EFFICIENT ON-DEMAND OPERATIONS IN LARGE-SCALE INFRASTRUCTURES

BY

STEVEN Y. KO

B.S., Yonsei University, 1999
M.S., Seoul National University, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Abstract

In large-scale distributed infrastructures such as clouds, Grids, peer-to-peer systems, and wide-area testbeds, users and administrators typically desire to perform *on-demand operations* that deal with the most up-to-date state of the infrastructure. However, the scale and dynamism present in the operating environment make it challenging to support on-demand operations efficiently, i.e., in a bandwidth- and response-efficient manner.

This dissertation discusses several on-demand operations, challenges associated with them, and system designs that meet these challenges. Specifically, we design and implement techniques for 1) on-demand group monitoring that allows users and administrators of an infrastructure to query and aggregate the up-to-date state of the machines (e.g., CPU utilization) in one or multiple groups, 2) on-demand storage for intermediate data generated by dataflow programming paradigms running in clouds, 3) on-demand Grid scheduling that makes worker-centric scheduling decisions based on the current availability of compute nodes, and 4) on-demand key/value pair lookup that is overlay-independent and perturbation-resistant. We evaluate these on-demand operations using large-scale simulations with traces gathered from real systems, as well as via deployments over real testbeds such as Emulab and PlanetLab.

To my wife Kyungmin and my daughter Natalie,
my mom,
and my dad who is in Heaven.

Acknowledgments

For God so loved the world that he gave his one and only Son, that whoever believes in him shall not perish but have eternal life.

John 3:16

I do not know how to express my gratitude to God. My words are just not enough. I have gained so much over the past 7 years of my life as a graduate student — I earned a degree, got married, and now have a daughter. But most of all, I have tasted a glimpse of joy that I can have in Christ alone; and I know it was only possible by His grace. All the glory belongs to Him.

I would like to thank my wife, Kyungmin Kwon. She is my joy. She is my strength when I need it the most. Her encouragement is always rejuvenating. Her biblical advice always challenges me. I cannot wait to spend the rest of my life with her, growing together as Christians.

I would like to thank my advisor, Indranil Gupta, tremendously for his patience, insights, mentoring, and high standard. It is very clear that I could not have finished my thesis without him. He was there literally whenever I needed him, and always full of analogies and insights that were thought-provoking.

I would like to thank my committee members, Dejan Milojicic, Klara Nahrstedt, and Tarek Abdelzaher. Dejan Milojicic has offered me invaluable advice ever since I first met him as an intern at HP Labs. His comprehensive and wide view towards research has opened my eyes to see the areas where I (still) need to grow. Klara Nahrstedt has offered me great insights about the problems at hand, whether it be research or career. Her questions have always led me to think twice about my assumptions and approaches. Tarek Abdelzaher always squeezed out enough time for me in his busy schedule.

I would like to thank current and former members of DPRG. I would like to thank Ramsés Morales and Jay Patel for our (sometimes heated) discussions over various topics and intellectual challenges that they posed upon me, Brian Cho and Imranul Hoque for their refreshing ideas and energy, our project ISS, and the future work for ISS, Yookyung Jo for HoneyAdapt, Lucas Cook for his feedback on my research, Charlie (and Jenny) Yang for our fun gatherings and their impeccable taste of good eats, and Nat (and Ramona) Thompson for their hospitality that enriched my graduate life.

I would like to thank Praveen Yalagandula, Subu Iyer, and Vanish Talwar from HP Labs for the Moara project. Praveen Yalagandula and I had countless discussions over the phone. I did not know two people can prove a theorem over the phone until I did it with him. It was quite an experience. Subu Iyer and Vanish Talwar also provided much help for project discussions and paper writing.

I would like to thank Jin Liang for his advice and support as a senior graduate student. I would like to thank Long Vu, Thadpong Pongthawornkamol, William Conner, Rahul Malik, Ahsan Arefin, Shameem Ahmed, Ying Huang, Wenbo He, and Won Jeon from the MONET group for responding to my endless requests for feedback on my papers and talks. I would like to thank professors Robin Kravets, Matt Caesar, Chandra Chekuri, and David Forsyth for their advice on my job search. I would also like to thank Yoonkyong Lee, Younhee Ko, Jin Heo, Young-Rae Cho, MyungJoo Ham, Eunsoo Seo, Hwangnam Kim, Jihyuk Choi, Keunsoo Yim, Kihwal Lee, and Seung Yi for helping me in various stages of my graduate life.

