RESOURCES MANAGEMENT AND SECURITY IN DISTRIBUTED COMPUTING SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Yuquan Shan

© 2018 Yuquan Shan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2018
The dissertation of Yuquan Shan was reviewed and approved* by the following:

George Kesidis  
Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Bhuvan Urgaonkar  
Associate Professor of Computer Science and Engineering

David J. Miller  
Professor of Electrical Engineering

Chiara Lo Prete  
Assistant Professor of Energy Economics

Chitaranjan Das  
Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

This thesis investigates the relationship among distributed computing systems and jobs/tasks running in them. More specifically, how the properties of the distributed system or the decisions made by the system affect the performance of the jobs/tasks running on it, and how the jobs/tasks in turn affect the distributed system as well as the other jobs running in the same system. The first part covers the modeling and improvement of jobs/tasks performance on a system. The second part of the thesis concerns resource scheduling which can be considered as a result of the interaction between a distributed system and its jobs/tasks modeled as a game among the latter. The third part investigates the scenario where jobs/tasks negatively influence the system by improperly using the system resources (as in a denial-of-service attack) and what the system can do to address this problem.
# Table of Contents

List of Figures vii

List of Tables x

Acknowledgments xi

Chapter 1

Introduction 1

1.1 Jobs v.s. tasks 3

1.2 A typical architecture of distributed system 3

1.3 Research contributions 5

1.4 Structure of this thesis 6

Chapter 2

Running Parallel Tasks on Distributed System 8

2.1 Introduction 8

2.2 Modeling the long-running distributed system 9

2.3 Modeling individual jobs running in distributed systems 15

2.3.1 Solving straggler problem in heterogeneous system 15

2.3.2 An alternative approach: heterogeneous macrotasking (HeMT) 18

2.3.2.1 Oblivious adapted HeMT 24

2.3.2.2 Heterogeneous MacroTasking with provisioned instance types 28

2.3.2.2.1 Heterogeneous MacroTasking (HeMT) - experiments with provisioned containers 28

2.3.2.2.2 HeMT - experiments with burstable instances 29

2.3.2.2.3 HeMT - repartitioning on multiple program stages 34

2.4 Conclusions 37
Chapter 3
Fairness in Private-Cloud Scheduling

3.1 Introduction .......................................................... 39
3.2 The background on the fairness notions ....................... 40
3.3 Static resource allocation ........................................ 44
  3.3.1 Max-min fairness ............................................. 44
  3.3.2 Proportional fairness ....................................... 46
3.4 Dynamic resource allocation .................................... 47
3.5 Resource allocation in practice ................................. 48
3.6 Maintaining both fairness and efficiency in progressive filling ................................. 50
  3.6.1 Running Spark on Mesos .................................... 51
  3.6.2 Experiment configuration .................................. 52
  3.6.3 Oblivious and workload characterized allocation ........ 52
  3.6.4 Fairness criteria in workload-characterized mode .... 55
3.7 Conclusion .......................................................... 58

Chapter 4
Defense against Resource Abuse in Distributed System

4.1 Vulnerability and threat model .................................. 62
4.2 Pre-attack defense: moving target defense .................. 63
  4.2.1 Coupon collection model ................................... 64
  4.2.2 Periodically replacing all coupon types .................... 65
    4.2.2.1 Deterministic coupon acquisition with Poisson coupon- type replacements .................. 65
    4.2.2.2 Poisson coupon acquisitions with independent Pois- son coupon-type replacement process ........ 66
    4.2.2.3 Discussion: renewal coupon acquisitions, Poisson coupon-type replacements ............... 68
  4.2.3 Individual coupon type random replacement, according to a single Poisson process .............. 69
    4.2.3.1 Proof of Prop. 4 when \( Y(0) \equiv 0 \) ................ 69
    4.2.3.2 Independent coupon-type replacement events, \( Y(0) \neq 0 \) .................................. 70
4.3 Post-attack defense: shuffle and fission ...................... 71
  4.3.1 Shuffle defense .............................................. 71
    4.3.1.1 Modeling shuffle defense ............................. 72
  4.3.2 Fission defense .............................................. 75
    4.3.2.1 Case study 1: SlowLoris attack ...................... 76
    4.3.2.2 Case study 2: BlackNurse attack .................... 76
    4.3.2.3 Modeling fission defense ............................. 77
4.4 Conclusions ........................................... 80

Chapter 5
   Future Work ........................................ 82
   5.1 Task orchestration and scheduling ............... 82
   5.2 Cluster resource scheduling ....................... 82
   5.3 Defense against resource abuse ................... 83

Appendix A
   Proof of Proposition 1 ............................... 85

Appendix B
   Proof of Proposition 2 ............................... 87

Appendix C
   Properties of Poisson Processes ..................... 89

Appendix D
   Poisson Arrival See Time Average and Coupon Collection Model with Poisson Replacement 92

Bibliography ........................................... 94
List of Figures

1.1 Tasks (Mappers 1 - 3, Reducers 1 - 2) in a simple WordCount job. 3

1.2 A typical distributed system architecture: client A runs a job in
cluster mode while client B runs a job in client mode. . . . . . . . . 4

2.1 A typical fork-join system architecture. . . . . . . . . . . . . . . . . 10

2.2 Eliminating straggler by creating more tasks than servers in the
system with heterogeneous servers. . . . . . . . . . . . . . . . . . . . . 16

2.3 The job completion time increases as we further partition the dataset
into smaller parts, indicating the overhead of microtasking. . . . . . 20

2.4 HDFS architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.5 An example of HDFS replica distribution. . . . . . . . . . . . . . . 21

2.6 $p_1, p_2$ vs. $n$ when the replication factor is 2. . . . . . . . . . . . . 23

2.7 The stage completion time with different partitioning granularity,
when network I/O is the universal bottleneck. . . . . . . . . . . . . . 23

2.8 Overview of proposed modifications to application frameworks and
cluster manager. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.9 The underlying system architecture for macrotasking. . . . . . . . 25
2.10 Adaptive workload balancing with introduced interfering processes at two points in time. ........................................ 27

2.11 Workload re-balance. ............................................. 28

2.12 HeMT and even partitioning (including HomT and the default 2-way even partitioning). One executor was assigned with 0.4 CPU and the other a full CPU. ............................................. 30

2.13 Mapped 10-min workload for a t2.small instance with 4 initial CPU credits. ............................................. 31

2.14 Transformed time v.s. workload plot from Fig. 2.13. .............. 32

2.15 Superposed workload from node 1 to 3. ............................. 33

2.16 Microtasking (HomT) vs. macrotasking (HeMT) on AWS burstables (t2.medium), where HeMT yellow beam is without fudge factor and red beam is with fudge factor, when CPU is the only bottleneck. ... 34

2.17 Stage completion time when we cut the bandwidth to 50%, where the node with sufficient CPU credits becomes bottlenecked by network bandwidth. Orange area represents the overlapping area between the confidence interval of naive CPU credit-based partitioning (yellow) and that of the adjusted partitioning (red). ................. 35

2.18 Data flows in even hash shuffling and skewed hash shuffling respectively. 36

2.19 K-Means finish time. ............................................. 37

2.20 PageRank finish time. ............................................. 38

3.1 CDRF is not envy-free. Since user 2 can run on all those servers, it reaches the 1/3 share when running 3 tasks, while others can only run 1 task to reach the same share. ................................. 43

3.2 Offer-based two-level scheduling in Mesos. .......................... 49

3.3 DRF in the cluster with heterogeneous slaves. ........................ 51

viii
3.4 Experiment setup. ......................................................... 53
3.5 Workflow of oblivious and workload-characterized allocation. .... 53
3.6 Comparison between oblivious and workload-characterized modes under DRF. ......................................................... 55
3.7 Comparison between DRF and PS-DSF in workload-characterized mode. ................................................................. 56
3.8 Comparison between TSF and rPS-DSF in workload-characterized mode. ................................................................. 57
3.9 Comparison between DRF, PS-DSF and Best-fit DRF in workload-characterized mode. ..................................................... 58
3.10 Performance of Best-fit DRF and rPS-DSF in the system who will add more resources in the future. ............................... 59

4.1 An example of shuffle defense. Note that in this particular case 75% nominal clients will be liberated after one shuffle step. ..... 71
4.2 $P_{20,1}(10, R)$ and $P_{20,2}(10, R)$ versus $R$. ...................... 73
4.3 Mean fraction of affected nominal clients after a single shuffle stage. ................................................................. 75
4.4 An example of fission defense. Note that 80% of the nominal users will be free from attack after 2 fission steps. All three attackers will be quarantined after 3 fission steps. ................................. 76
4.5 Fission can be used to mitigate SlowLoris attack. .................... 77
4.6 Fission can be used to mitigate BlackNurse attack. .................... 77
4.7 Mean fraction of nominal clients free from attack after multiple binary fission stages. .................................................... 80
4.8 Mean cumulative number of containers spawned after multiple binary fission stages. .................................................... 80
List of Tables

2.1 CPU credit earning rates and caps of AWS burstable instances. . . 30

4.1 Protocols with different bandwidth amplifiers [2]. . . . . . . . . . . . 61

4.2 DDoS defense and coupon collector jargon correspondence. . . . . . 64
Acknowledgments

I thank all the committee members for their valuable feedbacks on this dissertation. This research was supported in part by NSF CNS 1526133, NSF CNS 1717571 and DARPA XD3 grant.
Chapter 1  |
Introduction

The fast growth of public cloud over the past decade has enormously changed the IT industry. Software-as-a-Service (SaaS) and big-data companies do not need to maintain hardwares (storage, computing and networking appliances) themselves, instead they find it more convenient and economic to migrate their services to the public cloud because of [3]:

1. Essentially unbounded computing capacities available on demand (but for a fee of course).

2. The elimination of an up-front commitment by software service providers.

3. The elasticity of computing scale.

The projected spending of IT companies on different types of infrastructure indicates the growth of cloud demand at the expense of traditional IT infrastructure markets [4].

On the other hand, cloud-based service is playing an increasingly significant role in Platform-as-a-Service (PaaS) providers’ income. Morgan Stanley forecasted Microsoft Cloud Services/Licensing will represent 30% of Microsoft’s total revenue by 2018 [5]. Other companies, particularly Amazon, Google, and IBM, have also invested huge amounts in their cloud services businesses, and consider it as an important source of current and future revenue.

There are roughly two ways of scaling up the computation: vertical and horizontal. Scaling vertically means simply expanding the capacity of computational, I/O and storage devices such as using CPUs with high clock rate, using solid state disk (SSD) or non-volatile memory (NVM). However, vertical scaling is limited by
cost and the upper-bound of single-machine computation (e.g., processor-memory performance gap [6]). Therefore, horizontally scaling, i.e., parallelizing the workload on the distributed system has become a more practical approach. The main challenge of distributed computation is synchronization: first all the calculating entities should have a consistent view of the system in order to work properly, which has been studied thoroughly during the past decades [7–10]; second, some tasks’ computation may depend on the results from others tasks and the calculation results of those distributed tasks must be collected and unified together at some point of computation, which can be addressed by the design of computation paradigm [11–13]. Synchronization causes computation barriers which makes the speed of the entire computation depend on the slowest tasks. How to deal with such stragglers has drawn significant attention in recent years [14–17].

Resources management is another area of interest for a large-scale shared computing and storage systems supporting many distributed heterogeneous applications [18]. Middlewares such as Mesos [1] and Kubernetes provide an automatic and scalable platform for resource management for a server cluster supporting multiple frameworks/applications. If the resources of a cluster do not suffice to satisfy the demands of its frameworks/applications, a scheduling algorithm is needed to enforce fairness or modulate demand using dynamic pricing or auction mechanisms. Many algorithms based on different definitions of fairness have been proposed in recent years [18–21]. Properties of those fairness criterion have drawn attention not only in the field of computer science [22] but also in economics [23].

Security is another important issue of cloud computing that has attracted significant research spanning privacy, integrity, and confidentiality. In our research, we focus on availability; specifically, Distributed Denial-of-Service (DDoS) defense. A typical DDoS mitigation strategy is first detecting the abnormal network traffic then filtering it out in the network, e.g., Akamai Prolexic [24]. Our research, however, focuses on cloud-side defenses which take advantage of resource and configuration elasticity in today’s cloud systems [25] and can operate even without detecting the abnormal traffic. The performance and efficiency of such DDoS defense mechanisms will be modeled and evaluated.

In this thesis, we will address some of these challenges in part through models and analysis by assessing the performance of prototypes implementations on real systems.
1.1 Jobs v.s. tasks

Jobs and tasks are two basic concepts of distributed computing. A job is usually submitted by the users to the cluster and may break down to multiple tasks. Those tasks are distributed to the computation nodes on the cluster and usually generate intermediate outputs, which will be later collected (synchronized) and unified as a result of the job. Typically, tasks are orchestrated by a job driver.

Take a typical map-reduce job such as WordCount shown in Fig. 1.1 as an example. This simple job is divided into four tasks, three map tasks and two reduce tasks. The map task takes its data partition, maps each word to a key-value pair, where the key is the word and value is 1, the ad-hoc count of this word, sorts and puts those pairs to the different buckets according to the keys. The three map tasks can run in parallel without interfering each other. The reduce task fetches the key-value pairs generated by those map tasks and sums the values for the same key, so that the outputs of the reduce tasks are the word count of all of the input data.

![Figure 1.1](image_url)

Figure 1.1. Tasks (Mappers 1 - 3, Reducers 1 – 2) in a simple WordCount job.

1.2 A typical architecture of distributed system

Running only one job on a distributed system is trivial. The job driver can spawn tasks freely on any of the available nodes. However, in practice, a distributed system, no matter public (e.g., Amazon Web Service (AWS), Google Computation Engine
(GCE)) or private (e.g., Facebook [26, 27], Yahoo! [28]), usually accommodates multiple jobs at the same time. Therefore, a system manager is needed to allocate, monitor and recycle the resources so that different jobs can run on the same cluster in harmony.

A resource manager process (also known as cluster manager) runs on one of those machines to take job registration requests and allocates resources to the registered jobs. Each machine should have at least one node manager process (also known as its agent) to report available resources to the resource manager, execute and monitor tasks.

A job runs on a distributed system in either client mode or cluster mode. The only difference between these two modes is how and where the job master is initiated. In client mode, the job master is started by the user on her own machine and registers with the resource manager requesting resources to run its tasks. In cluster mode, we need an extra step: the user, instead of directly starting the job master, sends a job submission request to the resource manager (or some third party proxy in lieu of the resource manager), the resource manager then finds an available machine to host the job master.

A typical distributed system with two jobs running in the above two modes respectively is shown in Fig. 1.2.

**Figure 1.2.** A typical distributed system architecture: client A runs a job in cluster mode while client B runs a job in client mode.
1.3 Research contributions

This dissertation discusses the results of several major research projects involving task scheduling, resource allocation and resources usage abuse in distributed systems.

The first project is motivated by the straggler problem in distributed computation, especially for the map-reduce paradigm and single-queue task scheduling. We first theoretically modeled a more general multi-queue processing system and derived an upper-bound of the number of pending jobs in the system. We then explored the impact of stragglers on jobs and studied two plausible approaches to alleviate the straggler problem. One is micro-tasking, where we divide the job evenly into a large number of tiny tasks so that the workload can be more evenly distributed on the computation nodes, where the faster nodes simply pull more tasks. The other is macro-tasking with non-uniform simply according to assessed computation capacities of the nodes and assigned to them respectively. We also worked on a more practical application by studying the task scheduling of popular distributed computation platform such as Hadoop and Spark and built a data partition mechanism for Spark which alleviates the straggler problem and reduces the overall job execution time by avoiding micro-tasking overhead.

The second project is the resource scheduling in the private cloud setting, in which fairness is the major concern. We investigated dominant resource fairness (DRF) scheduling [18] and its extensions [19,20], and explored its weaknesses in the cloud with heterogeneous servers. Especially when coarse-grained progressive filling (a dynamic approximation to the optimum) is used to approximate DRF. We then study a recently proposed, alternative fairness criterion, Per-Server Dominant-Share Fairness (PS-DSF) and its variant, residual Per-Server Dominant-Share Fairness (rPS-DSF). Our simulations demonstrate that when progressive filling is used, PS-DSF and rPS-DSF achieve higher resource utilization and lower overall execution times. We also investigated the default DRF implementation in the Mesos cluster manager and integrated PS-DSF and rPS-DSF in Mesos.

The third project focuses on the defense strategies against Distributed Denial of Service (DDoS) attacks, which is based on the threat model where attacking clients are able to exploit flaws in applications or protocols and launch non-volumetric attacks on victim systems; such malicious clients are hard to detect. We studied
defense strategies for two phases of DDoS attacks: pre-attack phase, where the attacker identifies the vulnerable targets, and post-attack phase, where the attacker commands bots to attack them. For the pre-attack phase, we studied the efficiency of moving target defense (MTD), where the defender undermines the attacker’s process of identifying targets by intermittently changing their identities. For post-attack phase, we discussed two strategies to migrate DDoS attacks - shuffling and fissioning. In shuffling, all servers under attack are randomly reassigned clients, those who are assigned to servers without any attacking clients are liberated and will not participate in the following shuffling steps. In fissioning, several containers are spun-up for each container housing a server under attack; the clients on the attacked server are equally divided among those containers; if a container contains only benign clients, then all clients on that container would be free from attack and fissioning would stop on that container. An approach based on Stirling numbers of the second kind is used to analyze those strategies, but the numerical complexity of the huge numbers involved make it unsuitable to analyze a large system with more than hundreds of servers. So we also used a binomial model to approximate the client assignment process, which works well for large systems.

1.4 Structure of this thesis

The rest of this thesis is organized as three sections.

In Sec. 2, we focus on the quality-of-service (QoS) of distributed systems. Network calculus is used to analyze the service of a long-running generic parallel-computing stages which constantly receives and serves/executes tasks and a QoS-related bound is derived. We then consider a specific jobs and study different approaches to shorten execution time by well balancing the workload in a heterogeneous computing environment.

In Sec. 3, we discuss fair and efficient resource allocation for different jobs in a distributed heterogeneous system. Multiple fairness criteria are considered, corresponding optimization problems are set up for the batch scheduling. Then, we address the practical challenges of dynamically enforcing those fairness criteria achieving higher resource utilization given information regarding both workload demands and resource supplies of our system.

In Sec. 4, we explore the countermeasures against DDoS attacks on such
systems. Our defense strategies for pre-attack (moving target) and post-attack (fission, shuffling) phases are described and analyzed.

