, J. (2011). An Analysis of Technical Observations in Insider Theft of Intellectual Property Cases. Tech. Rep. CMU/SEI-2011-TN-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 24

HUANG, Y., BESSIS, N., NORRINGTON, P., KUONEN, P. & HIRSBRUNNER, B. (2013). Exploring Decentralized Dynamic Scheduling for Grids and Clouds Using the Community-aware Scheduling Algorithm. *Future Generation Computer Systems*, **29**, 402–415. 34, 36, 37

IORDACHE, A., MORIN, C., PARLAVANTZAS, N., FELLER, E. & RITEAU, P. (2013). Resilin: Elastic MapReduce over Multiple Clouds. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, best paper finalist. 27

ISARD, M., BUDIU, M., YU, Y., BIRRELL, A. & FETTERLY, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 59–72. 22, 23

JAYALATH, C., STEPHEN, J.J. & EUGSTER, P. (2014). From the Cloud to the Atmosphere: Running MapReduce across Data Centers. *IEEE Transactions Computers*, **63**, 74–87. viii, 4, 6, 26, 68

JONES, S. (2017). How the wannacry cyber attack spread. https://www.ft.com/content/82b01aca-38b7-11e7-821a-6027b8a20f23. 24

KANDIAS, M., VIRVILIS, N. & GRITZALIS, D. (2013). The insider threat in cloud computing. In *Critical Information Infrastructure Security*, vol. 6983 of *LNCS*, 93–103, Springer Berlin Heidelberg. 68

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J. & GRISWOLD, W.G. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 327–353. 101

KOHLWEY, E., SUSSMAN, A., TROST, J. & MAURER, A. (2011). Leveraging the Cloud for Big Data Biometrics: Meeting the Performance Requirements of the Next Generation Biometric Systems. In *2011 IEEE World Congress on Services*, 597–601. 5

KRAFT, J. (2017). Top Cloud Outages and IT Issues of 2016. https://www.ajubeo.com/blog/top-cloud-outages-issues-2016. 4

KREUTZ, D., RAMOS, F.M.V., VERÍSSIMO, P., ROTHENBERG, C.E., AZODOLMOLKY, S. & UHLIG, S. (2015). Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, **103**, 63. 122

KRISHNAN, S., BARU, C. & CROSBY, C. (2010). Evaluation of MapReduce for Gridding LIDAR Data. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, 33–40. 21

KURZE, T., KLEMS, M., BERMBACH, D., LENK, A., TAI, S. & KUNZE, M. (2011). Cloud Federation. In *Proceedings of the 2nd International Conference on Cloud Computing, Grids, and Virtualization*. 69

LACOSTE, M., MIETTINEN, M., NEVES, N., RAMOS, F.M., VUKOLIC, M., CHARMET, F., YAICH, R., OBORZYNSKI, K., VERNEKAR, G. & SOUSA, P. (2016). User-Centric Security and Dependability in Clouds-of-Clouds. *IEEE Cloud Computing*, **3**, 64–75. 4

LAMPORT, L., SHOSTAK, R. & PEASE, M. (1982). The Byzantine Generals Problem. *ACM Transactions on Programing Languages and Systems*, **4**, 382–401. vii, 28, 29

LEE, K.H., LEE, Y.J., CHOI, H., CHUNG, Y.D. & MOON, B. (2012). Parallel data processing with mapreduce: A survey. *ACM SIGMOD Record*, **40**, 11–20. 14

LI, B., MAZUR, E., DIAO, Y., MCGREGOR, A. & SHENOY, P. (2011). A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 985–996. 20

LI, M.L., RAMACH, P., SAHOO, S.K., ADVE, S.V., ADVE, V.S. & ZHOU, Y. (2008). Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. vii, 3

LIN, H., MA, X., ARCHULETA, J., FENG, W.C., GARDNER, M. & ZHANG, Z. (2010). MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 95–106. 20

LUCKY, D. (2015). Taking advantage of multi-cloud benefits with visibility. `https://www.datapipe.com/blog/2015/05/28/taking-advantage-of-multi-cloud-benefits-with-visibility/`. vii, 4