Jintae Kim, Jungmin So, and Eun-Gyu Kim have been supporting and challenging me as friends and fellow Christians. I am very thankful for the time we spent together, all the trips that we took, silly jokes, research discussions, Bible discussions, and the joy and struggles that we shared. It is my great gain to have such good friends.

I would like to thank Pastor Min, Ms. Kim, and the associate pastors at CFC. Through Pastor Min, I have learned how to interpret life through the lens of the Bible as well as the importance of prayer and His word. Ms. Kim has showed me by example how to pray and live biblically, and supported me and my family with endless prayers.

I would like to thank all my small group leaders, Jinju lee, Angelo DeGuzman, Miryung Kim, David Kim, Ayako Nishiguchi, Jintae Kim, Phil Su, and Jae Lee. I am blessed to have them as my leaders, and I always feel the support of their prayers. I would also like to thank all the people of GBS and MABS for sharing their lives with me and praying for me.

I would like to thank my in-laws. They made several trips to US to help me and my wife in many ways when we needed. They have sacrificed so much, so that I can concentrate on my thesis especially in the last few months of my graduate life.

I would like to thank my mom and dad. They have passed down the legacy of praise and worship to God that I would like to continue on. They have supported me and persevered with me throughout my life with prayers and encouraging words. I would also like to thank my sister, Hwakyoung Ko, for her prayers.

My research has been supported by generous grants from NSF (NSF IIS 0841765, NSF CNS 0448246, and NSF ITR CMS 0427089), HP Labs (Palo Alto), and the Department of Computer Science, University of Illinois at Urbana-Champaign.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The growth of the Internet has brought to us the daily use of large-scale distributed infrastructures. Web users interact with data centers daily via Web-based applications. Computer science researchers perform systems and networking experiments on wide-area testbeds such as PlanetLab [83] and local-area testbeds such as Emulab [26]. Scientific communities utilize various computational Grids to perform their experiments [28]. Recently, the promises of cloud computing initiatives by industry leaders and academics such as the OpenCirrus Cloud Computing Testbed [11], Amazon [2], Google [34], IBM [35], and HP [42] provide evidence that this trend will likely continue for years to come.

These infrastructures are often geographically distributed, and comprise tens of thousands of servers networked via the Internet and LANs. Workloads on these infrastructures are diverse and include Web 2.0 applications such as search and collaborative document editing, dataflow programming frameworks such as MapReduce [23], Pig [77], and Dryad [46], data-intensive scientific applications, etc. In addition, management software runs in the background in order to automate management of resources.

We believe that an important class of operations in these large-scale infrastructures is *on-demand operations*, defined broadly as *operations that act upon the most up-to-date state of the infrastructure.* These on-demand operations are necessitated either by the need from users and administrators or by the characteristics of workloads running on the infrastructures. For example, a data center administrator typically desires to have on-demand monitoring capability for systems characteristics of the data center. This capability for querying the up-to-date state (e.g., CPU utilization, software versions, etc.) of a data center on-demand allows the administrator the ability to understand the inner-working of the data center for trouble-shooting purposes, as well as to make well-informed decisions for management tasks such as capacity planning and patch management. Another example is on-demand scheduling in computational Grids, where scheduling decisions often need to reflect the most current resource availability such as the availability of compute nodes. Thus, supporting these on-demand operations efficiently is both desirable and necessary.

The challenges in supporting such operations arise from two categories — scale and dynamism. Each of these challenges appears in a variety of flavors