Finally, we summarize and discuss future work in Sec. 5.
Chapter 2  
Running Parallel Tasks on Distributed System

2.1 Introduction

To scale up computation, storage, and communication capacity of a parallel processing system, one can take the vertical approach, e.g., increasing the speed of hardware including CPU, memory, disk/network I/O devices. However, this approach is limited considering multiple factors such as performance plateau of CPUs, upgrading costs and possible increased threat of hardware failure (e.g., data loss in hard disk drive failure).

Given the above difficulties in expanding vertically, scaling up horizontally, i.e., loosely coupling machines interconnected in a network, is more appealing. Performing computation and storage in the distributed fashion has the following advantages [29]:

- Resource sharing: for example, a distributed file system allows a file to be shared by two clients running different operating systems.

- Computation speedup: jobs can be load-balanced to multiple machines instead of overloading a particular machine; moreover, with the help of distributed computing libraries and frameworks, a single job can be parallelized and run in a distributed fashion so that each machine is assigned a small amount of work and able to finish it quickly.

- Reliability: if a system contains multiple general-purpose machines, then it
can still run properly even some of those machines fail; if a distributed storage system such as network file system or distributed database, with enough redundancy, failures of some of the storage nodes will not result in the data loss.

- Scalability: little overhead is needed to incorporate new machines in a distributed system.

Over the past decade, many services have been designed to run in distributed fashion. For example, Apache Hadoop and Spark were developed to break a large data processing job into multiple small tasks so that the computation can be scaled up by running those tasks on a potentially large number of computing nodes in parallel.

Performance studies for distributed system take different approaches. Some studies [30, 31] consider the distributed system as a large queuing system that consistently receives and process job requests. Given a certain job arrival rate and system service rate, they are interested in how long an arriving job is queued and its execution, its execution/service time and the size of system’s backlog. The properties of queuing system with different arrival and service patterns such as waiting/sojourn time [32], queue length [30, 33] etc., are broadly studied in early ages. Various scheduling algorithms [34] were proposed with different objectives. Other studies, e.g., [14, 15, 35], investigate how to efficiently process given certain resources acquired from the distributed system. The following sections of this chapter present studies that cover both of the above directions.

### 2.2 Modeling the long-running distributed system

A branch of queuing theory known as network calculus focuses on a queuing system, where arriving tasks are characterized by an “arrival curve” and are served in according to a “service curve”. In this section, we use network calculus as a tool to analyze the queue/backlog length of a fork-join parallel processing system containing multiple servers and a service barrier at the join.

Consider a parallel processing system showing in Fig. 2.1, modeled as a bank of $K$ parallel queue, with queue-k provisioned with service/processing capacity $s_k$. Let $A$ be the overall system arrival curve representing the cumulative arrivals in
the time interval $[0, t)$.

The input is load-balanced among queues so that the $k$th queue has arrivals $a_k$ and departures $d_k$ in such a way that $\forall t \geq 0$,

$$A(t) = \sum_k a_k(t).$$

We assume $A(t) = 0$ for $t \leq 0$. The departures from the $k$th queue in $[0, t)$ are denoted by $d_k(t)$, with $d_k(t) \leq a_k(t)$. Define the virtual delay processes for hypothetical departures from queue $k$ at time $t \geq 0$ as

$$\delta_k(t) = t - a_k^{-1}(d_k(t))$$

where we define inverses $a_k^{-1}$ of a strictly increasing function $a_k$ as continuous from the left so that $a_k(a_k^{-1}(v)) \equiv a_k^{-1}(a_k(v)) \equiv v$. Taking $a_k$ as strictly increasing is a technical assumption, which can be achieved by incrementing $a_k$ at a negligibly small nominal rate, when there are no arrivals.

![Diagram](image.png)

**Figure 2.1.** A typical fork-join system architecture.

In the following definition of the cumulative departures, $D$, the output is determined by the most lagging (straggling) queue/processor, i.e., for all $t \geq 0$,

$$D(t) = A(t - \max_k \{\delta_k(t)\}) = A(\min_k \{a_k^{-1}(d_k(t))\})$$

Note that in the case of continuous, fluid arrivals (e.g., piecewise linear $A$), this definition of departures $D$ corresponds to periods of continual, possibly perpetual, barriers (synchronization times). In the case of discrete arrivals (piecewise constant
A with jump discontinuities at arrival instances), the barriers are discrete. The total backlog in the system at time \( t \) is given by \( A(t) - D(t) \).

Define the convolution \((\otimes)\) of two non-decreasing functions \( f \) and \( g \), with \( f(t) = g(t) = 0 \) for \( t \leq 0 \), by

\[
(f \otimes g)(t) = \inf_{0 \leq \tau \leq t} \{ f(\tau) + g(t - \tau) \}.
\]

We define a delay function \( \Delta_v \) for any \( v \geq 0 \) as

\[
\Delta_v(t) = \begin{cases} 
0 & \text{if } t \leq v, \\
+\infty & \text{if } t > v.
\end{cases}
\]

For any function \( f \), constant \( v \geq 0 \), and time \( t \),

\[
f(t - v) = (f \otimes \Delta_v)(t).
\]

Given an arbitrary queue with cumulative arrival and departure functions given by \( a(t) \) and \( c(t) \), respectively. The queue has a lower service curve \( s_{\min} \) if for all times \( t \) and arrivals \( a \),

\[
c(t) \geq (s_{\min} \otimes a)(t).
\]

A lower service curve is a non-decreasing function that describes a service guarantee of the queue. We assume that the arrivals to queue \( k \) are bounded by a bursiness curve (traffic envelope) \( b_{\in,k} \) in the sense that for all \( t \geq 0 \),

\[
a_k(t) \leq (a_k \otimes b_{\in,k})(t).
\]

In other words, \( b_{\in,k} \) is an upper bound on the arrivals to queue \( k \) in any time interval of length \( x \).

For a queue with lower service curve \( s_{\min} \) and arrivals with burstiness curve \( b_{\in,k} \), an upper bound on delay is

\[
d_{\max,k} = \min \{ z \geq 0 : \forall x \geq 0, s_{\min,k}(x) \geq (b_{\in,k} \otimes \Delta_z)(x) \}. \quad (2.1)
\]

Here, \( d_{\max,k} \) is the largest horizontal difference between \( b_{\in,k} \) and \( s_{\min,k} \) \([36]\).
Claim 1. A lower service curve of a fork-join system is given by

\[ s_{\min}(t) = \Delta_{\max_k\{d_{\max,k}\}}(t). \]

**Proof:** Note that this claim is equivalent to \( D(t) \geq A(t - \max_k\{d_{\max,k}\}), \forall t \geq 0. \) By Eq. 2.1, \( \forall t \geq v \geq 0 \) and \( \forall k, \)

\[ s_{\min,k}(t - v) \geq b_{\in,k}(t - v - d_{\max,k}) \geq a_k(t - d_{\max,k}) - a_k(v) \]

Thus, \( \forall t \geq v \geq 0 \) and \( \forall k, \)

\[ a_k(v) + s_{\min,k}(t - v) \geq a_k(t - d_{\max,k}) \]

\[ \Rightarrow (a_k \otimes s_{\min,k})(t) \geq a_k(t - d_{\max,k}) \]

\[ \Rightarrow a_k^{-1}((a_k \otimes s_{\min,k})(t)) \geq t - d_{\max,k} \]

where we have used the fact that, \( \forall k, a_k \) are nondecreasing. Thus,

\[ D(t) = A\left(\min_k\{a_k^{-1}(d_k(t))\}\right) \]

\[ \geq A\left(\min_k\{a_k^{-1}((a_k \otimes s_{\min,k})(t))\}\right) \]

\[ \geq A\left(\min_k\{t - d_{\max,k}\}\right) \]

\[ = A\left(t - \max_k\{d_{\max,k}\}\right) \]

where we have used the fact that \( A \) is nondecreasing. \( \square \)

**Remark:** The claim simply states that the maximum delay of the whole system is the maximum delay among the queues.

To simplify matters, we assume the workload process \( A \) satisfies a stationary bound, in the sense of strong (or generalized) stochastically bounded burstiness (gSBB) [37].

Claim 2. In the stationary regime at time \( t \geq 0, \) if

(A1) The service at queue \( k \) has a lower service curve \( s_k \) satisfying

\[ \forall v \geq 0, s_k(v) \gg v\mu_k \text{ with } \mu_k > 0; \]
For queue $k$, there exists a small $\epsilon_k > 0$ such that $\forall v \leq t$,

$$\left| a_k(t) - a_k(v) - \frac{\mu_k}{M}(A(t) - A(v)) \right| \leq \epsilon_k,$$

where $M := \sum_k \mu_k$.

The total arrivals have strong stochastically bounded burstiness,

$$P\left( \max_{v \leq t} \{A(t) - A(v) - M(t - v)\} \geq x \right) \leq \Phi(x),$$

where $\Phi$ decreases in $x > 0$;

then $\forall x > 2M \max_k \{\epsilon_k/\mu_k\}$,

$$P(A(t) - D(t) \geq x) \leq \Phi(x - 2M \max_k \{\epsilon_k/\mu_k\}).$$

Remark: By (A2), the mapper divides arriving work roughly proportional to the minimum allocated service resources $\mu_k$ to queue $k$, and does so almost surely. The claim expresses a probabilistic bound on the backlog of the system, which holds for all times $t$.

Proof:

$$P(A(t) - D(t) \geq x)$$

$$= P\left( A(t) - A(\min_k \{a_k^{-1}(d_k(t))\}) \geq x \right)$$

$$= P\left( \min_k \{a_k^{-1}(d_k(t))\} \leq A^{-1}(A(t) - x) := t - z \right)$$

$$= P\left( \min_k d_k(t) - a_k(t - z) \leq 0 \right)$$

$$= P\left( \max_k a_k(t) - d_k(t) - (a_k(t) - a_k(t - z)) \geq 0 \right)$$

$$\leq P\left( \max_k \max_{v \leq t} \{a_k(t) - a_k(v) - (t - v)\mu_k\} - x_k \geq 0 \right),$$

where $x_k := a_k(t) - a_k(t - z)$ and we have used the fact that $A$ and the $a_k$ are nondecreasing (cumulative arrivals) and the inequality is by (A1). Also, we have defined non-negative random variables $z$ and $x_k$ such that $\sum_k x_k = x = \ldots$
A(t) - A(t - z). So by using (A2) (twice) then (A3), we get

\[
P(A(t) - D(t) \geq x) \\
\leq P\left(\max_k \max_{v \leq t} \left\{ \frac{\mu_k}{M} (A(t) - A(v)) + \epsilon_k - (t - v)\mu_k \right\} - \frac{\mu_k}{M} x + \epsilon_k \geq 0 \right) \\
= P\left(\max_k \max_{v \leq t} \left\{ A(t) - A(v) - (t - v)M \right\} - x + 2M \epsilon_k \geq 0 \right) \\
= P\left(\max_{v \leq t} \left\{ A(t) - A(v) - (t - v)M \right\} \geq x - 2M \max_k \left\{ \epsilon_k / \mu_k \right\} \right) \\
\leq \Phi \left( x - 2M \max_k \left\{ \epsilon_k / \mu_k \right\} \right)
\]

□

The following corollary works with a more dynamic load matching.

Claim 3. If (A1), (A3) and A2' For each queue \( k \), there exists \( 0 < \epsilon_k, \delta_k \ll 1 \) such that \( \forall v \leq t \)

\[
P\left( a_k(t) - a_k(v) - \frac{\mu_k}{M} (A(t) - A(v)) > \epsilon_k \right) < \delta_k,
\]

then \( \forall x > 2M \max_k \left\{ \epsilon_k / \mu_k \right\} \),

\[
P(A(t) - D(t) \geq x) \leq \Phi(x - 2M \max_k \left\{ \epsilon_k / \mu_k \right\}) + 2\delta_{\text{arg max}_k \epsilon_k / \mu_k}.
\]

Proof: The corollary is proved by applying the following simple results:

\[
P(X > \tilde{X}) = P(X > \tilde{X} | |X - Y| \leq \epsilon)P(|X - Y| \leq \epsilon) + P(X > \tilde{X} | |X - Y| > \epsilon)P(|X - Y| > \epsilon) \\
\leq P(Y + \epsilon > \tilde{X}) + \delta.
\]

Similarly, if also \( P(|\tilde{X} - \tilde{Y}| \geq \epsilon) < \delta \) then

\[
P(X > \tilde{X}) \leq P(Y + \epsilon > \tilde{Y} - \epsilon) + 2\delta \\
= P(Y > \tilde{Y} - 2\epsilon) + 2\epsilon. \tag{2.2}
\]

Then we can prove this claim by applying the above results to where (A2) is
used in the proof of Claim 2, by considering

\[ X = a_k(t) - a_k(v) \]
\[ Y = \frac{\mu_k}{M} (A(t) - A(v)) \]
\[ \tilde{X} = a_k(t) - a_k(t - z) = x_k \]
\[ \tilde{Y} = \frac{\mu_k}{M} (A(t) - A(t - z)) = \frac{\mu_k}{M} x. \]

□

2.3 Modeling individual jobs running in distributed systems

In fact, the task scheduler of many popular processing platforms, such as Hadoop [38] and Spark [13] will generally not let tasks arrive the system in random fashion. Instead, input data is partitioned, with each data segment associated with a task and all the tasks are made ready at the beginning of a computation stage. Also, computing resources are utilized in a non-work-conserving fashion, i.e., the resources are reserved by a job until all of its tasks complete, even though sometimes the executors are idling. The scheduler will get notified when some server becomes available and assign tasks to it. In recent years, significant attention has been given to the “straggler” problem [14–17,35,39], i.e., when the workload is not well balanced, an entire job is slowed by one or more of its component tasks (the stragglers).

2.3.1 Solving straggler problem in heterogeneous system

This section introduces the straggler problem in heterogeneous systems, using the Spark platform as an example. The task scheduling system of Spark can be considered as a single queuing system. When Spark starts a processing stage for a job, it sets all the associated tasks as pending in a centralized queue, and when it is notified of available resources from some servers, it assigns a certain number of tasks to run on those servers thus occupying those resources.

Spark combines homogeneous microtasking [14], black-listing and speculation execution to heuristically alleviate the straggler problem.
Homogeneous microtasking, i.e., evenly dividing the entire job (its input dataset to be processed) into multiple independent small tasks, is the foundation of parallel computation. Without considering its overhead [14, 40], including I/O inefficiency [41], scheduling, serializing/deserializing, and shipping tasks, microtasking can be considered as a straightforward remedy of straggler problems that does not require knowledge of workload resource needs to meet its performance requirements nor of the speed of the executors/servers assigned to its tasks. For example, if there are two executors, one twice as fast as the other. To process workload $W$, the slow executor takes $\frac{W}{s}$ while the fast one takes $\frac{W}{2s}$. If we evenly divide this workload to those two executors. Then the workload finish time is $\frac{W}{2s}$ - the fast one has to idle for $\frac{W}{2s} - \frac{W}{4s} = \frac{W}{4s}$. However, if we evenly divide the workload into three pieces, then faster executor will pull another task once it finish the first one. As a result, the finish time would be $\frac{W}{3s}$, $\frac{W}{6s}$ faster than the previous case. The example can be illustrated in Fig. 2.2.

**Figure 2.2.** Eliminating straggler by creating more tasks than servers in the system with heterogeneous servers.

Using homogeneous microtasking to eliminate straggler can be more generally described as follows. Let $\{s_1, s_2, \ldots, s_n\}$ be the processing speed of $n$ servers in the system. We want

\[
\text{find } x_i : x_i \in \mathbb{Z}_+ \\
\text{s.t.,}
\frac{x_1}{s_1} = \frac{x_2}{s_2} = \ldots = \frac{x_n}{s_n}.
\]
where $\sum_{i=1}^{n} x_i$ is the total number of tasks. Note that the if $s_i$ are integers, the problem is reduced to finding common divisor (cd) of $\{s_1, s_2, \ldots, s_n\}$ and $x_i = \frac{s_i}{cd}$. Also note that if $\sum_{i=1}^{n} x_i = \infty$, meaning the job is divided into infinitesimal pieces, the straggler problem can be solved for any $\{s_1, s_2, \ldots, s_n\} \in \mathbb{R}_+^n$.

If the nodes\(^1\) computing capacities are constant during the entire job, We can also derive a upperbound on resource idling time as follows.

**Claim 4.** For pull-based task assignment (i.e., a worker pulls one more task if it is done with its assigned task and there are more tasks remaining), suppose all the tasks in a stage are pending at some initial time 0, the workload is evenly partitioned into tasks and the speed of the nodes is constant. Then the resource idling time (latest node finish time minus earliest node finish time) is upper-bounded by the single task duration of the slowest node.

**Proof.** Let $T_i$ be the task duration on node $i$, $t_i$ be the finish time of node $i$ (the time when node $i$ finishes its last task). Assume node 0 is the first node to be idle. Note that $\forall \epsilon > 0$, at $t_0 + \epsilon$, there is no pending task\(^2\) and all the remaining tasks are running on the other nodes. Let $t_0 - \Delta t_i, i \neq 0$ be the starting time of the running task on node $i$, so the finish time of the last task would be $\max_i \{t_0 - \Delta t_i + T_i\} = t_0 + \max_i \{T_i - \Delta t_i\}$. Note that $\Delta t_i \geq 0, \forall i, i \neq 0$. Therefore, $t_0 + \max_i \{T_i - \Delta t_i\} \leq t_0 + \max_i \{T_i\}$. \(\square\)

With the **blacklist** mechanism, Spark assumes that if too many tasks have failed or a given task has failed multiple retries on some executors or nodes (servers), then those executors/nodes are not suitable to run any task or run a certain task under certain circumstances. Consider the case where some tasks need to be allocated a small set of executors due to locality preferences, however some executors are running on a node with malfunctioning disk and its tasks will fail due to disk read error. So those tasks are very likely to fail many times before they are successfully executed and will be deemed stragglers. Therefore, a blacklist would be established to prevent repeatedly sending tasks to those flawed computing nodes.

To prevent assigning a task or any task to certain executors or nodes during a stage or entire application life span, Spark maintains separate blacklists for

---

\(^1\)Note executors are processes that run tasks. And nodes/servers are machines that host those executors.

\(^2\)Otherwise, node 0 would pull another pending task to run.
- (executor, task) pair
- (node, task) pair
- (executor, stage) pair
- (node, stage) pair
- executor
- node

Spark would put an executor/node to the corresponding blacklist when the number of failure exceeds a user-specified number.