MACK, E. (2013). Google outage reportedly caused big drop in global traffic. `https://www.cnet.com/news/google-outage-reportedly-caused-big-drop-in-global-traffic/`. 4

MALKHI, D. & REITER, M. (1997). Byzantine Quorum Systems. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, 569–578. 30, 42

MAROZZO, F., TALIA, D. & TRUNFIO, P. (2008). Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model. In *Proceedings of the 1st Workshop on Cloud Computing and its Applications*. 20

MAROZZO, F., TALIA, D. & TRUNFIO, P. (2012). P2P-MapReduce: Parallel Data Processing in Dynamic Cloud Environments. *Journal of Computer and System Sciences*, **78**, 1382–1402. 20

MATSUNAGA, A., TSUGAWA, M. & FORTES, J. (2008). CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *Proceedings of the 4th IEEE International Conference on eScience*, 222–229. 6, 68

MCKEEN, F., ALEXANDROVICH, I., ANATI, I., CASPI, D., JOHNSON, S., LESLIE-HURD, R. & ROZAS, C. (2016). Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Support for Dynamic Memory Management Inside an Enclave. In *ACM Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, 10:1–10:9. 123

MENON, R.K., BHAT, G.P. & SCHATZ, M.C. (2011). Rapid parallel genome indexing with mapreduce. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 51–58. 14

MEZA, J., WU, Q., KUMAR, S. & MUTLU, O. (2015). Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *Proceedings of the IEEE/IFIP 45th International Conference on Dependable Systems and Networks*, 415–426. 68

MIKAMI, S., OHTA, K. & TATEBE, O. (2011). Using the Gfarm File System As a POSIX Compatible Storage Platform for Hadoop MapReduce Applications. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing (GRID)*, 181–189. 26

MOHAMMED, E.A., FAR, B.H. & NAUGLER, C. (2014). Applications of the MapReduce programming framework to clinical big data analysis: current landscape and future trends. *BioData Mining*, **7**, 22. 5

MUKESH, R. (2015). Market Price Prediction Based on Neural Network using Hadoop MapReduce Technique. `http://citeweb.info/20151739232`. 5

MURRAY, D.G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A. & HAND, S. (2011). CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 113–126. 23

NAGARAJAN, V., WOLF, J.L., BALMIN, A. & HILDRUM, K. (2013). FlowFlex: Malleable Scheduling for Flows of MapReduce Jobs. In *Proceedings of the 14th ACM/IFIP/USENIX International Conference on Middleware*, 103–122. 33, 37

NARAYAN, K. (2016). 5 Devious Instances of Insider Threat in the Cloud. https://www.skyhighnetworks.com/cloud-security-blog/5-devious-instances-insider-threat-cloud/. 24

NATHUJI, R., ISCI, C., GORBATOV, E. & SCHWAN, K. (2008). Providing Platform Heterogeneity-awareness for Data Center Power Management. *Cluster Computing*, **11**, 259–271. 32

NIGHTINGALE, E.B., DOUCEUR, J.R. & ORGOVAN, V. (2011). Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 343–356. 3, 44, 68

NOGUEIRA, R., ARAÚJO, F. & BARBOSA, R. (2014). CloudBFT: Elastic Byzantine Fault Tolerance. In *Proceedings of the 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 180–189. 30

NOLTING, D. (2012). Commodity Clouds vs. Enterprise Clouds vs. Other Cloud Options: What's the Difference? http://www.bluelock.com/blog/commodity-clouds-vs-enterprise-clouds-vs-other-cloud-option-whats-the-difference/. vii, 3

NOVET, J. (2013). How NASA is using Hadoop to advance climate science. https://gigaom.com/2013/06/26/how-nasa-is-using-hadoop-to-advance-climate-science/. 14

O'DOWD, S. (2015). Top 10 big data trends in 2016 for financial services. https://mapr.com/blog/top-10-big-data-trends-2016-financial-services/. 5

OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R. & TOMKINS, A. (2008). Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 1099–1110. 21

PIRES, R., GAVRIL, D., FELBER, P., ONICA, E. & PASIN, M. (2017). A lightweight mapreduce framework for secure processing with SGX. *CoRR*, **abs/1705.05684**. 123