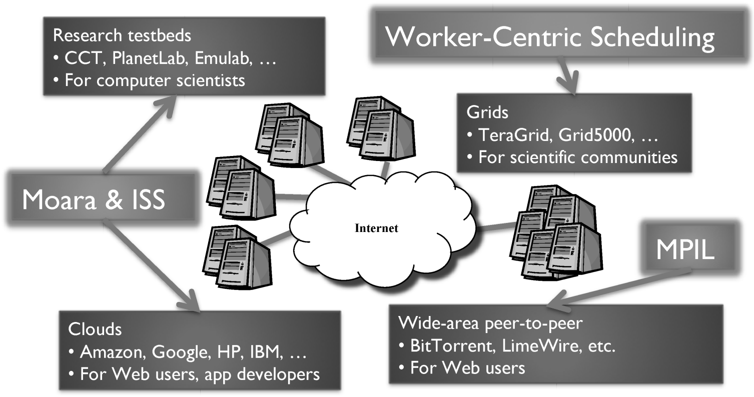
Research testbeds
• CCT, PlanetLab, Emulab, …
• For computer scientists
Worker-Centric Scheduling
Grids
• TeraGrid, Grid5000, …
• For scientific communities
Moara & ISS
Internet
MPIL
Clouds
• Amazon, Google, HP, IBM, …
• For Web users, app developers
Wide-area peer-to-peer
• BitTorrent, LimeWire, etc.
• For Web users

Figure 1.1: On-Demand Operations and Types of Infrastructures in This Dissertation

unique to the specific on-demand operation at hand. For example, scale comes not only from tens of thousands of physical machines in the infrastructure, but also from millions of attributes to monitor in the case of on-demand monitoring, petabytes of data in the cases of on-demand replication and scheduling, and tens of thousands of tasks to process in the case of on-demand scheduling. Dynamism comes not only from the failures of services and machines, but also from the changes in resource availability (e.g., CPU, bandwidth, and online-offline status), usage, workload, etc.

In this dissertation, we present various on-demand operations and detail each one of them in the next few chapters. We believe that exploring on-demand operations reveals the many faces of dynamism and scale present in large-scale infrastructures. By doing so, we can revisit well-known aspects of dynamism and scale, as well as uncover new aspects of them that were previously overlooked. Ultimately, the collective knowledge accumulated from this exploration can form a basis of reasoning when designing new large-scale infrastructures or services.

1.1 Thesis and Contributions

Our thesis is that *on-demand operations can be implemented efficiently in the face of scale and dynamism in a variety of distributed systems*. We validate this thesis by proposing on-demand operations for a variety of large-scale dynamic infrastructures. We evaluate our proposed on-demand operations using large-scale simulators and testbeds such as Emulab [26] and PlanetLab [81], using traces gathered from real systems and synthetic workloads that mimic users behaviors.

Our contributions are the design of four systems that advocate the necessity and benefits of on-demand operations. The four systems have been chosen to cover four diverse and popular types of distributed infrastructures — research testbeds, Grids, wide-area peer-to-peer environments, and clouds. We believe

System	**Scale**	**Dynamism**
Moara	Machines, Monitoring Data, and Groups	Group Churn and Workload
ISS	Machines and Intermediate Data	Resource Availability
W-C Scheduling	Machines and Data	Resource and Data Availability
MPIL	Machines	Perturbation

Table 1.1: Types of Scale and Dynamism for Each System

that this is a comprehensive list of large-scale infrastructures, and any large-scale distributed system can be classified into one of the four categories.

In addition, each of the four systems implements an essential on-demand operation in each category. Moara in Chapter 2 implements on-demand monitoring, which is an essential operation in research testbeds and clouds. ISS in Chapter 3 implements on-demand replication, which is another essential operation in research testbeds and clouds. Worker-centric scheduling algorithms in Chapter 4 implement on-demand scheduling algorithms essential in Grids. Lastly, MPIL in Chapter 5 implements an on-demand key/value lookup mechanism that is essential in wide-area peer-to-peer environments. Figure 1.1 shows the operations and types of infrastructures that this dissertation discusses.

All four systems address the challenges of scale and dynamism. Table 1.1 summarizes the different types of scale and dynamism each system addresses, and Figure 1.2 shows the taxonomy of the solution space. As shown in Figure 1.2, each of the four systems provides a solution for a large-scale and dynamic environment, in which current infrastructures are operating. Other three regions (small-scale static, small-scale dynamic, and large-scale static environments) represent environments in which traditional distributed systems have been operating. In each chapter, we revisit this taxonomy and discuss it in detail for the specific operation at hand.

We briefly summarize each system below.