**Speculative execution** [15, 35, 42] is another ad-hoc approach to reduce the impact of stragglers. When speculation is enabled, if the task monitor detects that some tasks run significant slower than its peers, then Spark will spawn another identical task that would run on a (hopefully faster) executor. When one of those identical tasks is completed, the remaining tasks will be killed. Concretely, the median completion time is used as the reference for speculation - if a task is X% (50% by default) slower than the median, it will be thus “speculated”. Note that if speculation is activated at the beginning of a stage, the several quickly-executed tasks would cause extremely low completion time median, leading to a high false alarm rate. Hence, the speculation quantile with default value 0.75 is used by Spark, to make sure that speculation will not be activated until at least 75% of tasks in a stage have been completed.

### 2.3.2 An alternative approach: heterogeneous macrotasking (HeMT)

Although homogeneous microtasking (HomT) has been shown to effectively balance the workload so as to alleviate the straggler problem in many cases [14], there are several weaknesses of microtasking in practice that prevents microtasking from being a pure-win solution in big data processing.

First, to let a task run on a remote executor, the task scheduler needs to first use its scheduler algorithm to determine the next task to run, then serialize and ship the task to the remote; on the remote executor side, this task will be deserialized; also, the task status might be transmitted between job driver and executor. Since
those steps are generally done on single-task basis, the time spent on those grows as the number of tasks increases. This fact prevents the usage of microtasking in Apache Hadoop, where every task needs a brand new host Java virtual machine, which make a task take seconds to spawn [14].

Second, as indicated in [41], too many tiny tasks would lead to a significant performance degradation due to low I/O efficiency in map-reduce paradigm. When we break a map stage into a large number of tasks, the output of each map task would be extremely small. As a result, when those tiny intermediate outputs are fetched in the next stage, we would have large amount of random disk accessing, each obtaining only 10s or 100s KB. Considering the fact that the size of modern disk driver cache is usually 8 or 16 MB [43], too many KB-level random disk accessing would make too much time wasted on disk location seeking and disk arm rotating.

To see the effect caused by the above microtasking overhead, we did word-count on the same 700MB+ text file with different numbers of partitions on a AWS cluster with three one-core executors (meaning each can run one task at the same time). Fig. 2.3 shows the degradation of job performance as we further split the job into more and more pieces.

Another problem of microtasking we found in our experiments is related to the distributed file system, e.g., HDFS (Hadoop Distributed File System): when the job is network I/O bottlenecked, if multiple tasks simultaneously access the same HDFS data block, then they are more likely to read on the same datanode, which may lead to inefficient overall network bandwidth usage in the HDFS cluster. On the other hand, the Spark by default sequentially schedules tasks so that consecutive tasks are more likely to access the same block as large tasks are increasingly divided into smaller ones.

HDFS is designed to store very large files in a distributed fashion. It follows a master/slave architecture. Namenode, the master, manages the file system metadata and coordinates the data placement. Datanodes serve as slaves that perform the actual data reads and writes. To operate on HDFS, a client first contacts the namenode, which will redirect the client to the correct datanodes for write and read. The HDFS architecture is illustrated in Fig. 2.4.

Each file is usually split into a sequence of blocks, and each block is replicated for fault tolerance. HDFS does not allow a single datanode to store multiple replicas
Figure 2.3. The job completion time increases as we further partition the dataset into smaller parts, indicating the overhead of microtasking.

Figure 2.4. HDFS architecture.

for the same block [44]. To simplify our analysis, we make two assumptions: first, rack-awareness [45] in block placement is turned off\(^3\); a simple placement policy is used such that namenode would randomly choose one among the equally-distant datanodes to place data block as well as its replicas. So when a remote user

\(^3\)In fact, Hadoop rack-awareness with less-randomness, intensifies the uplink competition since data are spread less broadly.
uploads a data block, HDFS randomly chooses datanodes, each storing exactly one replica of that block. Upon a read request, HDFS tries to choose the closest replica (on the same datanode or on the same rack) to the reading client. If there are multiple candidate replicas, then the replica choice is random. One typical replica distribution on HDFS is shown in Fig. 2.5.

![File metadata stored on namenode](//File1, 2, {2, 3})
(//File2, 3, {1, 4, 5, 6})

---

<table>
<thead>
<tr>
<th>Datanodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 5</td>
</tr>
<tr>
<td>2 4 6</td>
</tr>
<tr>
<td>1 6</td>
</tr>
<tr>
<td>4 5</td>
</tr>
<tr>
<td>2 1 6</td>
</tr>
<tr>
<td>1 5</td>
</tr>
</tbody>
</table>

**Figure 2.5.** An example of HDFS replica distribution.

In a HDFS cluster, suppose the number of datanodes is $n$ and the replica factor is $r$, and $n > r$ (the usual case). If two tasks access the same HDFS block, then the probability that they will read on the same datanode, competing for its uplink bandwidth\(^4\) is

$$p_1 = \frac{1}{r}. \quad (2.3)$$

If two tasks access different HDFS blocks, the probability that they will read on the same datanode is

$$p_2 = \sum_{v=\max(2r-n, 0)}^{r} P(v) \frac{v}{r^2}, \quad (2.4)$$

where $P(v)$ is the probability that there are $v$ datanodes that store replicas of both blocks. Assume HDFS randomly choose $r$ datanodes to store replicas when the

\(^4\)They may also compete for the disk bandwidth on that datanode. But considering disk bandwidth is usually larger than the network bandwidth, disk bandwidth is not the concern herein.
data is uploaded (appears so, need to verify if it is true in HDFS implementation),

\[ P(v) = \frac{{r \choose v} \frac{n-r}{r-v} \frac{v}{n \choose r}}{r^2} \]  \hspace{1cm} (2.5)

Considering Eq. 2.5 and 2.3, we can

**Claim 5.**

\[ p_1 \geq p_2, \]

with equality when \( r = n \).

**Proof.** With Eq. 2.5,

\[ p_1 \geq p_2 \iff \sum_{v=\max(2r-n,0)}^{r} \frac{{r \choose v} \frac{n-r}{r-v} \frac{v}{n \choose r}}{r^2} \leq \frac{1}{r} \]

\[ \iff \sum_{v=\max(2r-n,0)}^{r} \frac{{r \choose v} \frac{n-r}{r-v} \frac{v}{n \choose r}}{r} \leq 1 \]  \hspace{1cm} (2.6)

The inequality in (2.6) obviously holds by noting that within the summation in

(2.6), \( \frac{v}{r} \leq 1 \), and

\[ \sum_{v=\max(2r-n,0)}^{r} \frac{{r \choose v} \frac{n-r}{r-v} \frac{v}{n \choose r}}{r} = 1. \]

The plot of \( p_1 \) and \( p_2 \) versus \( n \) for \( r = 2 \), shown in Fig. 2.6, numerically supports the above conclusion that \( p_1 \geq p_2 \). That is, they indicate that two tasks reading on the same block are more likely to compete for the uplink bandwidth on the same datanode.

In another experiment to support the above analysis, a small HDFS cluster, with \( n = 4 \) datanodes and replication factor \( r = 2 \), had limited datanode uplink-bandwidth of 64 Mbps. Thus, the Spark tasks are always bottlenecked by network I/O. So, they execute at the same rate, even though the workers have different CPU resources. The datanodes are equally distant to the Spark workers so the selection of datanode for one block is random. The results are shown in Fig. 2.7. As can be observed, the stage completion time increases with the number of tasks/partitions.
Figure 2.6. $p_1, p_2$ vs. $n$ when the replication factor is 2.

Figure 2.7. The stage completion time with different partitioning granularity, when network I/O is the universal bottleneck.

On the other side, studies [46,47] have showed that the workload with larger task sizes, if properly programmed and orchestrated, would outperform the microtasking.

To avoid HomT overhead, the number of tasks can be set equal to the number of available “computation slots” (maximum parallel tasks). However, in case of
heterogeneous executors, synchronization delay may ensue if such “macrotasks” are equally sized. This motivates heterogeneous macrotasking (HeMT).

HeMT will require a reasonably accurate estimation of workload (reflected by task execution time) which can be easily obtained for many modern jobs due to their repetitive nature; e.g., many production workloads [27, 48, 49], and many machine-learning related jobs such EM and K-Means [50] that consist of multiple iterations of the same computational complexity. Much recent work on task scheduling, e.g., [51], is based on such an assumption.

We implemented this HeMT partitioning algorithm on the distributed-application framework Apache Spark and compared it with Spark’s default partitioning scheme in the following, as well as the aforementioned HomT. Spark’s default partitioning does not consider any resource heterogeneity of the cluster - it divides the input data regardless of the speed of computing nodes - and Spark tends to evenly divide on-memory data into as many partitions as the number of computing slots (usually processing cores). For files located on disk, e.g. HDFS files, baseline Spark, like Hadoop, assigns one file block to a task. Spark naturally supports HomT: users can specify a desired number of partitions and Spark would evenly divide data according to this number.

The aim of experimental studies described in the following is to illustrate the benefits and challenges of HeMT. We implemented HeMT in Spark, using information from middleware or directly from monitoring services (e.g., AWS CloudWatch). For scalability, the application frameworks perform most elements of (workload specific) HeMT learning, while the middleware scheduler may only perform more sophisticated workload scheduling (consolidation)\(^5\). The information exchange in our Spark-Mesos prototype are summarized in Figure 2.8.

And the underlying system architecture of our approach is shown in Fig. 2.9.

### 2.3.2.1 Oblivious adapted HeMT

In some environments, e.g., those without resource isolation leading to significant interprocess interference, determining the true workload processing power of available computational nodes may be challenging. So, a simple “oblivious” approach is needed to allow application frameworks and cluster managers to dynamically

\(^5\)Analogies can be made with exokernels or TCP congestion control in the Internet.
estimate the processing speed of available computational nodes according to the previous workloads of the same job, so that the future tasking can be well balanced.

A Spark-Mesos prototype was implemented to enable such an oblivious ad-hoc adaptive HeMT. The Mesos cluster manager obtains and passes on to the Spark application framework estimated executor processing speed through additional fields in their RPC messaging. Based on this information and the associated task sizes, Spark estimates the execution speed of different available executors and thereby determines how to partition future work into well-balanced tasks.

Consider a sequence of datasets of sizes \( \{D_k\} \) that need to be processed in the same way, i.e., the same job applied to each dataset. The \( k^{th} \) dataset \( D_k \) is divided (by the application framework) into a number of tasks, one for each executor \( i \in L_k \).

---

**Figure 2.8.** Overview of proposed modifications to application frameworks and cluster manager.

**Figure 2.9.** The underlying system architecture for macrotasking.
assigned to process the $k^{th}$ dataset $D_k$ (by the cluster manager). These tasks are created by dividing the dataset $D_k$.

For each executor $i \in L_k$, let $v_i$ be the most recent estimate of its “speed” for the job under consideration. Let $L_k^0 \subset L_k$ be the set of executors that have not before been assigned to this job. For all $i \in L_k^0$, let $v_i = \bar{v}$ where $\bar{v}$ is the average $v_j$ for $j \in L_k \setminus L_k^0$ (example other choices could be the minimum or maximum rather than the average or the average speed over all executors that have been applied to this job in the past). Let

$$V_k = \sum_{i \in L_k} v_i = |L_k| \bar{v},$$

where $|L_k|$ is the number of executors assigned to the $k^{th}$ job. Executor $i \in L_k$ is assigned a dataset of size $d_i = D_k v_i / V_k$. That is, the faster executor (larger $v_i$) is assigned to work on a larger dataset (larger $d_i$).

Let $t_i$ be the execution time of executor $i \in L_k$ on the assigned task of size $d_i$ of the $k^{th}$ job. For all executors $i \in L_k$, their speed can be updated according to a simple first-order autoregressive estimator

$$v_i \leftarrow (1 - \alpha) \frac{d_i}{t_i} + \alpha v_i$$

where forgetting factor $\alpha$ satisfies $0 < \alpha < 1$.

For the initial ($k = 1$) job, $D_1$ is evenly divided among the executors $i \in L_1$ and subsequently $v_i = d_i / t_i$.

The straightforward tradeoff in the choice of $\alpha$ is that smaller $\alpha$ means that the speed estimate is more responsive to the lastest speed datapoint $d_i / t_i$. But it’s entirely possible that different datasets of the same size, i.e., $d = d'$, will require different execution times $t \neq t'$ for the same job type under consideration. Over time, such variations will be “averaged out” in the executor speed estimates; i.e., each executor will experience the same task-difficulty distribution “per unit” input data (unless there is some bias so that some executors tend to receive more difficult tasks per unit input data for a given job). This motivates a forgetting factor $\alpha$ that this not close to zero.

Note that each application framework (different job types) will need to maintain its own estimates of (workload specific) executor speeds.
To see the effect of such adaptive workload partitioning, we performed an experiment with a two-node cluster where each node provides one CPU core. No resource isolation technology was used, so Spark executors could share CPU cycles with other processes. A sequence of fifty Spark WordCount jobs were presented through a submission queue. We introduced interfering processes [52] on one node at two different points in time during the experiment thus reducing the processing speed of Spark executors on that node. How Spark jobs were adaptively partition to re-balance their workloads is shown in Fig. 2.10.

One can see how overall job execution times (determined by the slowest task) increased dramatically but then rapidly fell as the task sizes were adapted with zero forgetting factor (here, for a given executor, execution time variation per unit document size - measured in MBytes - was low).

Figure 2.10. Adaptive workload balancing with introduced interfering processes at two points in time.

We did another experiments where one and 0.4 CPU cores are strictly isolated in two hosts (see Sec. 2.3.2.2.1 for more details) to compare the results in Sec. 2.3.2.2.1. The results are shown in Fig. 2.11. Spark learns the optimal way of partitioning workload after two trials so the map stage finish time is reduced to around 60 seconds, which is in line with the results shown in Fig. 2.12, where the optimal partitioning is immediately derived using the resource status provided by Mesos.

Our online adaptive task sizing can also be applied to the cases in the following sections (Sec. 2.3.2.2.1 and 2.3.2.2.2), where the computation capacities of the
nodes can be estimated and quantified a priori, but can be further tuned to achieve better performance.

2.3.2.2 Heterogeneous MacroTasking with provisioned instance types

In this section, we show how information regarding executors (e.g., known resource allocations to executors, information the service-level agreements) or some runtime “state” of an executor (e.g., token-bucket state) to determine initial/baseline heterogeneous task sizes. As above, task sizes be further adapted online based on, e.g., execution time information received from either cluster manager (e.g., Mesos) or monitoring services (e.g., AWS CloudWatch).

2.3.2.2.1 Heterogeneous MacroTasking (HeMT) - experiments with provisioned containers

We now compare HeMT with HomT through experiments with heterogeneous executors each assigned a different fraction of a core. Our implementation supports flexible CPU usage limitation by using containers. Baseline Spark does not support partial CPU usage. So we modified the Spark driver to be able to: accept a Mesos’ offer with partial CPU; launch an executor using the resources in the offer; and record the actual resources available to this executor so that the driver can use this information to rebalance the workload. Additionally, if the Spark’s executor is spawned on the container with partial CPU, we let Spark’s executor believe that it has one full core so that it is able to communicate with the driver asking for a task.
To evaluate the performance of HeMT with containerized and isolated resources, we did a set of Spark experiments on Mesos. The network bandwidth is large enough (\(\sim 500 \text{ Mbps}\)) so that CPU is the only bottleneck. In those experiments, we submitted Spark WordCount jobs with different tasking configurations to a Mesos cluster. WordCount is a simple two-stage Spark job in which most computations are done on the first map stage, so we can determine the effect of load-balancing by observing the finish time of the first stage. Our jobs are processing 2 GB data residing on a remote HDFS cluster. For each Spark job, we assign two executors to run associated tasks: one with one full core, the other with partial core.\(^6\) We used Mesos-supported CFS (complete fair scheduler) bandwidth control \(^5\) to limit the CPU usage of the containers.

Our experimental results are shown in Fig. 2.12. As we can see from the red beams, HeMT has good performance since it make Spark aware of the CPU fractions and balance its workload according to it. The U-shaped homogeneous tasking curve is similar to the results of \([14]\): When tasks sizes are too large (tasks too few in number) there are synchronization delays. When task sizes are too small (tasks are too large in number) there is microtasking overhead. Though \([14]\) gives a rule of thumb, an optimal homogeneous task size also needs to be learned.

2.3.2.2 HeMT - experiments with burstable instances  Amazon Web Service (AWS) provides a set of low-cost, general purpose instances types (T2 instances), called burstable instances. The CPU utilizations of those instances are governed by a token bucket mechanism. The CPU credits are earned and spent at a millisecond-level resolution, and the CPU credits earned when CPU(s) are idling can be used for future CPU bursts. One CPU credits can be used for one CPU running with 100% utilization for one minutes, and it can be fractured for a short CPU burst. Different types of T2 instances have different CPU credits earning rate, which is in line with their own baseline performance, and CPU caps, on which CPU credits stop accumulating. Table 2.1 shows the parameters regarding to the CPU earning behavior of T2 instances\(^7\) according to \([54]\).

On the free tier, AWS CloudWatch updates the CPU credits on an instance every 5 minutes by averaging the CPU credits over the most recent 5-minutes time.

\(^6\)Spark by default would create one map task for each HDFS block. So to make our experiments start with two tasks, we increase the HDFS block size from 128 MB to 1 GB. We use the same HDFS configuration in Sec. 2.3.2.2.2. For experiments with usual HDFS configuration, see Sec.
HeMT and even partitioning (including HomT and the default 2-way even partitioning). One executor was assigned with 0.4 CPU and the other a full CPU.

<table>
<thead>
<tr>
<th>Instance type</th>
<th>CPU credits earned per hour</th>
<th>CPU credits cap</th>
<th>vCPUs</th>
<th>Baseline CPU performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.nano</td>
<td>3</td>
<td>72</td>
<td>1</td>
<td>5%</td>
</tr>
<tr>
<td>t2.micro</td>
<td>6</td>
<td>144</td>
<td>1</td>
<td>10%</td>
</tr>
<tr>
<td>t2.small</td>
<td>12</td>
<td>288</td>
<td>1</td>
<td>20%</td>
</tr>
<tr>
<td>t2.medium</td>
<td>24</td>
<td>576</td>
<td>2</td>
<td>40%</td>
</tr>
<tr>
<td>t2.large</td>
<td>36</td>
<td>864</td>
<td>2</td>
<td>60%</td>
</tr>
<tr>
<td>t2.xlarge</td>
<td>54</td>
<td>1296</td>
<td>4</td>
<td>90%</td>
</tr>
<tr>
<td>t2.2xlarge</td>
<td>81</td>
<td>1944</td>
<td>8</td>
<td>135%</td>
</tr>
</tbody>
</table>

Table 2.1. CPU credit earning rates and caps of AWS burstable instances.