POWER, R. & LI, J. (2010). Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 1–14. 22

PREGUICA, N. & MARTINS, J.L. (2001). Revisiting Hierarchical Quorum Systems. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, 264–272. 29

QIANG WU, J.M., SANJEEV KUMAR & MUTLU, O. (2015). Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. vi, 2, 3, 28

RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G. & KOZYRAKIS, C. (2007). Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, 13–24. 21

RAPHAEL, J. (2014). The worst cloud outages of 2014 (so far). `http://www.infoworld.com/article/2606209/cloud-computing/162288-The-worst-cloud-outages-of-2014-so-far.html`. vii, 5

ROWSTRON, A., NARAYANAN, D., DONNELLY, A., O'SHEA, G. & DOUGLAS, A. (2012). Nobody Ever Got Fired for Using Hadoop on a Cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, 2:1–2:5. 104

SARMENTA, L.F.G. (2002). Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, **18**, 561–572. 6, 30, 40

SCHNEIDER, F.B. (1990). Implementing Fault-Tolerant Service Using the State Machine Aproach: A Tutorial. *ACM Computing Surveys*, **22**, 299–319. 6, 28, 40, 70

SCHROEDER, B. & GIBSON, G.A. (2007). Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, **78**. vi, 2

SCHROEDER, B., PINHEIRO, E. & WEBER, W.D. (2009). DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 193–204. vii, 2, 3, 44, 68

SHAHBAZ, M., CHOI, S., PFAFF, B., KIM, C., FEAMSTER, N., MCKEOWN, N. & REXFORD, J. (2016). PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 525–538. 122

SHARMA, A.V. (2016). Stock market forecasting using fuzzy logic. 5

SHARWOOD, S. (2016). Salesforce.com crash caused DATA LOSS. https://www.theregister.co.uk/2016/05/13/salesforcecom-crash-caused-data-loss/. 4

SHI, J., QIU, Y., MINHAS, U.F., JIAO, L., WANG, C., REINWALD, B. & ÖZCAN, F. (2015). Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings VLDB Endowment*, **8**, 2110–2121. 7

SMOLAKS, M. (2015). AWS suffers a five-hour outage in the US. http://www.datacenterdynamics.com/content-tracks/colo-cloud/aws-suffers-a-five-hour-outage-in-the-us/94841.fullarticle. 69

SNIJDERS, C., MATZAT, U. & REIPS, U.D. (2012). Big data: Big gaps of knowledge in the field of internet science. *International Journal of Internet Science*, **7**, 1–5. v, 1

SOTIRIADIS, S., BESSIS, N., ANJUM, A. & BUYYA, R. (2015). An Inter-Cloud Meta-Scheduling (ICMS) Simulation Framework: Architecture and Evaluation. *IEEE Transactions on Services Computing*, 1–1. 36, 37

SPECTATOR, C. (2015). Choosing Between Commodity and Enterprise Cloud. http://docplayer.net/2000690-Choosing-between-commodity-and-enterprise-cloud.html. vii, 3

STEPHEN, J.J. & EUGSTER, P. (2013). Assured Cloud-Based Data Analysis with ClusterBFT. In D.M. Eyers & K. Schwan, eds., *Proceedings of the 14th ACM/IFIP/USENIX International Conference on Middleware*, 82–102, Springer. 31

SUBRAMANI, V., KETTIMUTHU, R., SRINIVASAN, S. & SADAYAPPAN, P. (2002). Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*. 37

SWAN, M. (2015). *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 1st edn. 123

TATEBE, O., HIRAGA, K. & SODA, N. (2010). Gfarm Grid File System. *New Generation Computing*, **28**, 257–275. 26

TRUONG, H.L. & DUSTDAR, S. (2009). On analyzing and specifying concerns for data as a service. In *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*, 87–94. 5

TSIDULKO, J. (2015). The 10 Biggest Cloud Outages Of 2015 (So Far). https://www.ajubeo.com/blog/top-cloud-outages-issues-2016. 4

VENTURES, C. (2017). DDos Attack Report. http://cybersecurityventures.com/ddos-attack-report-2017/. 4