Moara – an On-Demand Group Monitoring System [56] : We argue that a monitoring system should be 1) group-based, as users and administrators typically desire to monitor implicit groups of machines (e.g., web servers, databases, etc.), and 2) on-demand, as the users should be able to query the most up-to-date data about the groups. Although previous systems have developed techniques to support aggregation queries, they target "global" aggregation — each machine in the whole infrastructure receives and answers every aggregation query, thus wasting bandwidth if a query actually targets only a group of machines. Our system called Moara is the first system (to the best of our knowledge) that implements a distributed technique for aggregating data from a group (or multiple groups) in a bandwidth-efficient way without contacting

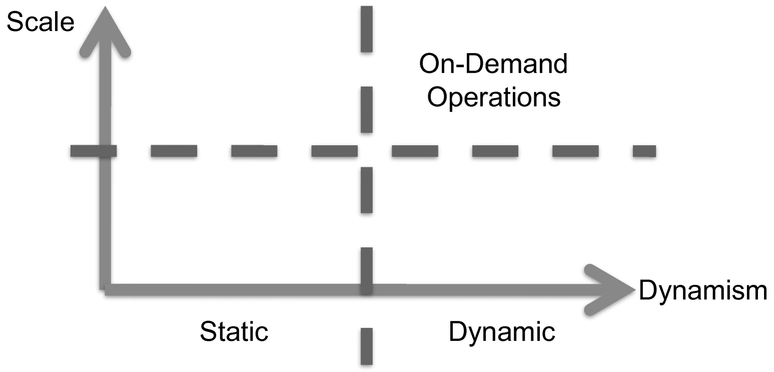

Figure 1.2: Taxonomy of the Solution Space

all machines in the infrastructure.

Our on-demand group aggregation technique addresses a new dimension added to the challenges of scale with respect to the number of machines and the number of attributes to monitor - the number of groups. This new dimension has been recognized in the context of multicast [5]. However, since aggregation employs different techniques such as in-network processing, we need a solution that is tailored towards the harmony with the techniques for aggregation.

The challenge of dynamism comes from two sources, group churn (i.e., group membership change) and workload. To be more specific, the size and composition of each group can change over time due to joining and leaving of machines as well as changes in the attribute-value space. In addition, a user may need to aggregate data from different groups at different times, so we can expect that the query rate and target might vary significantly over time. Thus, a group aggregation system has to be adaptive to the group churn and changes in the user workload.

Moara addresses these challenges with three techniques that we detail in Chapter 2.

ISS (Intermediate Storage System) [54]: We argue for the need to design a storage system that treats distributed intermediate data as the first-class citizen. Intermediate data is generated by emerging new multi-stage dataflow programming frameworks such as MapReduce [23], Hadoop [108], Pig [77], and Hive [29] in the clouds, and plays a critical role in execution of dataflow programs. Specifically, a failure during a job execution resulting in a loss of the intermediate data greatly hampers the overall performance, resulting in *cascaded re-execution*, i.e., some tasks in *every stage from the beginning* have to be re-executed sequentially up to the stage where the failure happened. As a result, the job completion time increases dramatically.

Our approach to this problem is on-demand replication that provides intermediate data availability. The challenge of dynamism arises from replication interfering with foreground dataflow programs as their resource usages of disks and network conflict with each other. We especially identify network bandwidth as the bottleneck, and hence argue that it is important to minimize network interference. The challenge of scale comes from two main sources of machine scale and data scale, as dataflow programming paradigms aim to scale up to several thousands of machines handling petabytes of data.

We detail how we address these challenges in Chapter 3.

On-Demand Worker-Centric Scheduling [55]: We argue that efficient scheduling algorithms for data-intensive tasks are on-demand, i.e., they make scheduling decisions based on the most up-to-date state such as the availability of compute nodes (or "workers") and the characteristics of the current input data. This is due to the challenges of scale and dynamism. The challenge of scale comes from the number of workers and the amount of data that needs to be processed. The challenge of dynamism arises from dynamically-changing resource availability and the availability of data at each worker.

In order to address this scale and dynamism, we have developed a family of on-demand worker-centric scheduling algorithms for data-intensive tasks. These algorithms consider the availability of workers and the availability of data at each worker as the primary criterion. Our algorithms make a scheduling decision for a task to a worker only when the worker can start executing the task immediately. Our worker-centric scheduling algorithms are different from task-centric scheduling (e.g., [94, 13, 86]), in which a central scheduler assigns a task to a worker without considering whether or not the worker can start executing the task immediately after the task assignment.