For example, t2.nano instance earns 3 CPU credits per hour, so if its initial CPU credit is 0, then the actual CPU running time can be at most 3 minutes in one hour, in line with $3/60 = 20\%$ baseline performance of t2.nano instance. Same for t2.medium instance, whose CPU credits earning rate is 24/hour. Suppose its initial CPU credit is again 0, the maximum total actual CPU running time is 24 minutes, in any sharing pattern between its two CPUs, in one hour, in line with its $24/60 = 40\%$ baseline performance. In terms of CPU credits cap, for example, t2.micro’s cap is 144, meaning if the instance keeps idling, its CPU credits stop growing when it has 144 credits.

User can optionally enable Detailed Monitoring by paying $2.10 down to $0.14 per instance per month for 1-minute updating frequency [55]. The updated
CPU credits can be accessed through AWS’s webpage dashboard or API.

HeMT can also be adapted to support lower cost\textsuperscript{8}, general purpose Amazon Web Services (AWS) burstable instance types (T2).

Given the current amounts of CPU credits and the baseline CPU performance, suppose we are able to estimate the computational workload of our job, the calculation of the amount of work that a node can process within a certain time, $W(t)$, can be easily evaluated: Suppose a t2.small instance initially has 4 CPU credits. If its vCPU is continually busy, then its CPU credits will be used up in $4/(1 - 0.2) = 5$ minutes, and, afterwards, its CPU performance will drop to the 20% baseline. The workload it can process in 10 minutes can be calculated as

$$W(10) = 1 \times \frac{4}{1 - 0.2} + 0.2 \times (10 - \frac{4}{1 - 0.2}) = 6,$$

reflecting the size of shaded area in Fig. 2.13.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig213.png}
\caption{Mapped 10-min workload for a t2.small instance with 4 initial CPU credits.}
\end{figure}

To divide a workload ($W_0$) to multiple servers with different amount of initial CPU credits so that they can finish at the same time, for each server, we first transform the time-credits plot in Fig. 2.13 into the time-workload plot as shown in Fig. 2.14. Superposing such time-workload graphs together into a single piecewise linear function ($\hat{W}(t)$), we can find $t'$ such that $\hat{W}(t') = W_0$. We then divide the

\textsuperscript{8}compared to on-demand instances allocated with maximum resources available to the burstable
workload proportionally according to $\{W_i(t')\}$, where $i$ is the index of node $i$.

For example, suppose we have three computation nodes with 4, 8, 12 initial CPU credits respectively, and the current data process job requires a CPU running at 100% performance for 20 minutes. We can first superpose the time-workload graphs for these three nodes together as $W_s(t)$, then find $t' = \frac{80}{11}$ such that $W_s(\frac{80}{11}) = 20$, as shown in Fig. 2.15. Finally we divide the entire workload for the three nodes according to their weights $\{W_1(\frac{80}{11}), W_2(\frac{80}{11}), W_3(\frac{80}{11})\} = \{\frac{60}{11}, \frac{80}{11}, \frac{80}{11}\} \propto \{3, 4, 4\}$.

As the CPU credit-monitoring API provided by AWS is not very responsive, it may not be useful to apply HeMT for short tasks. So, we created the following experimental scenario: First, there are only two types of nodes, one with sufficient CPU credits that will not be depleted throughout the entire life-span of a job, and the other with zero CPU credits\(^9\), where `sysbench` [52] was used to deplete CPU credits. That is, to create heterogeneity, we made depleted one node’s CPU credits so its CPU works with baseline performance (40% of CPU for AWS t2.medium instance). We let our map tasks fetch input data from HDFS. To make the duration of a single task long enough, we set the HDFS block size to 1GB, so each task would process 1GB data (one HDFS data block) in Spark’s default setting.

\(^9\)We can only ensure zero CPU credits on a particular node when starting the job subject to the AWS monitor’s update latency.
Fig. 2.16 shows the completion time of the map stage under different configurations. This set of experiments was done in a small Spark cluster with two executors, each with one core (on two separate AWS burstable instances). The tasks read input from a remote HDFS cluster consisting of four datanodes, each an AWS t2.small instances. In this experiment, the network bandwidth is large enough ($\sim 500$ Mbps) so that CPU is the only bottleneck.

The yellow beam of Fig. 2.16 shows one-σ confidence interval of a naive implementation of workload skewing, where we partition the data strictly according to CPU peak and baseline performance (1:0.4 for one core on AWS t2.medium). That is, the task running on the faster node gets $1/1.4$ of the total data, while the other gets the rest. However, we found that the node with zero CPU credit runs even slower than 40% of peak speed. We suspect that this is because the task running with baseline performance is likely facing a higher degree of CPU cache and TLB contention than the other task (e.g., it is possible that the first task is sharing a physical CPU with one or more other workloads while the second task has an entire CPU to itself). Our workload had a significant fraction of memory instructions that are delayed due to such cache/TLB contention. By employing short/trial probing tasks, we found that data partitioning by 1:0.32 further improves load-balancing, i.e., this fudge factor is learned from runtime observations (recall Sec. 2.3.2.1). This is shown by the red beam in Fig. 2.16 - the performance of
Figure 2.16. Microtasking (HomT) vs. macrotasking (HeMT) on AWS burstables (t2.medium), where HeMT yellow beam is without fudge factor and red beam is with fudge factor, when CPU is the only bottleneck.

HeMT with this fudge factor improves performance over the best configuration HomT (8-way) we had tried.

But we found the behavior of microtasking and macrotasking approaches were different when uplink bandwidth of a HDFS datanode was reduced to 50% (∼ 250Mbps), see Fig. 2.17. In this case, for the node with sufficient CPU credits, network I/O becomes the bottleneck, while CPU remains to be the bottleneck for the task running on the node with zero CPU credits. Note that 8-way partitioning is no longer one of the best HomT/microtasking approaches because this relatively coarse-grained partitioning\(^{10}\) fails to well-balance the workload in this case. It can also be observed that HeMT, including the naive CPU-credit-based partitioning (even it does not make sense in this scenario since the node with sufficient CPU credits is now bottlenecked by the network), started to significantly outperform HomT, because latter is more likely to incur datanode uplink contention, as explained in Sec. 2.3.2.

\(^{10}\)where a task running on credit-abundant node would run for about 15 seconds, while a task running on the other node would run for about 30 seconds
Figure 2.17. Stage completion time when we cut the bandwidth to 50%, where the node with sufficient CPU credits becomes bottlenecked by network bandwidth. Orange area represents the overlapping area between the confidence interval of naive CPU credit-based partitioning (yellow) and that of the adjusted partitioning (red).

2.3.2.2.3 HeMT - repartitioning on multiple program stages  Our heterogeneous macrotasking can certainly be applied to more realistic workload. A typical MapReduce workload consists of one or more jobs, each job has multiple basic computation stages presented in the previous sections concatenated together through data shuffling. So for the first computation stage, we can simply divide the initial input data according to the computation capacities of the executors.

A partitioner defines the how a task assigns its intermediate results to different “buckets” which will be fetched by different tasks in the following stage respectively. For the following stages, task data are fetched from the intermediate outputs of the tasks in the previous stages. The tasks in the previous stages first shuffle the processed records into different buckets (each corresponding to one fetching task in a future stage) according to a partitioner function, then those buckets are written onto storage media for associated future tasks to fetch. The default hash partitioner shuffles those records into those buckets in a statistically even fashion. So, we need to define a new partitioner that can skew the shuffle buckets for HeMT. We show one implementation of skewing using hash code in Algorithm 1.

The comparison of effective data flows when using the default hash partitioner
**Data:** Record $r$ to be assigned to a bucket; array of executors’ computation capacities, $executors$

**Result:** The index of the target bucket

$sum = 0$;

for $e$ in 0 until $executors.length$ do

$\text{sum } += \text{executors}[e]$;

$\text{executors}[e] = \text{sum}$;

end

$hash = r\text{.hashCode mod } executors\text{.sum}$;

**return** the number of elements in $executors$ greater than or equal to $hash$.

**Algorithm 1:** Partitioning function of skewed hash partitioner

and our skewed hash partitioner respectively is shown in Fig. 2.18\textsuperscript{11}. Relevant idea of balancing workload through partitioner can be found in [16,56].

![Data flows in even hash shuffling and skewed hash shuffling respectively.](image)

**Figure 2.18.** Data flows in even hash shuffling and skewed hash shuffling respectively.

We present the performance of HeMT using two typical workloads - K-Means and PageRank. Those two have different and representative computation patterns. K-Means consists of repetitive simple two-stage Spark jobs. PageRank, on the other hand, is a single Spark job containing multiple computation stages concatenated together through shuffling.

Again, we run K-Means on the cluster with two executors hosted on two containers, one was allocated with one CPU core, the other was allocated with 0.4 core. To make results more consistent, instead of setting a convergence criterion to stop the iterations, we fix the number of iterations to 30. The input source is 256 MB data file on HDFS, with block size 128 MB (So there are two blocks).

\textsuperscript{11}Certainly, more sophisticated partitioning algorithm can be made given more information regarding key distribution and processing complexity of each record.
The entire job finish times of HeMT and HoMT are shown in Fig. 2.19, which is consistent with the single-stage results shown in the previous sections.

![Figure 2.19. K-Means finish time.](image)

On the same cluster, we run PageRank for 100 iterations. The results are shown in Fig. 2.20. Note that the PageRank, compared with K-Means, is more sensitive to microtasking, because each iteration of PageRank is relatively short (around 10s in the default 2-way partitioning), therefore each task is shorter as well. For example, if we use 64-way partitioning, then each task generally only 0.1 - 0.2 seconds. Therefore, the relative task scheduling overhead would be larger in PageRank workload.

### 2.4 Conclusions

This chapter addressed modeling and optimizing parallel-computing system and distributed jobs. We started by modeling a single-stage fork-join system, which receives randomly arriving parallel computation requests and serves them at a possibly not constant rate. The jobs in such systems are divided into multiple tasks which run in parallel. The tasks are coordinated in a way that their outputs must synchronized at a computation barrier to complete the job. QoS bounds were derived under “network calculus” conditions.
We then focused on the performance of a single job in a distributed system. In particular, we investigated the “straggler problem”, where, due to the computing capacity heterogeneity among nodes in the system or the imbalance of workload-computational node assignment, the execution times of the distributed tasks vary largely, so that some nodes idle for long periods and their resources not well utilized. Two major approaches were discussed. One is homogeneous microtasking, where tasks are very small and equally-sized so that when heterogeneous nodes pull pending tasks, faster nodes would fetch more workload while slower nodes would fetch less. Hence the overall workload is balanced if the tasks are divided small enough. The other is heterogeneous macrotasking, where the workload is rebalanced and assigned to the computing nodes according to the modeling of their computational capacities. We implemented the workload-skewed macrotasking on Apache Spark. With several sets of experiments done on containers and AWS burstable instances, we demonstrate that our macrotasking approach, like microtasking, can well balance the workload in a heterogeneous environment while avoiding the side effects of microtasking such as scheduling overhead and I/O inefficiency.

Figure 2.20. PageRank finish time.
Chapter 3  
Fairness in Private-Cloud Scheduling

3.1 Introduction

Schedulers in private or public clouds have different perspective. A public cloud offers a certain set of services with according charges; notable examples are Amazon Web Service (AWS), Google Compute Engine (GCE), Microsoft Azure. A main purpose of public cloud service provider is to maximize its profit subject to service-level agreements (SLAs) [57–59]. The public clouds tenants need to acquire a certain amount of computing resources to satisfy their needs while minimizing their costs [60–62]. Depending on the SLA, specifically, how much resources a tenant would require and what the tenant does with its resources are not the concern of the public cloud.

We herein focus on the scheduling resources to jobs in private cloud. In a private distributed computing system such as Penn State’s Advanced Cyber-Infrastructure (ACI) or Facebook’s computing cluster [27], the system resources are shared among multiple groups of employees who usually will not pay for using the IT services. Therefore, without pricing as an “invisible hand” [63], a good system scheduler is expected to make quick resource allocation decisions to achieve high scalability [64], fair resource share [18, 20, 23] and efficient resource utilization [65, 66].

This chapter reviews the conceptual evolution of fairness over the recent years, the mathematical setup for batch resource scheduling with fairness, and our practical insights of maintaining fairness and efficiency under dynamic resource allocation.
3.2 The background on the fairness notions

If we consider a system with only one resource type, max-min fairness [18, 67, 68] is a well-known fairness criterion. If an allocation is max-min fair, then we cannot increase the allocation for a user/tenant without decreasing the allocation of users who have equal or smaller total allocation.

Things are more complicated in the multi-resource case, since the allocation has to be made on each resource type and different resource types are subject to server constraints. Consider a private cloud system with two resource types - CPUs and memory - and two users - user A running CPU-intensive jobs and user B running memory-intensive jobs. We may consider it as two separate single-resource type allocations and apply max-min fair allocation on CPUs and memory respectively. If these two users do not have any placement preference, then each of them would equally share CPUs and memory, which causes waste of memory by user A and waste of CPUs by user B. Therefore, a better fairness criterion is needed to be identified.

Researchers have proposed several desired properties of a good fairness criterion:

1) **Sharing incentive**, which ensures that each user is better off sharing the distributed system, than statically partitioning the system.

2) **Strategy-proofness**, which ensures that no user can benefit by lying about its resource demands.

3) **Envy-freeness**, which ensures that no user prefers the allocation of another user.

4) **Pareto efficiency**, which ensures that no other allocation can make any one user better off without hurting another user’s benefit.

Clearly the allocation in the above example is not Pareto efficient as moving user B’s wasted CPUs to A will benefit A without causing any loss to B.

Over the years, several fairness notions have been proposed such as Dominant Resource Fairness (DRF) [18], DRF with Heterogeneity (DRFH) [20], Containerized DRF (CDRF) [19], Task Share Fairness (TSF) [69] and Per-Server Dominant Fair Share (PS-DSF) [21]. DRF models the distributed system as a single large server by aggregating the resources of all the servers together. For a user $i$ with the current resource allocation $a_i = (a_{i,1}, a_{i,2}, ...)$\(^1\) where $a_{i,r}$ is the allocation of resource type $r$

\(^1\)If resources are used in task-level granularity, let $x_i$ be the number of tasks of user $i$, and
and a system with total resources \( \{c_1, c_2, \ldots\} \) \( (c_r \) is total amount of resource type \( r) \), define \( i \)'s dominant resource as
\[
 r_i = \arg \max_r \frac{a_{i,r}}{c_r}.
\]
Alternatively, if the resources are allocated at task granularity and the dominant share of user \( i \) with demand vector (i.e., the resources required by a single task) \( d_i = (d_{i,1}, d_{i,2}, \ldots) \), its dominant resource can also be defined as
\[
 r_i = \arg \max_r \frac{d_{i,r}}{c_r}.
\]
The dominant resource share of user \( i \) is
\[
 s_i = \frac{a_{i,r_i}}{c_{r_i}}.
\]
So the DRF allocation/scheduling problem becomes
\[
 \max \min_i \{s_i\}.
\]
\[
 \text{s.t. } \sum_i a_{i,r} \leq c_r, \forall r.
\]

DRFH, CDRF and TSF respect the resource capacity and allocation of individual servers. They have the similar mathematical set-up - the only difference is the definition of share. DRFH generalizes DRF to a cluster with heterogeneous servers by further splitting the resource allocations and capacity constraints of DRF down to individual servers. Let \( c_j = (c_{j,1}, c_{j,2}, \ldots) \) denote the resource capacities of server \( j \), and \( (a_{i,j,r}) \) be the amount of resource \( r \) on server \( j \) allocated to user \( i \). So \( a_{i,r} = \sum_j a_{i,j,r} \) and \( c_r = \sum_j c_{j,r} \). The DRFH scheduling problem becomes
\[
 \max \min_i \{s_i\}.
\]
\[
 \text{s.t. } \sum_i a_{i,j,r} \leq c_{j,r}, \forall r, j.
\]
However, \cite{19} shows that DRFH does not satisfy the sharing incentive property. 

\( \{d_{i,r}\} \) be the d-vector of user \( i \), then \( a_{i,r} = x_{d_{i,r}} \).

\( \text{If resources are used in task-level granularity, let } x_{i,j} \text{ be the number of tasks of user } i \text{ running on server } j, \text{ and } \{d_{i,r}\} \text{ be the d-vector of user } i, \text{ then } a_{i,j,r} = x_{i,j}d_{i,r}. \)
Consider a case where there are two users with demand vector $d_1 = (1, 1/2)$, $d_2 = (1/2, 1)$ and two servers each with resources $c_1 = (15, 15), c_2 = (16, 0)$. Then under DRFH, user 1 gets 12 tasks while user 2 gets 6, because although server $(16, 0)$ is useless in practice, it will implicitly help user 1 by making its dominate resource $r = 2$. However, suppose we evenly divide these two servers, so each user will get $(7.5, 7.5)$ from server 1 and $(8, 0)$ from server 2. Clearly user 2 is more satisfied since it now can create 7 tasks. Therefore, DRFH does not satisfy sharing incentive since in this example user 2 prefers static even partitioning over sharing the cluster under the DRFH criterion.

CDRF solves this problem by replacing dominant resource share in 3.1 with the number of tasks created of a user allocated so far over the number of tasks that can be created if the user monopolizes the entire system. Therefore in the above example, with CDRF, server 2 has no impact on scheduling decisions because no task can be created on it, so user 1 and user 2 both get 10 tasks running on server 1.

Although CDRF satisfies sharing incentive, it loses strategy-proofness and envy-freeness when user placement constraints are brought into the picture [69]. First consider the example where we have two identical servers with $c = (18, 18)$, and two users with demand vector $d_1 = (1, 2)$ and $d = (1, 3)$. Suppose user 1 can run jobs on both server 1 and 2, but user 2 can only run on server 2. If both users are truthful, then according to CDRF, user 1 will create 18 tasks if it monopolizes the system while user 2 can only create 6 on server 2. If CDRF is used in allocation, user 1 has 9 tasks running on server 1 and 3 on server 2, while user 2, punished for being picky, has only 4 tasks running on server 2. However, if user 2 lies about its placement preference by claiming it can run on both servers, then the scheduler would falsely think user 2 can create 12 tasks if it monopolizes the system. As a result, user 1 would run 9 tasks running on server 1 and user 2, benefiting from being untruthful, would run 6 tasks running on server 2.