VERÍSSIMO, P., BESSANI, A.N. & PASIN, M. (2012). The TClouds architecture: Open and resilient cloud-of-clouds computing. In *Dependable Systems and Networks (DSN) Workshops*, 1–6. 25

VERMA, A., CHERKASOVA, L. & CAMPBELL, R.H. (2011). Resource Provisioning Framework for MapReduce Jobs with Performance Goals. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, 165–186. 33, 37

VERMA, A., CHERKASOVA, L. & CAMPBELL, R.H. (2014). Profiling and evaluating hardware choices for MapReduce environments: An application-aware approach. *Performance Evaluation*, **79**, 328–344. 33

VERONESE, G.S., CORREIA, M., BESSANI, A., CHUNG, L. & VERISSIMO, P. (2009). Minimal Byzantine Fault Tolerance: Algorithm and Evaluation. DI/FCUL TR 09–15, Department of Computer Science, University of Lisbon. viii, 7, 30, 40, 42

VERONESE, G.S., CORREIA, M., BESSANI, A.N. & LUNG, L.C. (2010). EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium*. 29

VERONESE, G.S., CORREIA, M., BESSANI, A.N., LUNG, L.C. & VERÍSSIMO, P. (2013). Efficient Byzantine Fault Tolerance. *IEEE Transactions on Computers*, **62**, 16–30. 29, 70

VIDELA, A. & WILLIAMS, J. (2012). *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Pubs Co Series, Manning Publications Company. 82, 100

WANG, C.M., CHEN, H.M., HSU, C.C. & LEE, J. (2010). Dynamic Resource Selection Heuristics for a Non-reserved Bidding-based Grid Environment. *Future Generation Computer Systems*, **26**, 183–197. 35, 37

WANG, L., TAO, J., RANJAN, R., MARTEN, H., STREIT, A., CHEN, J. & CHEN, D. (2013). G-Hadoop: MapReduce Across Distributed Data Centers for Data-intensive Computing. *Future Generation Computer Systems*, **29**, 739–750. viii, 4, 6, 14, 27, 68

WARNEKE, D. & KAO, O. (2009). Nephele: Efficient Parallel Data Processing in the Cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, 8:1–8:10. 23

WEISSMAN, J.B. & GRIMSHAW, A.S. (1996). A Federated Model for Scheduling in Wide-Area Systems. In *Proceedings of the 5th International Symposium on High Performance Distributed Computing (HPDC '96)*, 542–550. 35, 37

WHITE, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly, 1st edn. vi, xvii, 2, 14, 15, 18, 41, 57, 112

WILLIAMS, M. (2009). Microsoft loses Sidekick users' personal data. http://www.computerworld.com/article/2528964/networking/microsoft-loses-sidekick-users--personal-data.html. 24

WU, D., SAKR, S., ZHU, L. & WU, H. (2017). Towards Big Data Analytics across Multiple Clusters. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 26

WYNNE, M. & HELLESOY, A. (2012). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf. 120

XHAFA, F. & ABRAHAM, A. (2010). Computational Models and Heuristic Methods for Grid Scheduling Problems. *Future Generation Computer Systems*, **26**, 608–621. 34

YANG, C., YEN, C., TAN, C. & MADDEN, S. (2010). Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proceedings of the IEEE 26th International Conference on Data Engineering*. 22

YIN, J., MARTIN, J., VENKATARAMANI, A., ALVISI, L. & DAHLIN, M. (2003). Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 253–267. 55

YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P.K. & CURREY, J. (2008). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 1–14. 23

ZAHARIA, M., KONWINSKI, A., JOSEPH, A.D., KATZ, R. & STOICA, I. (2008). Improving MapReduce performance in heterogeneous environments. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation*, 29–42. 33, 37

ZHANG, X., WU, Y. & ZHAO, C. (2016). MrHeter: improving MapReduce performance in heterogeneous environments. *Cluster Computing*, **19**, 1691–1701. 33, 37

ZHENG, L., JOE-WONG, C., TAN, C.W., CHIANG, M. & WANG, X. (2015). How to Bid the Cloud. In *Proceedings of the 2015 ACM Conference on Data Communication (SIGCOMM)*, 71–84. 92