This is discussed in detail in Chapter 4.

On-Demand Key/Value Store [53] We argue that an on-demand key/value store such as peer-to-peer distributed hash tables (DHTs) (e.g., Pastry [93], Chord [100], Tapestry [111], Kelips [36], etc.) needs to be 1) overlay-independent, i.e., it should run over any arbitrary overlay structure without requiring aggressive maintenance, and 2) perturbation-resistant, i.e., it should be resistant to ordinary stresses in the operating environment such as churn, short-term unresponsiveness, etc.

These two requirements come from the challenges of scale and dynamism that on-demand key/value stores face when running in the real environment. The challenge of scale comes from a well-known fact that a large number of machines participate in wide-area peer-to-peer systems. The challenge of dynamism arises from the operating environment, i.e., the Internet, where short-term unresponsiveness of peers is common. Overlay-independence helps to meet both challenges, as it reduces bandwidth consumption in dynamic environments in a scalable manner. Perturbation-resistance helps to meet the challenge of dynamism as it tolerates short-term unresponsiveness of each peer.

Our system called MPIL (Multi-Path Insertion and Lookup) satisfies both requirements of overlay-independence and perturbation-resistance. MPIL achieves this with low maintenance cost with slightly increased per-lookup cost compared to DHTs.

We detail the design in Chapter 5.

1.2 Related Work

There is a large body of work that can be classified as on-demand operations. This dissertation contributes to this research literature by identifying scale and dynamism as a collective set of challenges and concretely showing how on-demand operations can be efficient in spite of them in large-scale distributed systems. In comparison, previous work in the literature has been focusing only on the designs of operations themselves.

In this section, we discuss related work for on-demand operations in general. We focus on-demand operations pertaining to large-scale infrastructures such as clouds, Grids and PlanetLab. In each subsequent chapter, we discuss related work for the specific on-demand operation we discuss in that chapter.

MON [62] is among the first systems that support on-demand management operations for large-scale infrastructures. It supports propagation of one-shot management commands. In PlanetLab, there are other management tools that support on-demand operations, such as vxargs and PSSH [84]. However, these tools are mainly centralized, and thus they do not address scalability explicitly.

A few Grid scheduling algorithms can be considered as on-demand scheduling since they make scheduling decisions based on the current status of the Grid. Examples include a pull-based scheduler proposed by Viswanathan *et al.* [106] and theoretical work by Rosenberg *et al.* [91, 92]. Our work on worker-centric scheduling in Chapter 4 shares the on-demand philosophy with these approaches. However, our contribution is not just the design, but also the concrete discussion about why and how on-demand approaches work better for the requirements of scalability and dynamism.

There are many on-demand operations that are not the focus of this dissertation. Recently, cloud computing services such as Amazon Web Services [2], Google AppEngine [34], and Microsoft Azure [3], Right Scale [90] have started to provide on-demand scale-up and scale-down — these platforms automatically add or reduce CPU, memory, disk, and network allocation to a hosted service depending on the user traffic the service receives.

Gossip-based multicast systems [8, 37, 52] perform multicast on-demand. These systems typically do not impose and maintain certain structures over participating nodes. Instead, they employ robust gossip-based mechanisms to perform multicast over dynamic non-deterministic structures. As a result, these systems are successful in combating dynamism among participating nodes.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents Moara, a system for on-demand monitoring in large-scale infrastructures. We discuss the use cases of Moara, and three techniques that we develop in order to achieve Moara's goals of scalability, flexibility, and efficiency. We present the evaluation results of Moara using large-scale simulations and testbeds such as Emulab and PlanetLab. Chapter 3 presents ISS, a new storage system for managing intermediate data in dataflow programming paradigms. We discuss the problem of managing intermediate data, requirements of a solution, and the design of ISS. We present the evaluation results of ISS using Emulab. Chapter 4 presents worker-centric scheduling strategies for data-intensive Grid applications. We discuss the problems of task-centric scheduling strategies, and show why worker-centric scheduling strategies are effective in dealing with the problems. We present the evaluation results using simulations driven by a real application trace. Chapter 5 discusses MPIL, an on-demand key/value store. We discuss how MPIL achieves desired properties of overlay-independence and perturbation-resistance in dynamic environments. We present the evaluation results of MPIL using large-scale simulations. Finally, Chapter 6 concludes with future directions.