Next consider the example where there is only one type of resource. There are three identical servers, all have $c = (3)$. There are seven users with the same demand vector $d = (1)$. Suppose user 1 can only run task on server 1; user 3 and 4 can only run on server 2; user 5, 6 and 7 can only run on server 3; user 2 can run on any server. The resulting CDRF allocation is shown in Fig. 3.1, which demonstrates that CDRF is not envy-free, since user 1 envies the allocation of user
Figure 3.1. CDRF is not envy-free. Since user 2 can run on all those servers, it reaches the 1/3 share when running 3 tasks, while others can only run 1 task to reach the same share.

TSF slightly modifies CDRF by removing the placement constraints, assuming users can run all the servers. It solves the strategy-proofness problem in the example that exposes CDRF’s loss strategy-proofness because user 2 need not lie about its placement preference - the scheduler automatically assumes that it can run all the servers.

The way that TSF provides strategy-proofness is controversial. It solves the problem caused by placement constraints by removing them. Hence, with TSF, there is no way that the system administrator can punish those picky users$^3$.

TSF also solves the envy-freeness problem in the example that exposes CDRF’s loss of envy-freeness. If TSF is enforced, user 1 and 2 have the same allocation priority. As a result, user 1 would create 2 tasks on server 1 while user 2 would create 2 tasks on server 1 and 2 respectively.

Under PS-DSF $[21]$, the Virtual Dominant Share (VDS) for user $i$ on server $i$ is defined as

$$s_{j,i} = \max_r \frac{a_{i,r}}{c_{j,r}}.$$  

On server $i$, PS-DSF tries to find $\max \min s_{j,i}$ subject to the server constraints in (3.2). Unlike other fairness criteria, $s_{j,i}$ is correlated among servers, i.e., PS-DSF involves a set of optimization problems whose objective functions are correlated. The authors in [21] argued that PS-DSF is equivalent to the proportional fairness problem, which again is a optimization with single objective function. It can be shown that PS-DSF satisfies all the above properties except for Pareto efficiency.

$^3$Admittedly, whether we should punish picky users is an arguable issue. It should be decided by the service provider.
The properties of the above fairness criteria are summarized in the following table [21,69].

<table>
<thead>
<tr>
<th>Property</th>
<th>DRFH</th>
<th>CDRF</th>
<th>TSF</th>
<th>PS-DSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing incentive</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Strategy-proofness</td>
<td>√</td>
<td>?</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Envy-freeness</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Pareto efficiency</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

### 3.3 Static resource allocation

In static resource allocation, the user demand and resource pool are known prior to the allocating process, so the allocator can make allocation decisions once for all by solving an optimization problem with multiple constraints such as server capacities, placement preferences, fairness, etc. The following discussion only considers server capacities and fairness, although it is easy to extend to incorporate user placement preferences [19,69,70].

#### 3.3.1 Max-min fairness

As mentioned in Sec. 3.2, max-min (MMF) has been widely studied [19–21,69]. Consider the following general-purpose fairness criterion for framework \( n \),

\[
U_n = \phi_n^{-1} \sum_{i} u_{n,i} x_{n,i},
\]

for scalars \( u_{n,i} > 0 \) and priorities \( \phi_n > 0 \). Under DRF [18,20], frameworks \( n \) are selected using criterion

\[
M_n = \frac{1}{\phi_n} x_n \max_r \frac{d_{n,r}}{\sum_j c_{j,r}},
\]

where \( x_n = \sum_i x_{n,i} \). The PS-DSF criterion can be written as

\[
K_{n,j} = \sum_i x_{n,i} \frac{d_{n,\rho(n,j)}}{\phi_n c_{j,\rho(n,j)}}
\]

where bottleneck resource \( \rho \) is such that

\[
d_{n,\rho(n,j)}/c_{j,\rho(n,j)} := \max_r d_{n,r}/c_{j,r} \text{ when } \delta_{n,j} = 1.
\]
The feasibility conditions are

$$\forall i, r, \sum_n x_{n,i} d_{n,r} \leq c_{i,r}. \quad (3.7)$$

Regarding fully booked resources in server $i$ under allocations $x = \{x_{n,i}\}$, let

$$R_i := \{(x, r) \mid \sum_n x_{n,i} d_{n,r} = c_{i,r}\}. \quad (3.8)$$

**Definition 1.** A feasible allocation $\{x_{n,i}\}$ satisfying (3.7) is said to be $U$-MMF if:

$$U_\ell > U_m, \ x_m, x_\ell > 0, \text{ and } \exists r \text{ s.t. } \sum_n x_{n,i} d_{n,r} = c_{i,r}$$

implies that $x_{\ell,i} = 0$.

Note that if instead $x_{\ell,i} > 0$ in this definition, then $x_{\ell,i}$ can be reduced and $x_m$ increased to reduce $U_\ell - U_m$. Also, if $\{x_{n,i}\}$ is U-MMF and $x_m, x_{\ell,i} > 0$ for some server $i$ then $U_m = U_\ell$. Consider the optimization problem

$$\max_x \sum_n \phi_n g(U_n) \text{ s.t. } (3.7), \quad (3.9)$$

for strictly concave and increasing $g$ with $g(0) = 0$.

**Proposition 1.** A solution $x = \{x_{n,i}\}$ of the optimization (3.9) s.t. (3.7) has at least one resource $r$ fully booked in each server $i$. In addition, there is a unique $U$-MMF solution if also

$$(x, r) \in R_i \Rightarrow d_{m,r} = d_{l,r}. \quad (3.10)$$

Note that for uniqueness, the proof given in Appendix A, requires the strong assumption (3.10) that frameworks that can share servers have identical demand parameters $d$ for fully utilized resources.
3.3.2 Proportional fairness

Another alternative notion of fairness is proportional fairness. For weighted proportional fairness, consider the objective

$$\max_x \sum_n \phi_n g_a(x_n),$$  \hspace{1cm} (3.11)

i.e., without dividing by \(\phi_n\) in the argument of \(g_a\) \cite{71}. For parameter \(a > 0\) specifically take

$$g_a(X) = \begin{cases} \log(X) & \text{if } a = 1 \\ (1-a)^{-1}X^{1-a} & \text{else} \end{cases} \hspace{1cm} (3.12)$$

i.e., \(g'_a(X) = 1/X^a\), again see \cite{71}. Obviously, in the case of \(a = 1\) \((g = \log)\), whether the factor \(\phi\) is in the argument of \(g\) is immaterial.

The following generalizes Lemma 2 of \cite{71} on Proportional Fairness. See also the proportional-fairness/efficiency trade-off framework of \cite{22} for a single server.

**Proposition 2.** A solution \(x^*\) of the optimization (3.11) s.t. (3.7) is uniquely \((weighted) (\phi, a) x\)-PF, i.e., for any other feasible solution \(x\),

$$\Phi(x, x^*) := \sum_n \phi_n \frac{x_n - x_n^*}{(x_n^*)^a} \leq 0. \hspace{1cm} (3.13)$$

The proof is given in Appendix B. A possible definition of the **efficiency** of a feasible allocation is

$$\sum_n \phi_n \sum_i x_{n,i} = \sum_n \phi_n x_n, \hspace{1cm} (3.14)$$

(corresponding to \(a = 0\)), i.e., the weighted total number of tasks scheduled. The optimization of Proposition 2 with \(a = 1\) gives an allocation \(x^*\) that is related to a task efficient allocation. Clearly, \(x^*\) satisfying (3.13) for all other allocations \(x\) with \(a = 1\) does not necessarily maximize (3.14). This issue is analogous to estimating the mean of the ratio of positive random variables \(E(X/X^*)\) using the ratio of the means \(EX/EX^*\), see e.g. p. 351 of \cite{72} or (11) of \cite{73}. For simplicity in the following, we use (3.14) instead of (3.13).

Note that the priority \(\phi_n\) of framework \(n\) could factor its resource footprint \(\{d_{n,r}\}_r\). Alternatively, the resource footprints of the frameworks can be explicitly
incorporated into the main optimization objective via a fairness criterion. The proof of the following corollary is just as that of Proposition 2. Recall that the generic fairness criterion $U_n$ (3.3) is a linear combination of $\{x_{n,i}\}_i$.

**Corollary 1.** A solution $x^*$ of the optimization problem

$$
\max_x \sum_n \phi_n \log(U_n) \quad s.t. \quad (3.7)
$$

is uniquely $(\phi, 1)$ U-PF, i.e., for any other feasible $x$,

$$
\sum_n \phi_n \frac{U_n - U_n^*}{U_n^*} \leq 0.
$$

Again, optimal $\{U_n^*\}$ would be unique but $x^* = \{x_{n,i}^*\}_{n,i}$ may not be.

Recall for DRF and PS-DSF, the $K_n = \sum_i K_{n,i}$ and $M_n$, respectively, are proportional to $x_n$. Thus, using $U_n = K_n$ or $U_n = M_n$ in Corollary 1 reduces to the result of Proposition 2 when $a = 1$.

### 3.4 Dynamic resource allocation

In the static environment where the users and servers are fixed, the allocation can be made at once after solving an optimization problem. However, in practice, we usually face a dynamic environment, where users’ computation requests frequently arrive and leave the system; tasks are asynchronously created with the obtained resources and terminated, releasing the occupied resources back to the system; new servers or resources might be added to a running system. Therefore, resource allocation is not one-off deal, numerous scheduling decision will be made during the lifespan of the system.

There are two ways of triggering resource scheduling. One uses batch window, scheduling for the user workload arrived and resources accumulated during the scheduling interval. In such batch scheduling, the system administrator needs to carefully tune the scheduling interval. If the scheduling interval is too large, then the system will not response quickly to demands. For example, it is unacceptable for a task that will run 30 seconds to wait for 10 seconds to be scheduled.

The other is event-driven scheduling - the scheduling is triggered when an event,
such as new resources or users registered, resources released back to the system, etc., happens. In this way, the system administrator is free from trading-off on the length of batch window. However, event-driven scheduler tends to be more short-sighted than batch scheduling. For example, when the first user registers, the event-driven scheduler will allocate the entire resources to the only user. If the second user arrives at the next second, it has to wait until the first user releases resources back. The batch scheduler, on the other hand, would still hold on for the remaining batch window, in which other new users may arrive, so that the scheduler can make better (fairer) allocation.

The scheduling also falls into two categories in terms of how to derive the optimal allocation. One way is to directly solve the optimization problems as in the last section. The complexity of solving the problem can be high, especially when the resources are consumed on the basis of tasks, meaning the optimization becomes a integer-programming problem, which is NP-hard [74]. Hence directly solving the optimization problem would cause extremely high computation overhead if the scheduling interval is small in batch scheduling or the environment drastically changes in event-driven scheduling. Alternatively, when achieving max-min fairness, the scheduler can greedily allocate resources using progressive filling. In progressive filling, the scheduler always allocates a small portion of the resources to the user with the minimum share so that all the users’ shares slowly grow with the same rate until no more resources can be allocated.

As will be shown in the following sections, progressive filling may not yield the global optimal allocation. Due to its simplicity and empirically good performance, it is used in many main-stream resource managers, such as Mesos [1] and Yarn.

3.5 Resource allocation in practice

Several resource managers have been developed in the recent years, such as Apache Mesos, Apache Yarn, Kubernetes. They apply the similar master-slave architecture [75] [76] [77]. In this section we briefly explain how the system resources are allocated in practice, using Apache Mesos as an example.

Apache Mesos is an open-source cluster resource manager which dynamically allocates resources from slaves to registered frameworks such as Hadoop, Spark, and Flink. It uses an offer-based two-level scheduling architecture as shown in Fig.
3.2. Instead of unilaterally assigning resources to frameworks, the Mesos master provides resource offers to the application frameworks. The scheduler in framework then decides whether to accept the resource offers or not. Once the framework decides to accept the offer, it schedules a task to run by building task information including resources needed, the slave worker on which the task resides, executor to run, etc., and sends this information back to master. The master then accordingly assigns the task to the executor on a slave and launches the executor if it is not running. To run a framework on Mesos, the user needs to write her own framework scheduler and executor, unless the framework, like Spark and Hadoop, already has module to communicate with Mesos. Thus, the behavior of a system supported by Mesos depends not only on Mesos, but also on the customized scheduler and executor inside the framework.

Figure 3.2. Offer-based two-level scheduling in Mesos.

Mesos combines both batch and event-driven scheduling. The default allocation interval is 1 second, meaning every second, the master will check available resources and send resource offers to the frameworks if possible. Besides, events such as new framework/slave added, slave info updated, and framework offer filter removed will also trigger scheduling.

The default Mesos scheduler tries to achieve dominant resource fairness (DRF) [1]. More specifically, the Mesos master keeps track of resource share among all resources for each framework, and always sends resource offers to the framework
with the minimum “dominant share” first. Since different frameworks have different per-task requirements and Mesos is myopic to those requirement, a coarse-grained allocation is used, that is, Mesos by default would include all the available resources on a slave in an offer. However, as we will see in the following section, the coarse-grained allocation can lead to a poor DRF performance, especially when we have many frameworks but not-so-many slaves. To remedy such problem, Mesos provides the cluster administrator a control knob to manually configure resource upper bounds in its offers. In Mesos, one resource offer never contains resources from different slaves, which implies resource affinity in task creation, i.e., a task can only be created by using the resources from one slave. Therefore, more accurately, the dominant resource fairness adopted by Mesos is in fact DRFH [20]. Moreover, Mesos delegates allocation decisions to a pluggable allocation module, so that developers can tailor the scheduling algorithm to their needs [1].

Cluster administrator can use weights in Mesos scheduling. However, instead of directly assigning weights to frameworks, the administrator assigns weights to roles [78]. A role can be considered as a label shared among frameworks in a group, and a framework can optionally specify the role it would like to be associated with when it registers with the master. With roles, Mesos’ master allocates resources in a hierarchical fashion: it picks the roles in ascending order of their dominant share; within one role, Mesos sends offers to frameworks in ascending order of their dominant share.

3.6 Maintaining both fairness and efficiency in progressive filling

The fairness criteria introduced in Sec. 3.2 can be theoretically proved to have desired properties regarding fairness. However, in practice, especially in progressive filling allocation, solely focusing on fairness may lead to poor resource utilization.

Consider a cluster with two slaves A and B having IT resources (12 CPUs, 6 GB memory) and (6 CPUs, 12 GB memory) respectively. Suppose two users 1 and 2 whose per-"task" requirements are (2 CPUs, 1 GB memory) and (1 CPU, 2 GB memory) respectively. If DRF is used and the resources from these servers are allocated in random order, then it is possible that we could assign all the resources...
Figure 3.3. DRF in the cluster with heterogeneous slaves.

from slave A to the memory-intensive user 1 and assign all the resources from slave B to the CPU-intensive user 2, and the allocation stops in the state as shown on the left side in Fig. 3.3, which has much lower resource utilization than the optimal one shown on the right.

To further study the low efficiency issue while enforcing fairness, we implemented different fairness criteria on Apache Mesos, tested them with Spark workloads and explored some practical insights on resource scheduling and other hidden properties of some criteria that could affect the resource utilization.

3.6.1 Running Spark on Mesos

In Spark, a job (Mesos framework) is divided into multiple tasks. Multiple Spark executors will be spawned for a job. The executors can simultaneously run a certain maximum number of tasks depending on how many cores on the executor and how many cores are required per task; when a task completes, the executor informs the driver to request another, i.e. executors pull in work. Each executor is a Mesos task in the default “coarse-grained” mode [79] and an executor resides in a container of a Mesos agent [80]. Plural executors can simultaneously reside on a single Mesos agent. An executor usually terminates as the entire Spark job terminates [81]. When starting a Spark job, the resources required to start an executor (d) and the maximum number of executors that can be created to execute the tasks of the job, may be specified. The Spark driver will attempt to use as much of its allocated resources as possible.
3.6.2 Experiment configuration

In our experiments, there are two Spark submission groups (“roles” in Mesos’ jargon): group Pi submits jobs that accurately calculate $\pi = 3.1415...$ via Monte Carlo simulation; group WordCount submits word-count jobs for a 700MB+ document. The executors of Pi require 2 CPUs and about 2 GB memory (Pi is CPU bottlenecked), while those of WordCount require 1 CPU and about 3.5 GB memory (WordCount is memory bottlenecked). Each group has five job submission queues, which means there could be ten jobs running on the cluster at the same time. Each queue initially has fifty jobs. Again, each job is divided into tasks and tasks are run in plural Spark executors (Mesos tasks) running on different Mesos agents.

The Mesos agents are six servers (AWS c3.2xlarge virtual-machine instances), two each of three types in our cluster. A type-1 server provides 4 CPUs and 14 GB memory, so it would be well utilized by 4 WordCount tasks. A type-2 server provides 8 CPUs and 8 GB memory, so it would be well utilized by 4 Pi tasks. A type-3 server provides 6 CPUs and 11 GB memory, so it would be well utilized by 2 Pi and 2 WordCount tasks. The Mesos master operates in a c3.2xlarge with 8 cores and 15 GB memory.

The experiment setup is illustrated in Figure 3.4.

3.6.3 Oblivious and workload characterized allocation

There are two progressive filling allocation modes working under different circumstances. The original Mesos provides oblivious allocation, where the allocator do not need to know the resource demands of the users. When a slave has resources available, the allocator tends to allocate all the resources on that slave to a user with minimum share. This coarse grained allocation can roughly approximate fair allocation especially when the system has many slaves or the users are configured to accept part of the resource offer to create one task.

In our customized Mesos, the workload-characterized (fine-grained) allocation is also supported, where each user simply informs the Mesos allocator of its resource demands per task. The Mesos allocator then selects a single task worth of resources from a given slave with unassigned or recently released resources.

The concrete workflow of those above allocations is shown in Fig. 3.5.

---

4 Called “coarse-grained allocation” in Mesos
Figure 3.4. Experiment setup.

Figure 3.5. Workflow of oblivious and workload-characterized allocation.
We use DRF as an example to illustrate the difference between those two allocation modes. The monitored traces on allocated CPU and memory are shown in Fig. 3.6. First note that theoretically oblivious DRF cannot reach the optimal allocations since it tends to include all the remaining resources in an offer to one user, which makes it hard to allocate optimally on type-3 servers. However, occasionally we would observe that the oblivious DRF momentarily stayed in the optimal state (e.g., at around 4200 sec). Besides, according to the previous oblivious DRF description in Sec. 3.6.3, there could be at most six jobs running at the same time, each occupying a server. However, we observed that sometimes there could be ten jobs running at the same time (which is not shown in Fig. 3.6). The above phenomena happened because when a Spark job finishes, its executors may not simultaneously release resources from the Mesos allocator’s point-of-view. So under oblivious allocation, it’s possible that multiple Spark frameworks can share the same server, as is typically the case under workload-characterized scheduling.

Secondly note that in the traces of workload-characterized DRF, there are two phase changes at around 2700 and 4700 sec. This is because occasionally group WordCount and Pi release some resources at the same time, in which a resource swap between two groups may happen. For example, initially a type-1 server has two WordCount tasks and one Pi and a type-2 server has two Pi tasks and one WordCount tasks. If group Pi releases resources on that type-1 server and group WordCount releases resources on that type 2 server at the same time, then with 50% chance, group WordCount will receive resources on type-1 server and and Pi will receive resources on type-2 server, leading to a higher resource utilization. It is possible that such resource swap may lower the resource utilization. But it requires that at the same time each group releases resources from multiple tasks on the same server. Such event is more unlikely to happen.

Despite those above contingencies, oblivious allocation is a coarse-grained enforcement of progressive filling, making uneven resource distribution between two groups and among jobs in the same group. Therefore, in Fig. 3.6, we see under oblivious allocation the amount of allocated resources drops more sharply when a Spark job ends, and variance of utilized resources under oblivious allocation is

\[\text{5For example, when a type-1 server has four WordCount tasks and a type-2 server has four Pi tasks, WordCount group releases resources from two tasks which will later be used to create a Pi task, and Pi group releases resources from two tasks which will later be used to create a WordCount task.}\]
Figure 3.6. Comparison between oblivious and workload-characterized modes under DRF.

larger than under workload-characterized. Consequently, the entire job-batch tends to finish sooner under workload-characterized allocator.

### 3.6.4 Fairness criteria in workload-characterized mode

In this subsection, we compare the allocation efficiency among different fairness criteria in workload-characterized dynamic allocation mode. The comparison between DRF and PS-DSF is shown in Fig. 3.7. As mentioned in Fig. 3.5, the DRF implementation in Mesos first randomly shuffles the available servers then allocates the resources on those servers in that random order. Since DRF is not a server-specific fairness criterion, the random shuffle allocation would easily lead to low resource utilization, as illustrated in Fig. 3.3. PS-DSF, on the other hand,
is a server-specific criterion, i.e., on each server, it calculates the shares of users based on the characteristic on that server. Therefore, it is more likely to allocate a server’s resources to users whose demand pattern matches the server best. As a result, in Fig. 3.7, the resource utilization of PS-DSF is higher than DRF and the jobs under PS-DSF finish sooner.

![Fractions of Allocated CPUs](image1)

![Fractions of Allocated Memory](image2)

**Figure 3.7.** Comparison between DRF and PS-DSF in workload-characterized mode.

The comparison between another pair of server-myopic and server-specific representatives - TSF and rPS-DSF - is shown in Fig. 3.8, which is consistent with the above result.

Some heuristics can be used to solve the low allocation efficiency problem in those server-myopic criteria. For example, in [20], authors proposed Best-fit DRF, in which once a user is picked according to DRF, the scheduler selects the resources from the server whose normalized resource vector is closest to the user’s normalized demand vector in $L^1$-norm distance. It can improve the resource allocation efficiency, when the total amount of allocatable resources to the users is constant.
Figure 3.8. Comparison between TSF and rPS-DSF in workload-characterized mode.

Although Best-fit DRF introduces the sever-selective feature (where to allocate), it does not change the DRF’s server-myopic nature when determining who to allocate to. Hence in some situations, Best-fit DRF is still unable to yield most efficient allocation. Suppose under a current allocation, we have one Spark-Pi and two Spark-WordCount executors on the type-1 server, two Spark-Pi and one Spark-WordCount executors on the type-2 server, and two Spark-Pi and two Spark-WordCount executors on the type-3 server. So, whenever a Pi or WordCount framework releases its executor’s resources back to the cluster, its DRF “score” is reduced so the scheduler will always sends a resource offer to the same client framework in this scenario. On the other hand, rPS-DSF will make a decision considering the amount of (remaining) resources on the server, and so will make a more efficient allocation.

We illustrate this with the example of Figure 3.10. In this experiment, we let
Figure 3.9. Comparison between DRF, PS-DSF and Best-fit DRF in workload-characterized mode.

Each group submit their Spark jobs through five queues with 20 jobs each. To create the above scenario, instead of exposing all the servers to the client frameworks, we register servers one by one from type-1 to type-3. From the figure, note that both rPS-DSF and BF-DRF have an initial inefficient memory allocation, but rPS-DSF is able to adapt and quickly increase its memory efficiency, while BF-DRF does not.

3.7 Conclusion

In this chapter, we studied two important aspects of resource allocation in the cloud. We first go over several proposed fairness criteria. Those criteria are evaluated with desired properties - sharing incentive, strategy-proofness, envy-freeness, and Parento efficiency. A static allocation scenario was first considered
Figure 3.10. Performance of Best-fit DRF and rPS-DSF in the system who will add more resources in the future.

where the resource demands come in a batch and fair allocation can be reduced to an optimization problem constrained by resource capacities of servers. We studied both max-min and proportional fairness, the latter related to efficiency.

In the latter section, we move on to a more practical case where resources and demands arrive and depart the system in an dynamic fashion. Progressive filling was used to ensure max-min fairness in such dynamic system. In a system with multiple heterogeneous resource pools, if a fairness criterion fails to recognize such heterogeneity, then progressive filling could lead to inefficient allocation. We used illustrative examples and experiments to illustrate this problem and implemented several alternative fairness criteria in Apache Mesos. Experimental results showed that our implementation can improve resource utilization and reduce execution times.
Chapter 4 | Defense against Resource Abuse in Distributed System

Generally, a public distributed computing system is designed for multiple users and has interfaces to the public commodity Internet. Therefore, it needs to be robust enough against resource abuse, i.e., excess resources occupied by some malicious users and so unavailable to others, which is commonly known as Distributed Denial of Service (DDoS) attack. DDoS attacks fall into two categories - volumetric and non-volumetric. A volumetric attack may orchestrate a huge amount of bots (hacker-owned VMs or hijacked PCs or Internet-of-Things (IoTs) devices) to request and occupy the resources in the system and so deprive nominal users. Examples of volumetric attacks include

- Ping floods, where the attacker floods the victim system with ICMP “echo” (ping) requests, consuming incoming bandwidth; if the system replies to every ping request, the outgoing bandwidth is also occupied by this attack.

- Reflection attacks, where the attack does not send huge traffic to the victim. Instead, the attacker uses publicly accessible services as her “amplifier” by sending queries to them with the target’s IP address as the source (return) IP address. Those services then respond with a larger traffic volume to the target and overwhelm it. Some popular amplifiers that could be leveraged by DDoS attackers, with different amplification factors, are listed in Table 4.

- SYN floods, where attackers send SYN messages to the server to establish a TCP connection, the server then creates a TCP buffers and variables to
the connection, and send SYNACK message back to the attackers informing her connection request was approved. If the attacker, with many spoofed IP addresses, unlike nominal users who also create a buffer and communicate on the newly established connection, leaves the connection half-opened and continues sending TCP connection requests to the server with another spoofed address, then those buffers for the half-opened connections could quickly overwhelm the server’s memory.

The other type of attack is non-volumetric, where attackers leverage a design flaw of applications or protocols in the system so that only relatively small amount of such attack is needed to cripple the system. Examples of non-volumetric attacks include

- Slowloris [82], where the attacker tries to open connections to the server as long and many as possible. Specifically, the attacker opens a HTTP connection to the server, and periodically sends partial HTTP requests to the server but never complete it. As more and more cheap and useless HTTP connections have been established, the server reaches the maximum of concurrent connections and denies additional connections requested by nominal users. In this way, even a single machine can take down a web server.

- BlackNurse [83], where the attacker attacks firewall with the computation-
intensive ICMP Type 3 Code 3 packets. Therefore BlackNurse could take down a server at low bandwidth.

Traditionally, there are two ways of handling DDoS attacks in practice:

**Scaling the system resources to load balance:** In GCE, the frontend infrastructure can automatically scale to absorb certain types of attacks (e.g., SYN floods). If the user has provisioned a sufficient number of instances, the autoscaler can ramp up the backend servers (proxies or replicas) inline with the traffic spike that needs to be handled [84]. In AWS, Amazon CloudWatch alarms are used to initiate the autoscaling on the size of user’s Amazon EC2 fleet in response to the user-defined events [85].

**Detecting and filtering excess traffic:** In MS Azure, there is a non-user-configurable DDoS protection layer in a virtual network of the multiserver system. If an Internet-facing IP address (of a proxy server in our multiserver system) was under DDoS attack, the DDoS protection layer would detect the sources of attacks and scrub the offending traffic [86]. In GCE, anti-spoofing protection is provided for the virtual network by default [84]. In AWS, Elastic Load Balancing (ELB) accepts only well-formed TCP connections to protect users from SYN floods or UDP reflection attacks; Amazon CloudFront can automatically terminate connections from “slow reading-writing” attackers (e.g., Slowloris) [85]. Users can also define the firewall rules of their virtual network to block the source IP addresses that are engaging in the attack or restrict access to some ports [84–86].

In this chapter, we investigate alternative approaches which neither physically scale up the system volume nor first identify the attackers.

### 4.1 Vulnerability and threat model

We now consider the following scenario:

- The distributed system cannot physically scale up by adding more resources;
- A server under attack (an overloaded server) cannot determine which of its assigned client(s) are responsible for this attack;
- It can be reliably determined whether a server is under attack either through the use of: probing “canary” clients that can determine when response
times of their mock workloads have grown too high, heartbeat signals between
proxy/replica servers and monitors, or server and client diagnostic performance
checks that are continually run by many public-cloud providers (e.g., AWS
CloudWatch),

Under those above assumptions, we want to exploit the feasibility and efficiency of
the DDoS defenses which we will introduce in the following sections.

4.2 Pre-attack defense: moving target defense

A complete DDoS attack usually consists of two major phases: reconnaissance
phase and attacking phase.

In the reconnaissance phase, the attackers use a certain amount of bots to
probe and derive the most efficient ways of attacking. For example, in Apache
Cassandra [87], a distributed database system, the attacker may probe the resource
bottleneck by sending requests with a certain range of keys to her coordinator
(proxy). If the coordinator is using the default hash partitioner, then the requests
with the same key will be redirected to the same replica node, hence the attacker
could estimate the processing time of different keys, determine the busiest keys and
launch DDoS attack on that key subset. For those IP addresses and port numbers of
hosts that are designed to be concealed from the Internet but accidentally exposed
to the public due to careless configuration, the attackers can sniff them out by
sending requests to random IP address in the network, the hosts who respond will
be identified and attacked.

In the attack phase, as the name suggests, the attackers launch DDoS attack
against the vulnerabilities collected from reconnaissance phase.

In this section, we investigate a method to sabotage the reconnaissance phase
of DDoS attack - moving target defense (MTD). In MTD, the defender frequently
changes the addresses or identities of vulnerable servers, so that by the time the
DDoS attack is launched, the attacking surface has been changed. Many techniques
can be used to implement MTD under different scenarios. For example, in the
Cassandra case, the dynamic partitioner can be developed to direct requests to
different replicas after a certain interval. Also to change the addresses of vulnerable
hosts without severely disturbing the service, techniques such as Software Defined
Network (SDN) [88,89] and Network Address Translation (NAT) [90] are used in
the literatures. We, instead of focusing on any specific MTD techniques, want to derive a general mathematical model to evaluate the efficiency of MTD under different parameters.

In Sec. 4.2.1, we consider the process that DDoS attacker collects the identity of vulnerable servers as a coupon collection model, in which we are able to derive the stationary mean number of servers exposed to the attacker (mean number of different coupon types possessed by the collector) in different defense scenarios (different models of coupon acquisition and purge).

### 4.2.1 Coupon collection model

Coupon collection is a classical problem of probability theory, e.g., [91]. Suppose there are a number of different coupon types and the aim is to obtain as many coupon types as possible. Coupons may be requested sequentially. The types of coupons obtained through each request is random.

In this section we consider the scenario where the defender protects its vulnerable servers by periodically changing their identities (e.g., addresses or roles (the kinds of requests they will serve)) so that the botnet is unable to collect sufficient targets to launch its attack efficiently. This defense process is equivalent to a special coupon collection model wherein the coupon types (server identities) are periodically reset. In Table 4.2.1, we give the correspondence between DDoS defense and the coupon collector model. The defender can change the server identities in two fashions: one is changing all the identities at the same time; the other is randomly selecting a server to change its identity. On the other side, the coupon acquisition interval can be modeled as a constant or a random variable in exponential distribution. In the following, for those different identity collecting and changing fashions, under the assumption that the times between coupon type (server identity) changing event are independent and (memorylessly) exponentially distributed (This indicates that the coupon changing process is Poisson. For more discussion on properties of Poisson
process, see Appendix C), we develop the corresponding continuous-time models using mathematical tools such as Stirling number of the second kind, Poisson process, etc.

4.2.2 Periodically replacing all coupon types

4.2.2.1 Deterministic coupon acquisition with Poisson coupon-type replacements

Let \( m \) be the number of coupon types. Suppose the attacker’s coupon acquisition (target probing) happens at a constant rate and all coupon types are changed every exponentially distributed interval (Poisson coupon-type replacements). We have the following proposition.

**Proposition 3.** Under constant rate coupon acquisition (constant rate \( \beta \)), and Poisson coupon-changing process (mean rate \( \delta \)) determining when all coupon types are replaced, the stationary mean number of currently valid, different coupon types obtained is

\[
EY = \frac{1}{e^{1/\rho} - (1 - \frac{1}{m})},
\]

where \( \rho = \frac{\beta}{\delta} \).

**Proof.** At constant \( \beta \) coupon acquisitions/s, the number of coupons selected over \([0, T]\) is simply

\[K = \beta T.\]

Since coupon replacement is Poisson, \( T \) is a exponential random variable with mean \( 1/\delta \). It is easy to know \( K \) is a geometric random variable with the following distribution,

\[P(K = k) = P(k/\beta \leq T < (k + 1)/\beta) = e^{-k/\rho} \left(1 - e^{-1/\rho}\right),\]

for \( k = 0, 1, 2, \ldots \), with probability generating function

\[Ez^K = \frac{1 - e^{-1/\rho}}{1 - ze^{-1/\rho}}.\]
So,

\[ EY = Em \left( 1 - \left( 1 - \frac{1}{m} \right)^K \right) = \frac{e^{-1/\rho}}{1 - (1 - \frac{1}{m})e^{-1/\rho}} \]

which is 4.1. Since Poisson Arrivals See Time Averages (PASTA) [92] (See Appendix D for more details), this is also the expected number of different types of coupons collected at a typical time.

\[ \square \]

### 4.2.2.2 Poisson coupon acquisitions with independent Poisson coupon-type replacement process

Now suppose all-coupon-changing process is still Poisson, and attacker’s coupon acquisition is now modeled as a Poisson process, we have the following proposition

**Proposition 4.** Let \( m \) be the number of coupon types. Under Poisson coupon acquisition (rate \( \beta \)), Poisson coupon-changing process (rate \( \delta \)), the stationary mean number of currently valid, different coupon types obtained is

\[ EY = \frac{m \rho}{m + \rho}. \]  \hspace{1cm} (4.2)

**Proof.** After \( K \) coupon acquisitions, for coupon type \( i \), define an indicator \( I_i \) such that

\[ I_i = \begin{cases} 1 & \text{If the coupon type was selected;} \\ 0 & \text{Otherwise.} \end{cases} \]

So the mean number of coupon types collected just before coupon types are replaced is

\[ E \sum_{i=1}^{m} I_i = Em(1 - (1 - 1/m)^K). \]

Conditioning on \( T \), \( K \sim Poisson(\beta T) \). So, the mean number of different coupons obtained conditioned on \( T \) is simply

\[ \sum_{k=0}^{\infty} m(1 - (1 - 1/m)^k) \frac{(\beta T)^k e^{-\beta T}}{k!} = m(1 - e^{-\beta T/m}). \]  \hspace{1cm} (4.3)

Using the moment generating function of \( T \) \( \exp(\delta) \), we get

\[ Em(1 - (1 - 1/m)^K) = Em(1 - e^{-\beta T/m}) \]

66
which is 4.2. Finally, since PASTA, this also the expected number of different coupons collected at a typical time.

Note that 4.1 and 4.2 are close when \( \rho \to \infty \) since

\[
\lim_{\rho \to \infty} e^{1/\rho} - (1 + 1/\rho) = 0.
\]

We also derive the stationary distribution on the number of coupon types collected by the attacker in the model as follows.

**Proposition 5.** Under Poisson coupon-acquisition (rate \( \beta \)) and Poisson coupon-changing process (rate \( \delta \)), in steady state,

\[
P(Y = l) = \frac{1}{\rho \rho + 1} \frac{m}{m - l} \frac{m - 1}{m^{\rho + 1}/\rho - 1} \cdots \frac{m - l}{m^{\rho + 1}/\rho - l},
\]

for \( 1 \leq l \leq m \).

**Proof.** Again, \( K \) given \( T \) is \( \sim \text{Poisson}(\beta T) \). The unconditional distribution of \( K \) is geometric:

\[
P(K(T) = k) = \int_0^\infty \delta e^{-\delta t} (\beta t)^k e^{-\beta t} \frac{k!}{k!} dt
\]

\[
= \left( \frac{\rho}{\rho + 1} \right)^k \frac{1}{\rho + 1}, \quad k \geq 0
\]

(4.4)

Thus, for \( l \in \{0, 1, \ldots, m\} \),

\[
P(Y(K(T)) = l) = \sum_{k=0}^\infty P(Y(k) = l) \left( \frac{\rho}{\rho + 1} \right)^k \frac{1}{\rho + 1}
\]

\[
= \sum_{k=0}^\infty (m)_l \{ k \} \left( \frac{\rho}{\rho + 1} \right)^k \frac{1}{\rho + 1}
\]

\[
= \frac{(m)_l}{\rho + 1} \frac{\rho + 1}{\rho/m} \sum_{k=0}^\infty \{ k \} \left( \frac{\rho/m}{\rho + 1} \right)^{k+1}
\]

\[
= \frac{m}{\rho} (m)_l \frac{1}{(\rho/m)_{l+1}}
\]
where the last equality uses the conclusion from (1.94c) on p. 82 of [93],

\[
\sum_{k=0}^{\infty} x^{k+1} = \frac{1}{(1/x)^{t+1}},
\]

where \((a)_t := a(a-1)(a-2)\ldots(a-l+1)\).