Chapter 2

Moara: An On-Demand Group Monitoring System

This chapter presents the motivation, use cases, design, and evaluation of Moara, an on-demand group monitoring system. Moara addresses the challenge of scale stemming from the number of machines, the amount of monitoring data, and the number of groups. In addition, Moara addresses the challenge of dynamism arising from group churn and workload. We detail how Moara addresses each challenge in this chapter, starting from the motivation behind Moara.

2.1 Motivation

A frequent need of the users and the administrators of such infrastructures is monitoring and querying the status of *groups* of machines in the infrastructure, as well as the infrastructure as a whole. These groups may be static or dynamic, e.g., the PlanetLab slices, the machines running a particular service in a data center, or the machines with CPU utilization above 50%. Further, users typically desire to express complex criteria for the selection of the host groups to be queried. For example, "find top-3 loaded hosts where (ServiceX = true) and (Apache = true)" is a query that targets two groups - hosts that run service X and hosts that run Apache. Dynamic groups mean that the size and composition of groups vary across different queries as well as time.

In general, users and administrators desire to monitor the performance of these groups, to troubleshoot any failures or performance degradations, and to track usage of allocated resources. These requirements point to the need for a *group-based on-demand querying system* that can provide instantaneous answers to queries over in-situ data targeting one or more groups. In fact, several existing distributed aggregation systems [74, 88, 109] can be considered as a special case of group-based querying systems, as they target querying of only a single group, i.e., the entire system.

Any group-based querying system should satisfy three requirements: *flexibility*, *efficiency*, and *scalability*. First, the system should be flexible to support expressive queries that deal with multiple groups, such as unions and intersections of different groups. Second, the system should be efficient in query resolution—it should minimize the message overhead while responding quickly with an answer. Third, the system should scale with the number of machines, the number of groups, and the rate of queries.

We propose Moara, a new group-based distributed aggregation system that targets all three requirements. A query in Moara has three parts: (*query-attribute*, *aggregation function*, *group-predicate*), e.g., (Mem-Util, Average, Apache = true). Moara returns the resulting value from applying the *aggregation function* over the values of *query-attribute* at the machines that satisfy the *group-predicate*.

Moara resolves a query on-demand, i.e., it propagates a query to end-hosts that are monitored. Each end-host manages and stores its own monitoring data locally, and there is no periodic collection of monitoring data. Each end-host also participates in the query execution.

Moara makes two novel design contributions over existing systems [74, 88, 109]. First, Moara maintains aggregation trees for different groups adaptively based on the underlying environment and the injected queries to minimize the overall message cost and query response time. Basically, the aggregation tree for a group in Moara is an optimized sub-graph of a global spanning tree, which spans all nodes in the group. By aggregating data over these group-based aggregation trees, Moara achieves lower message cost and response latency for queries compared to other aggregation systems that contact all nodes. Further, we adapt each aggregation tree to deal with dynamism.

Second, Moara's query processor supports *composite* queries that target multiple groups simultaneously. Composite queries supported by Moara are arbitrary nested set expressions built by using logical operators `or` and `and`, (respectively set operations $\cup$ and $\cap$) over simple group-predicates. Simple group-predicates are of the form (*attribute op value*), where $op \in \{<, >, \leq, \geq, =, \neq\}$. Consider our previous example "find top-3 loaded hosts where (ServiceX = true) and (Apache = true)", which is a composite query that targets the intersection of two groups - hosts that run service X and hosts that run Apache. Instead of blindly querying all the groups present in a query, Moara's query processor analyzes composite queries and intelligently decides on contacting a set of groups that minimizes the communication overhead.

We implemented a prototype of Moara by leveraging the FreePastry DHT (Distributed Hash Table) [93] and SDIMS [109] systems. Our evaluation consists of experiments on Emulab [26] and PlanetLab, as well as large-scale simulations. Our experimental results indicate that, compared to previous global hierarchical aggregation systems, Moara reduces response latency by up to a factor of 4 and achieves an order of magnitude bandwidth savings. Our scalability experiments confirm that Moara's overhead for answering a query is independent of the total number of nodes in the system, and only grows linearly with the group size. Finally, we show that Moara can answer complex queries within hundreds of milliseconds in systems with hundreds of nodes under high group churn.