Again since PASTA, this is the steady-state distribution of the number of coupon types obtained. \(\square\)

### 4.2.2.3 Discussion: renewal coupon acquisitions, Poisson coupon-type replacements

Suppose that the coupon acquisition process \(K(t)\) is a renewal counting process\(^1\) with interarrival times \(X\) of finite variance, \(\sigma^2 < \infty\), and mean \(1/\beta\). The classical central limit theorem for renewal process is (e.g., [94]),

\[
\lim_{t \to \infty} P\left(\frac{K(t) - \beta t}{\sqrt{\sigma^2 \beta^3 t}} < y\right) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y} e^{-z^2/2} dz,
\]

i.e., \(K(t) \sim N(\beta t, \sigma^2 \beta^3 t)\) as \(t \to \infty\). Let \(v = -\log(1 - 1/m) > 0\). Using the moment-generating function of a Gaussian, we can approximate the expected number of different coupon types collected just before they are replaced as:

\[
\text{Em} \left(1 - (1 - 1/m)^{K(t)}\right) \sim \text{Em} \left(1 - e^{-\beta T v + \sigma^2 \beta^3 T v^2/2}\right)
\]

\[
= m \left(1 - \frac{\delta}{\delta + \beta v - \sigma^2 \beta^3 v^2/2}\right)
\]

requiring \(\delta + \beta v - \sigma^2 \beta^3 v^2/2 > \delta\), i.e., \(\beta^{-1} > \sigma \sqrt{v}/2\). Again since PASTA, this is the stationary number of different coupon-types collected. When \(\sigma = 0\), this limit agrees with the deterministic coupon acquisition 4.1 of Prop. 3 when \(m\) and \(\rho\) are large since \(v \sim 1/m\) and \(e^{1/\rho} \sim 1 + 1/\rho\).

\(^1\)Poisson process is one special case of renewal counting process
4.2.3 Individual coupon type random replacement, according to a single Poisson process

We still model the coupon type replacement event as a Poisson process, but instead of replacing all the coupon types in a single event, as we did in last section, the coupon types are independently replaced at random. This is tricky since the coupon types survived in the current replacement would pass down to the next coupon replacement.

Consider the discrete-time Markov chain $Y(k)$ satisfying

$$Y(k) = Y(k-1) + \xi_k, \; k \geq 1,$$

where $Y(0) \in \{0, 1, 2, \ldots, m\}$ (i.e., $Y(0)$ is not necessary zero) and

$$P(\xi_k = 0 | Y(k-1), \ldots, Y(0)) = \frac{Y(k-1)}{m},$$
$$P(\xi_k = 1 | Y(k-1), \ldots, Y(0)) = 1 - \frac{Y(k-1)}{m}.$$

Here, $Y(k)$ is the number of different types of coupons collected after the $k^{th}$ coupon acquisition including those initially had, $Y(0)$.

4.2.3.1 Proof of Prop. 4 when $Y(0) \equiv 0$

Note that if $Y(0) \equiv 0$, then the problem reduces to all coupon type replacement as we discuss in Sec. 4.2.2. So we can alternatively verify 4.2 by simply solving the recursion for $EY(k)$ when $Y(0) = 0$:

$$EY(k) = 1 + (1 + \frac{1}{m}) + (1 - \frac{1}{m})^2 + \cdots + (1 - \frac{1}{m})^{k-1}$$
$$= m \left( 1 - \left( 1 - \frac{1}{m} \right)^k \right). \quad (4.6)$$

Now let us collect coupons at the ticks of a Poisson process $K(t)$, $t \geq 0$, with rate $\beta$ and independent coupon types selected independently and uniformly. Thus, at real time $t \geq 0$, we have collected

$$Y(K(t))$$

different types of coupons.
So,
\[ E(Y(K(t))) = \sum_{k=0}^{\infty} m \left( 1 - \left(1 - \frac{1}{m}\right)^k \right) \mathbb{P}(K(t) = k) \]
\[ = m \left( 1 - e^{-\beta t/m} \right). \]

So if \( t = T \) is exponentially distributed with rate \( \delta \), we have
\[ E(Y(K(t))) = \int_0^{\infty} m(1 - e^{-\beta t/m})\delta e^{-\delta t} dt = \frac{m \rho}{m + \rho}, \]
(4.7)
same as Eq. 4.2 in Prop. 4.

4.2.3.2 Independent coupon-type replacement events, \( Y(0) \neq 0 \)

A generalization of 4.2 (\( Y(0) \) is not necessarily equal to 0) is:

**Proposition 6.** For Poisson coupon acquisition (at rate \( \beta \)), Poisson coupon-type replacements (at rate \( \delta \)), and individual coupon types are independently replaced with probability \( r \), the mean number of valid coupon types obtained in steady state is:

\[ \frac{m \rho}{rm + \rho} \]  
(4.8)

**Proof.** For \( Y(0) \neq 0 \), solving the recursion

\[ EY(k) = EY(k - 1) + E\xi_k = 1 + (1 - 1/m)EY(k - 1) \]
gives, for \( k \geq 1 \),

\[ EY(k) = 1 + (1 - 1/m) + \cdots + (1 - 1/m)^{k-1} + (1 - 1/m)^k EY(0) \]
\[ = m(1 - (1 - 1/m)^k) + (1 - 1/m)^k EY(0). \]
(4.9)

As above, since \( K \sim \text{Poisson}(\beta T) \) and \( T \sim \text{exp}(\delta) \),

\[ EY(K(T)) = \frac{m \rho}{m + \rho} + \frac{m}{m + \rho} EY(0). \]
(4.10)
In steady-state,

\[(1 - r)E(K(T)) = EY(0). \tag{4.11}\]

Eliminating \(E(0)\) from Eq. 4.10 and 4.11 gives \(E(K(T))\) equals Eq. 4.8. The proposition follows since PASTA.

\[\square\]

4.3 Post-attack defense: shuffle and fission

In this section, we focus on the mitigation strategies after attackers have successfully launched a non-volumetric DDoS attack. Two novel defense types are introduced - shuffle and fission. Those defenses take effect without identifying the attacking bots. We explore mathematical tools to analyze the cost and efficiency of those defenses and present the effects of those defenses with those tools.

4.3.1 Shuffle defense

Suppose all clients using servers that are detected to be overloaded could be periodically reassigned to (shuffled among) the attacked servers at random. By change, some overloaded proxies are assigned only normal clients and those clients have thus been effectively “liberated”. This server-level overloaded/attacked detection and shuffling among those overloaded/attacked servers are repeated until attacking/heaving-hitting clients are sufficiently quarantined. An example of shuffle defense is illustrated in Fig. 4.1.

![Shuffle Defense Diagram](image)

**Figure 4.1.** An example of shuffle defense. Note that in this particular case 75% nominal clients will be liberated after one shuffle step.
4.3.1.1 Modeling shuffle defense

Suppose a single attacker can take down a server, i.e., the server is deemed under attack when an attacker is assigned to it. Stirling number of the second kind can be used to analyze the efficiency of shuffling defense. Let $K$ be the number of attackers, $M$ be the number of servers in our system. The number of ways that $K$ distinct attackers can be assigned to exactly $R \leq K \wedge M := \min\{K, M\}$ distinct servers (i.e., such that exactly $R$ servers have at least one assigned attacker and $M - R$ servers have no assigned attackers) is

$$
\frac{M!}{(M-R)!} \binom{K}{R} \text{ where } \binom{K}{R} = \frac{1}{R!} \sum_{j=0}^{R} (-1)^{R-j} \binom{R}{j} j^K
$$

is a Stirling number of the second kind. Suppose attackers are assigned to servers in equally likely fashion, the probability that $K$ distinct attackers are assigned to exactly $R$ of $M$ distinct servers is

$$
P_K(M, R) := \frac{M!}{(M-R)!} \binom{K}{R} \text{ where } M^K = \sum_{R=1}^{\min\{K, M\}} \frac{M!}{(M-R)!} \binom{K}{R}.
$$

The situation becomes more complicated if a server can tolerate more than one attacker. Let $S_{K,n}(M, R_n)$ be the number of ways $K$ distinct attackers can be assigned to $M$ distinct servers so that exactly $R_n \geq 1$ of the servers are assigned at least $n$ attackers. Also let $U_j = \sum_{i=1}^{j} R_j$ and $U_0 = 0$. By first deciding which servers are assigned at least one attacker, and then restricting assignment of the remaining attackers to these servers, we get the following expression

$$
S_{K,n}(M, R_n) = \sum_{R_n = 1}^{K-U_n + K-nR_{n-1}} \cdots \sum_{R_{n+2} = 1}^{K-U_{n+1} + R_{n+1}} \sum_{R_{n+1} = 1}^{K-U_n \wedge R_{n-1}} \sum_{R_{n-1} = R_n}^{K-2R_n-U_{n-3}} \sum_{R_{n-2} = R_n}^{K-(n-2)R_{n-2}-U_1} \sum_{R_{n-3} = R_n}^{K-(n-3)R_{n-3}-U_2} \cdots \sum_{R_2 = R_n}^{K-2R_n-U_{n-3}} \sum_{R_1 = R_n}^{K-(n-1)R_{n-1}-U_0} \sum_{R_0 = R_n}^{K-2R_n-U_{n-3}} \\
\prod_{j=1}^{n+K-nR_n} \frac{(K-U_{j-1})!}{(K-U_{j-1}-R_j)!} \binom{R_{j-1}}{R_j} / (j!)^{R_j - R_{j+1}}, \tag{4.12}
$$

72
Associated probabilities are

\[ P_{K,n}(M, R) = \frac{S_{K,n}(M, R)}{M^K}. \]

Alternatively, a recursive algorithm using r-associated Stirling number to calculate the same value is given in [95].

Fig. 4.2 is an indication of how fewer servers are “under attack” when they can tolerate two attacking bots compared to just one.

![Figure 4.2.](image)

Figure 4.2. \( P_{20,1}(10, R) \) and \( P_{20,2}(10, R) \) versus \( R \).

Although Stirling number, Eq. 4.12, and r-associated algorithm could yield accurate results, the large number (especially Stirling number of the second kind \(^2\)) and superpolynomial property involved (Eq. 4.2) make these approaches numerically formidable when dealing with a very large system.

When the number of servers, \( M \), is large and a fixed number of clients per server is assumed, a binomial distribution model can be used to simplify the problem. In the following, let

- \( M \) be the number of servers
- \( U = Eu \) be the mean number of nominal users
- \( K = E\kappa \) be the mean number of attacker users.

\(^2\)Some asymptotic approximations can be used to partially solve this problem. [96]
Let $a(i)$ be the probability that a server has $i$ attacking/heavy-hitter clients. With the binomial approximation, $a(i)$ can be expressed as

$$a(i) = \text{binom}(v, p)$$

where $v = \frac{U + K}{M}$, $p = \frac{K}{U + K}$

i.e.,

$$a(i) = \binom{v}{i} p^i (1 - p)^{v-i} \text{ for } i \in \{0, 1, 2, \ldots, v\}. \quad (4.13)$$

Note that $K = E\kappa = Mvp$ and $U = Eu = Mv(1 - p)$. Also note that the standard deviations $\sigma(\kappa) = \sqrt{Mvp(1 - p)} = \sigma(u)$ so that $\sigma(\kappa)/E\kappa$ and $\sigma(u)/Eu$ are small for large $M$, i.e., $u \approx U$ and $\kappa \approx K$ with high probability. Let $A \in \{1, 2, \ldots, v\}$ be the number of attacking clients required to overload the server. So, the probability that a server is overloaded is taken to be

$$\Omega := \sum_{i=A}^{v} a(i).$$

Consider a shuffling strategy that excludes the clean server from the next shuffling\(^3\), the expected number of freed clients after $n$ shuffling can be recursively expressed as

$$L_n(K, U, M) = L_0(K, U, M) + \sum_{n=Mp}^{M\wedge K} B(n)L_{n-1}(K, vn - K, n). \quad (4.14)$$

where

$$L_0(K, U, M) = M \sum_{i=0}^{A} a(i)v.$$

The first term in Eq. 4.14 (i.e., $L_0(K, U, M)$) is the number of nominal clients saved on the initial assignment, without purposely shuffling.

Suppose an single attacker can take down a server. The mean fraction of nominal attackers under attack after single shuffle stage is shown in Fig. 4.3.

\(^3\)An alternative strategy would be, as mentioned above, moving all the clients on the clean server to a safe zone, and recycle this server in the next shuffle stage. The problem of approach is that if a server can tolerate more than one attacker, then some attackers might be moved to the safe zone as well.
4.3.2 Fission defense

Under $m$-ary fission defense, $m - 1$ containers are spun-up for each container housing a server detected under attack (and this typically on the same physical server or the same VM). The clients assigned to the attacked server are equally divided among the $m$ resulting containers, and attack detection is repeated. Since the resources (both physical and logical) are isolated among the containers, fission may allow the defender free part of the nominal clients from DDoS and quarantine attackers in a few containers. Containers housing only nominal clients (i.e., containers deemed not under attack) may be consolidated to reduce the number of containers in play. Fission and detection are repeated until the attacking clients are sufficiently quarantined. An example of fission defense is illustrated in Fig. 4.4.

Note that fission may be feasible only under a main assumption in Sec. 4.1 - low-volume attack. Under high volume attack, where the physical bottleneck resources (e.g., network bandwidth) are illegally occupied or a single server is attacked by too many bots, either spawning containers are completely useless or fission defense would be less efficient and more costly as we will discuss in the following sections.
4.3.2.1 Case study 1: SlowLoris attack

SlowLoris [82] attack is a low volume attack that depletes the victim web server’s concurrent connection pool by sending partial HTTP requests and keep them open as long as possible. Slowloris attack targets logical resources (HTTP connections) and does not need large bandwidth to succeed, therefore, it is such an efficient attack that even a single computer can bring down a web server.

Suppose only there is a client that launched SlowLoris attack, after we fission the container that holds the web server into two identical containers and redirect half clients to the new container, then the service for the client in the container without the attacker will not be affected. This explanation can be illustrated in Fig. 4.5.

4.3.2.2 Case study 2: BlackNurse attack

BlackNurse [83] attacks firewall with ICMP Type 3 Code 3 packets. Processing such message would consume significant computing resources on some firewalls. Therefore BlackNurse could be highly effective even at low bandwidth. It has been reported that even a single laptop can take down servers.
Fission can also handle this low-volume attack on physical resources due to the isolation nature of containers. Under BlackNurse attack from a client, if the service hosting container is fissioned to two and the attacking client is assigned to one of the containers, then although the container with the attacker can be taken down more easily, the other container will not be affected since its computation resources are strictly isolated from the attack. This is illustrated in Fig. 4.6.

4.3.2.3 Modeling fission defense

In this section, we want to discuss the efficiency of fission defense under different environmental variables. Let

- $u$ be the number of nominal users in a container;
- $k$ be the number of attackers in a container;
- \( m \) be the number of containers after one fission is applied to a container, i.e., \( m \)-ary fission.

Consider binary fission, after one fission step, there will be two servers, each with \((u + k)/2\) clients. The probability that \( l \) attackers have been assigned to the first container is

\[
\phi(l | k; u) = \binom{k}{l} \left( \frac{u}{u + k} \right) \left( \frac{u + k}{2} \right) / \left( \frac{u + k}{2} \right),
\]

where \( \binom{b}{c} = 0 \) if \( c < 0 \) or \( c > b \).

Let \( \Lambda_n(k_0; u_0) \) be the mean number of nominal users liberated after \( n \) binary fission steps given initial \( k_0 \) attackers and \( u_0 \) nominal users. After one binary fission step, the number of liberated nominal users is

\[
\Lambda_1(k_0; u_0) = \sum_{k_1=0}^{k_0} \phi(k_1 | k_0; u_0) \left( 1\{k_1 = 0\} \frac{k_0 + u_0}{2} + 1\{k_1 = k_0\} \frac{k_0 + u_0}{2} \right),
\]

(4.15)

where \( 1\{\text{Cond}\} = 1 \) if \( \text{Cond} \) is true, \( 0 \) otherwise. With Eq. 4.15, \( \Lambda_n(k_0; u_0) \) for \( n \geq 2 \) can be recursively expressed as

\[
\Lambda_n(k_0; u_0) = \sum_{k_1=0}^{k_0} \phi(k_1 | k_0; u_0) \left( \Lambda_{n-1}(k_1; \frac{k_0 + u_0}{2} - k_1) + \Lambda_{n-1}(k_0 - k_1; k_1 - \frac{k_0 - u_0}{2}) \right),
\]

(4.16)

where \( \Lambda_n(k; 0) = 0 \) and \( \Lambda_n(0; x) = x \).

We use the total containers created through the fission steps as the cost of fission defense. Let \( C_n \) be the cumulative number of containers used after \( n \) binary fission steps including the initial containers before fissioning.

After one binary fission step,

\[
C_1(k_0, u_0) = \begin{cases} 
1 & \text{if } k_0 = 0 \text{ or } k_0 + u_0 = 1 \\
2 & \text{otherwise}
\end{cases}
\]

(4.17)

After \( n > 1 \) binary fissions: if \( k = 0 \) or \( k + u = 1 \), then \( C_n(k, u) = 1 \), else:

\[
C_n(k_0, u_0) = \sum_{k_1=0}^{k_0} \phi(k_1 | k_0; u_0) \left( C_{n-1}(k_1; \frac{k_0 + u_0}{2} - k_1) + C_{n-1}(k_0 - k_1; k_1 - \frac{k_0 - u_0}{2}) \right)
\]

(4.18)

This framework is easily generalized to \( m \)-ary fissions for \( m > 2 \), where larger \( m \) results in faster quarantine (fewer fission/detection stages) but overall more
containers may need to be simultaneously mobilized, i.e., more overhead. Obviously, if \( m = k + u \) (all clients are individually containerized), then only one fission step is required.

Now consider a system with \( M \) servers and \( U \) nominal users, which is under attack from \( K \) bots. To apply the above fission analysis onto this multi-server system, we need to know the probability of how many attackers initially assigned to a specific server \( (k_0) \). We can get the probabilities by modifying the recursive expression in ?. However, that calculation involving Sterling number is prohibitively expensive. Suppose those \( K \) attackers, together with \( M \) nominal users, are randomly assigned to \( M \) servers. Note that for the same reason as we discussed in 4.3.1, the binomial model can be a good candidate for analyzing this scenario.

With binomial model, the probability that a client is an attack is

\[
p = \frac{K}{U + K}
\]

So, \( a(i) \), the probability that a server has \( i \) attacker is

\[
a(i) = \binom{v}{i} p^i (1 - p)^{v-i} \text{ for } i \in \{0, 1, 2, ..., v\}, \tag{4.19}
\]

where \( v = (U + K)/M \) is the number of clients (both nominal and attacking) assigned to a server.