In this work, we focus on efficiently supporting one-shot queries (as opposed to repeated continuous queries) over a common set of groups, since we expect this type of queries to be more common in the kind of infrastructures we are targeting at — data centers and federated computing systems. We expect most users will be performing one-shot queries over common groups (e.g., the same PlanetLab slice, machines in a data center, etc) during the time when their service or experiment is running. Further, a user interested in monitoring groups continually can invoke one-shot queries periodically. Our use cases in Section 2.3 motivates this design decision further.

Any distributed system subjected to dynamism in the environment, suffers from the CAP dilemma [9], which states that it is difficult to provide both strong consistency guarantees and high availability in failure-prone distributed settings. Moara treads this dilemma by preferring to provide high availability and scalability, while providing eventual consistency guarantees on aggregation results. This philosophy is in line with that of existing aggregation systems such as Astrolabe [88] and SDIMS [109]. Moara could also allow the use of metrics proposed by Jain et al. [47, 48] in order to track the imprecision of the query results; however, studying these is beyond the scope of this dissertation.

2.2 Taxonomy

Figure 2.1 shows the taxonomy of the monitoring solution space. Traditionally, popular monitoring solutions used a centralized DB-based approach, where all monitoring data is collected and stored periodically in a DB. This approach works well in small-scale environments (the bottom two regions in Figure 2.1) because the collection time can be short and the rate of collection can be frequent. Even in a large-scale static environment (the upper left region), a scalable DB-based approach such as a replicated DB can be used, since attribute values do not change; after collecting the values once, there is no need to collect them again. However, anectotal evidence from the industry suggests that this monitoring data collection time can take up to several hours with a few thousands of machines, which led to the development of on-demand monitoring solutions. Moara implements an on-demand monitoring operation that provides a solution for a large-scale dynamic environment.

2.3 Use Cases

We highlight the need for on-demand flexible querying and for dealing with dynamism by presenting two motivating scenarios - data centers and federated infrastructures.

Consolidated Data Centers: In the last few years, medium and large-scale enterprises have moved away from maintaining their own clusters, towards subscribing to services offered by consolidated data centers. Such consolidated data centers consist of multiple locations, with each location containing several

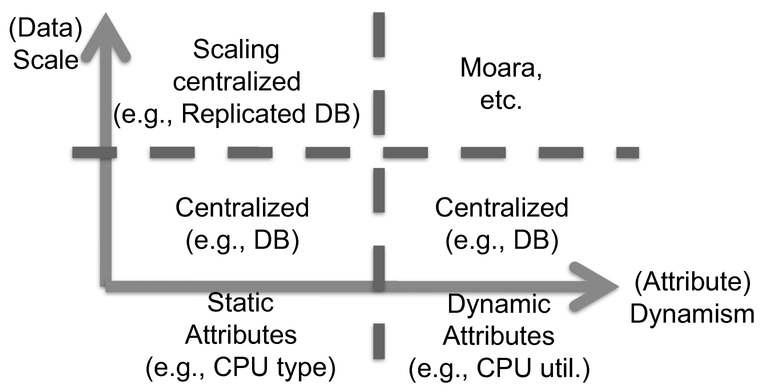
(Data) Scale
Scaling centralized (e.g., Replicated DB)
Moara, etc.
Centralized (e.g., DB)
Centralized (e.g., DB)
(Attribute) Dynamism
Static Attributes (e.g., CPU type)
Dynamic Attributes (e.g., CPU util.)

Figure 2.1: Taxonomy of the Monitoring Solution Space

thousands of servers. Each server runs heterogeneous operating systems including virtual machine hosts. While such consolidation enables running unified management tasks, it also introduces the need to deal with scale.

Workloads on these data centers typically include Terminal Services, SOA-based transaction workloads (e.g., SAP), and Web 2.0 workloads, e.g., searching and collaboration. Table 2.1 presents some on-demand one-shot queries that data center managers and service owners typically desire to run on such a virtualized enterprise. Several of these one-shot queries are for aggregating information from a common group of nodes including cases where groups are expressed as unions of groups (e.g., the third query in table), or intersections (e.g., the last query). We would like to generalize this to provide managers with a powerful tool supporting flexible queries using arbitrarily nested unions and intersections of groups. In addition, these workloads vary in intensity over time, causing considerable dynamism in the system, e.g., terminal services facing high user turnaround rates.