Combining Eq. 4.15 to 4.18, we obtain the expected number of liberated and containers spawned after \( n \) binary fission steps in the \( M \)-server system expressed in Eq. 4.20 and 4.21 respectively.

\[
M \sum_{k_0=1}^{v} a(k_0) \Lambda_n(k_0; v) \tag{4.20}
\]

\[
M \sum_{k_0=1}^{v} a(k_0) C_n(k_0, v) \tag{4.21}
\]

Suppose a system with 1000 servers and 50000 nominal users, Fig. 4.7 depicts the mean fraction of nominal users free from attack and Fig. 4.8 shows the number of containers created as binary fissioning goes on.
4.4 Conclusions

In this chapter, we investigated cloud-side methods to address DDoS attacks targeting cloud resource without using traditional strategies which typically involve detecting, blacklisting and filtering within the Internet. We identified two defense categories - pre-attack (proactive) defense and post-attack (reactive) defense.

Specifically, we studied proactive Moving Target Defense (MTD), which under-
mines the attacker’s processing of gathering vulnerable targets by intermittently changing targets’ identities. We modeled attacker’s target-collecting process as a coupon collection and replacement process. Using this model, we can analyze the efficiency of MTD in preventing enough targets to launch an effective attack.

For post-attack defense, we study two novel attack mitigating approaches. One is shuffling, which continuously and randomly shuffles the users, including both nominal users and attacking bots, among attacked servers. Nominal users are liberated when allocated to a attacker-free server. The other is fissioning, which leverages container technology to divide one virtual server into multiple virtual servers and do attack detection on the smaller/“fissioned” servers. In this way we can identify nominal users in the attack-free servers and consolidate them to the safe zone. The performance of these two approaches were analyzed using Stirling numbers and binomial models.
This chapter highlights a few challenging future research directions.

5.1 Task orchestration and scheduling

In this thesis, we showed that heterogeneously sized macrotasking outperforms homogeneous microtasking when we can accurately estimate the processing speed of our executors. In a stable environment, we can estimate the processing speed accurately with simple model, for example, by using the amount of resources allocated to (reserved for) a container. In reality, however, the execution environment may have variation that will affect execution speed, e.g., due to interference from other processes in CPU scheduling, memory bandwidth, data placement, disk/network bandwidth, caching, etc. In Sec. 2.3.2.1, we used monitoring data collected from previous workloads or benchmark to adaptively change the workload partitioning scheme. If we have a model involving those above factors to accurately estimate the processing speed of our executors, then our heterogeneous macrotasking may achieve good synchronization performance even without initial runtime trials to probe the conditions of the executors. So in the future, developing such a model would be an interesting research direction.

5.2 Cluster resource scheduling

In Chapter 3, we discussed fairness in resource scheduling in private clouds. We empirically evaluated the trade-off between fairness and allocation efficiency for
different max-min fairness criteria, especially in a dynamic environment where users and resources randomly join and leave the system and progressive filling is a widely used approximation method. Alternatively, the trade-off between fairness and efficiency can be formulated as a multi-objective optimization problem. The properties of the solutions of such problem might be interesting to investigate.

Besides max-min fairness, we also discussed proportional fairness. The relationship between max-min fairness and proportional fairness is worth further investigation. Moreover, to apply proportional fairness in practice, an online approximation method similar to progressive filling in max-min fairness needs to be developed.

Lastly, it is possible to bridge private cloud scheduling and public cloud scheduling if we consider the allocation priority $\phi$ in Eqs. 3.3 and 3.11 as tenant bids for virtual machines or containers with fixed resource allocations $d$ in the public cloud. Given tenant utilities and a model of cloud revenue, a game-theoretic model could be set up and its equilibrium could inform cloud pricing.

### 5.3 Defense against resource abuse

In [97], we verified the validity of Moving Target Defense (MTD) against the Distributed Denial-of-Service (DDoS) attacks for cloud-based Web (HTTP) services. However, a broader survey and prototypes are needed to check the feasibility of MTD in more general scenarios such as database, game/video/music streaming services, etc. Moreover, the costs of MTD needs to be more carefully evaluated, especially for “stateful” services. For example, in multi-player games, migrating services to a new machine may cause degradation of video quality and desynchronize the actions of different players.

In post-attack defenses (Sec. 4.3), in most cases, we assume non-volumetric attacks where even a single bot can take down a server. However, in practice, an attacker may need plural bots to cripple a server. In future work, it is worthwhile to explore how a server may have a range of load state (not just overloaded or not overloaded). The defense objective could be to maintain servers remain in an acceptable load range. Such an objective requires a more complex system model to accurately analyze the efficiency of our post-attack defense.

In the defense of Sec. 4.3.1, clients assignments to servers were randomly
shuffled. Shuffling could be more efficient when combined with client histories. For example, a reputation system could be used. The reputation of a client is downgraded by association with a overloaded server. Certainly a nominal client’s reputation may be downgraded, but as we keep shuffling, the expected reputation of an attacker is lower than that of a nominal user. Reputation-based shuffling could more quickly cluster clients, leading to white-listing clients with high reputations and black-listing clients with low reputations.
Appendix A
Proof of Proposition 1

Proof. Rewrite the constraints 3.7 as

\[ \forall i, r, \sum_n x_{n,i} B_{n,i,r} \leq 1, \text{ where } B_{n,i,r} := \frac{d_{n,r}}{c_{i,r}}. \]

Also consider the implicit constraints \( \forall n, i, x_{n,i} \geq 0 \). For the optimization problem in Proposition 1, we have the Lagrangian to be maximized over \( x \) and over Lagrange multipliers \( \lambda, \nu \geq 0 \):

\[ L = \sum_n \phi_n g(U_n) + \sum_{i,r} \lambda_{i,r} \left( 1 - \sum_n x_{n,i} B_{n,i,r} \right) + \sum_{i,n} \nu_{n,i} x_{n,i}. \]

The first-order optimality condition (recall the definition of \( U_n \) in 3.3),

\[ \forall i, n, 0 = \frac{\partial L}{\partial x_{n,i}} = u_{n,i} g'(U_n) - \sum_r \lambda_{i,r} B_{n,i,r} + \nu_{n,i}, \quad (A.1) \]

and \( g \) strictly increasing imply

\[ \forall i, n, \sum_r \lambda_{i,r} B_{n,i,r} > \nu_{n,i} \geq 0. \quad (A.2) \]

So, \( \forall i, \exists r, \text{ s.t., } \lambda_{i,r} > 0 \). Thus, complementary slackness is

\[ \forall i, r, \lambda_{i,r} \left( 1 - \sum_n x_{n,i} B_{n,i,r} \right) = 0 \]

\[ \Rightarrow \forall i, \exists r \text{ s.t., } \sum_n x_{n,i} B_{n,i,r} = 1, \quad (A.3) \]
i.e., in every server $i$, one resource $r$ (which may depend on $i$) is fully booked. So, the set of fully booked resources in server $i$ under allocations $x = \{x_{n,i}\}$ can be characterized by $\{r|\lambda_{i,r} > 0\}$. Now by A.1 and assume strict concavity of $g$, uniquely

$$\forall i, n, U_n = (g')^{-1}\left(\sum_r \lambda_{i,r} \frac{B_{n,i,r}}{u_{n,i}} - \frac{\nu_{n,i}}{u_{n,i}}\right) = (g')^{-1}\left(\sum_{r: \lambda_{i,r} > 0} \lambda_{i,r} \frac{B_{n,i,r}}{u_{n,i}} - \frac{\nu_{n,i}}{u_{n,i}}\right).$$

Now consider two frameworks $m$ and $l$ and server $i$ such that $x_{m,i} > 0$ and . So, complementary slackness

$$\forall i, n, \nu_{n,i} x_{n,i} = 0,$$  \quad (A.4)

implies $\nu_{m,i} = 0$. Thus, because $(g')^{-1}$ is strictly decreasing ($g$ strictly concave),

$$U_m = (g')^{-1}\left(\sum_{r: \lambda_{i,r} > 0} \lambda_{i,r} \frac{B_{m,i,r}}{u_{m,i}}\right) \leq (g')^{-1}\left(\sum_{r: \lambda_{i,r} > 0} \lambda_{i,r} \frac{B_{l,i,r}}{u_{l,i}} - \frac{\nu_{l,i}}{u_{l,i}}\right) = U_l,$$  \quad (A.5)

where we have used assumption 3.10 for the inequality. Because of this and A.3, a solution $x = \{x_{n,i}\}$ of the optimization (3.9) s.t. (3.7) is U-MMF. \hfill \Box
Appendix B
Proof of Proposition 2

Rewrite the constraints 3.7 as
\[\forall i, r, \sum_n x_{n,i} B_{n,i,r} \leq 1, \text{ where } B_{n,i,r} := \frac{d_{n,r}}{c_{i,r}}.\] (B.1)

Also consider the implicit constraints \(\forall n, i, x_{n,i} \geq 0\), the Lagrangian of the optimization problem in Proposition 2 is

\[L = \sum_n \phi_n g_a(x_n) + \sum_{i,r} \lambda_{i,r} (1 - \sum_n x_{n,i} B_{n,i,r}) + \sum_{i,n} \nu_{n,i} x_{n,i}\]

where the Lagrange multipliers \(\lambda, \nu \geq 0\). A first-order optimality condition is

\[\forall i, n, 0 = \frac{\partial L}{\partial x_{n,i}}(x^*) = \phi_n g'_a(x^*_n) - \sum_r \lambda_{i,r} B_{n,i,r} + \nu_{n,i}.\] (B.2)

Multiplying B.2 by \(x_{n,i} - x^*_{n,i}\) and summing over \(i\) and \(n\) gives\(^1\)

\[0 = \sum_n \phi_n g'_a(x^*_n) (x_n - x^*_n) + \sum_{i,n} \nu_{n,i} (x_{n,i} - x^*_{n,i}) - \sum_{i,r} \sum_n \lambda_{i,r} B_{n,i,r} (x_{n,i} - x^*_{n,i})\]

where the first term is \(\Phi(x, x^*)\). Thus, note that \(\lambda_{i,r} \neq 0\), \(\forall i, r\) s.t., \(\sum_n x^*_{n,i} B_{n,i,r} = 1\), by complementary slackness of A.3 and A.4 in Appendix A,

\[\Phi(x, x^*) = \sum_{i,r \in \{i,r : \lambda_{i,r} \neq 0\}} \lambda_{i,r} \left(\sum_n x_{n,i} B_{n,i,r} - 1\right) - \sum_{i,n} \nu_{n,i} x_{n,i}.\]

\(^1\)Simply use Fubini’s theorem for the first term, \(\sum_i \sum_n \phi_n g'_a(x^*_n) (x_{n,i} - x^*_{n,i}) = \sum_n \sum_i \phi_n g'_a(x^*_n) (x_{n,i} - x^*_{n,i})\).
Finally, no resource overbooking, i.e., all $x$ are constrained by B.1, implies $\Phi(x, x^*) \leq 0$. 
Appendix C
Properties of Poisson Processes

Poisson process is a classic stochastic which randomly place points along a mathematical space which is widely used in queuing theory due to its mathematical conveniences such as formula simplicity and memorylessness. It is often defined on the real line as follows

Definition 2 (Poisson process). One real line, a counting process \( \{N(t), t \in [0, \infty)\} \) is called Poisson with parameter \( \lambda \) if all the following conditions hold:

- \( N(0) = 0 \);
- \( N(t) \) has independent increment, i.e., for any two intervals \((a_1, b_1]\) and \((a_2, b_2]\), \(N(b_1) - N(a_1) \) and \(N(b_2) - N(a_2)\) are independent;
- The following equation holds for any \( t > 0 \),

\[
P\{N(t) = n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t}. \tag{C.1}
\]

Equivalently, Poisson process can also be defined by the distribution of distance between two consecutive points as follows

Definition 3 (Poisson process redefinition). If in a stochastic process, the distance between two consecutive points are independent and exponentially distributed, then this process is Poisson.

The proof of equivalence between the above two definitions is given in the following.
Proof. First prove the condition in Definition 2 leads to the one in 3. Let \( \{t_i, i \geq 1\} \) be the position of Poisson points on the time line, \( N(t) \) be the count of Poisson points in \((0, t]\) and \( \{X_i, i \geq 1\} \) be the intervals between consecutive Poisson points. Note that \( X_1 = t_1 \) is the interval between the first Poisson point and time 0. We have the following probability

\[
P\{X_n > x | X_{n-1} = x_{n-1}, \ldots, X_2 = x_2, X_1 = x_1\} = P\{N(t_{n-1} + x) - N(t_{n-1}) = 0 | N(t_{n-1}) - N(t_{n-2}) = 1, \ldots, N(t_1) = 1\}.
\]

(C.2)

Because of the non-overlapping-intervals-independence rule in 2, Eq. C.2 equals

\[
P\{N(t_{n-1} + x) - N(t_{n-1}) = 0\} = P\{X_n > x\} = P\{N(x) = 0\} = e^{-\lambda x}
\]

so

\[
P\{X_n < x\} = 1 - e^{-\lambda x} \tag{C.3}
\]

Take derivative on \( x \) in C.3, we get

\[f_{X_n}(x) = \lambda e^{-\lambda x}.
\]

Therefore, \( \{X_i, i \geq 1\} \) are independent and exponentially distributed.

Then prove the condition in Definition 3 leads to the one in Definition 2. Let the intervals \( \{X_i, i \geq 1\} \) be independent and exponentially distributed with parameter \( \lambda \). We can get Eq. C.1 by deduction, first

\[
P\{N(t) = 0\} = P\{X_1 > t\} = \int_{t}^{\infty} \lambda e^{-\lambda x} dx = e^{-\lambda t}.
\]
Then suppose Eq. C.1 is true until $n$. We have

$$P\{N(t) = n + 1\} = \int_0^t \lambda e^{-\lambda(t-t')} P\{N(t') = n\} dt' = \frac{(\lambda t)^{n+1}}{(n+1)!} e^{-\lambda t}.$$
Appendix D

Poisson Arrival See Time Average and Coupon Collection Model with Poisson Replacement

Poisson Arrival See Time Average (PASTA) is the foundation of Poisson sampling—the statistics (e.g., expected system state value) observed by the arrivals from a Poisson process equals the actual statistics through the entire time line.

Let \( N(t) \) be system state at time \( t \), \( A(t) \) be the accumulated Poisson arrival curve and \( B \) be a certain set of system state. We define

\[
U(t) = \begin{cases} 
1 & \text{if } N(t) \in B \\ 
0 & \text{otherwise,} 
\end{cases}
\]

\[
V(t) = t^{-1} \int_0^t U(s)ds,
\]

\[
Y(t) = \int_0^t U(s)dA(s), \text{ and}
\]

\[
Z(t) = Y(t)/A(t).
\]

where \( U(t) \) is the indicator that the system state is in \( B \) at time \( t \), \( V(t) \) is the fraction of time during \([0, t]\) when system stays in \( B \), \( Y(t) \) is how many times that the Poisson arrivals observed that the system is in state \( B \), and \( Z(t) \) is the fraction of arrivals in \([0, t]\) who see \( N \) in \( B \).

And we have the following definition regarding the relation between system state \( U(t) \) and Poisson arrivals \( A(t) \).

**Definition 4** (Lack of Anticipation Assumption (LAA)). \( \{ A(t + u) - A(t), u \geq 0 \} \)
and \{U(s), 0 \leq s \leq t\} are independent \forall t \geq 0.

And PASTA is mathematically described as follows

**Theorem 1** (PASTA condition). *Under LAA, PASTA holds.*

The proof of Theorem 1 is given in [92]. Note that as long as LAA is satisfied, PASTA holds even if the Poisson arrivals \(A(t)\) changes the system state \(N(t)\) as in our collection collection model.

Since in our model, the future coupon replacements are independent from the number of coupons prior to the coupon replacement, PASTA holds for our model even the Poisson coupon type replacement clears the coupon types collected.
Bibliography


95

URL http://doi.acm.org/10.1145/2600057.2602889


URL http://doi.acm.org/10.1145/1807167.1807278

URL http://dx.doi.org/10.14778/1454159.1454167


URL http://doi.acm.org/10.1145/2825236.2825256


URL http://dx.doi.org/10.1016/j.comnet.2009.03.004


URL http://dl.acm.org/citation.cfm?id=2616448.2616475


[95] Shan, Y., “Use R-associated Sirling Number to calculate the number of cases where K attackers are assigned to M servers with m servers containing at least r attackers,” http://www.cse.psu.edu/yxs182/doc/rosso.pdf, [Online; accessed 24-Jul-2018].


Vita
Yuquan Shan

Education:
Ph.D. 2018 Pennsylvania State University, University Park, PA
Computer Science, GPA 3.97
B. Eng. 2013 Jinan University, Guangzhou, China
Electrical Engineering, GPA 86.6/100

Employment:
Research Assistant
Pennsylvania State University, University Park, PA
2014.6 - 2018.11
Software Engineer Intern, Facebook Inc., Menlo Park, CA
2017.9 - 2017.12
Research Intern
IBM Research, San Jose, CA
2017.5 - 2017.8

Publications:
Scheduling Distributed Resources in Heterogeneous Private Clouds.
MASCOTS 2018
Y. Shan, C. Lo Prete, G. Kesidis, D. J. Miller
A Simulation Framework for Uneconomic Virtual Bidding in Day-ahead Electricity Markets.
ACC 2017: 2705-2712
Y. Shan, G. Kesidis, D. Fleck
Cloud-Side Shuffling Defenses against DDoS Attacks on Proxied Multiserver Systems.
CCSW@CCS 2017: 1-10
G. Kesidis, N. Nasiriani, Y. Shan, B. Urgaonkar, I. Lambadaris
Multicommodity Games in Public-cloud Markets Considering Subadditive Resource Demands.
INFOCOM Workshops 2017: 654-658
Y. Shan, G. Kesidis, D. Fleck, A. Stavrou
Preliminary Study of Fission Defenses against Low-volume DoS Attacks on Proxied Multiserver Systems.
MALWARE 2017: 67-74
X. Hu, Y. Shan, G. Kesidis, S. Sarkar, R. Dhar, S Fdida
Multiperiod Subscription Pricing for Cellular Wireless Entrants.
CISS 2016: 326-330
Y. Shan, G. Kesidis
Optimal Power Flow with Random Wind Resources.
HICSS 2016: 2426-2434
G. Kesidis, Y. Shan, B. Urgaonkar, J. Liebeherr
Network Calculus for Parallel Processing.
Y. Shan, J. Raghum, G. Kesidis, C. Griffin, K. N. Levitt, D. J. Miller, J. Rowe, A. Scaglione
Generation Bidding Game with Potentially False Attestation of Flexible Demand.