Federated Computing Infrastructures: In today's federated computing infrastructures such as PlanetLab [83] and global Grids [28], as well as in proposed infrastructures, e.g., GENI [76], users wish to query current statistics for their distributed applications or experiments. For instance, PlanetLab creates virtual subgroups of nodes called "slices" in order to run individual distributed applications. Monitoring is currently supported by tools such as CoMon [80] and Ganglia [70], which periodically collect CPU, memory, and network data per slice on PlanetLab [83]. Due to their periodic nature, they are not open to on-demand queries that require up-to-date answers. Further, increasing the frequency of data collection is untenable due to storage and communication costs.

In contrast to the above systems, we need a system to answer one-shot queries that seek to obtain up-to-date information over a common group of machines, that can be run on-demand or periodically by an end-host, and are

Tasks	**Queries**
Resource Allocation	Average utilization for servers belonging to (i) floor F, (ii) cluster C, and (iii) rack R
	Number of machines/VMs in a given cluster C
VM Migration	Average utilization of VMs running application X version 1 or version 2
	List of all VMs running application X are VMWare based
Auditing/Security	Count of all VMs/machines running firewall
	Count of all VMs running ESX server and Sygate firewall
Dashboard	Max response time for Service X
	Count of all machines that are up and running Service X
Patch management	List of version numbers being used for service X
	Count of all machines that are in cluster C and running service X.version Y

Table 2.1: Illustrative Queries for Managing the Virtualized Enterprise

flexibly specified. Some examples of our target queries include: number of slices containing at least one machine with CPU utilization $> 90\%$ (basic query), CPU utilization of nodes common to two given slices (intersection query), or free disk space across all slices in a given organization (union query).

Need for Group-based Aggregation: As illustrated by above two target scenarios, we expect that most of the queries are one-shot queries over common groups of machines. Moreover, the predicate in a query specified as a logical expression involves several groups, e.g., some groups in the above examples include the set of nodes in a PlanetLab slice, the set of nodes running a given Grid task, the set of nodes with CPU utilization $> 90\%$, etc. In the worst case, such a group may span the entire system.

In practice though, we expect the group sizes to vary across different queries and with time. In Figure 2.2, we plot the distribution of PlanetLab slice sizes, analyzed from an instance of CoMon [80] data. Notice that there is a considerable spread in the sizes. As many as 50% of the 400 slices have fewer than 10 assigned nodes, thus a monitoring system that contacts all nodes to answer a query for a slice is very inefficient. If we consider only nodes that were actually in use (where a slice has more than one process running on a node), as many as 100 out of 170 slices have fewer than 10 active nodes. In another example case, Figure 2.3 presents the behavior of two jobs over a 20-hour period from a real 6-month trace of a utility computing environment at HP with 500 machines receiving animation rendering batch jobs. This plot shows the dynamism in each group over time.

These trace studies indicate that group sizes can be expected to be varying across time in both consolidated centers as well as in federated computing infrastructures. Thus, an efficient querying system has to avoid treating the entire

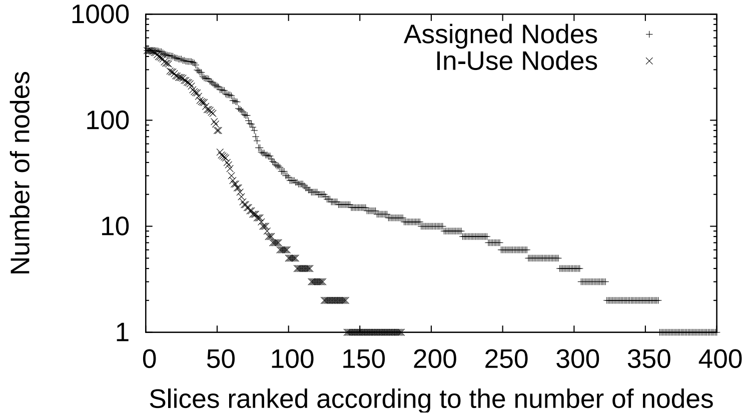

Figure 2.2: Usage of PlanetLab nodes by different slices. We show both node assignment to slices and active usage of nodes. Data collected from a CoTop snapshot [84].

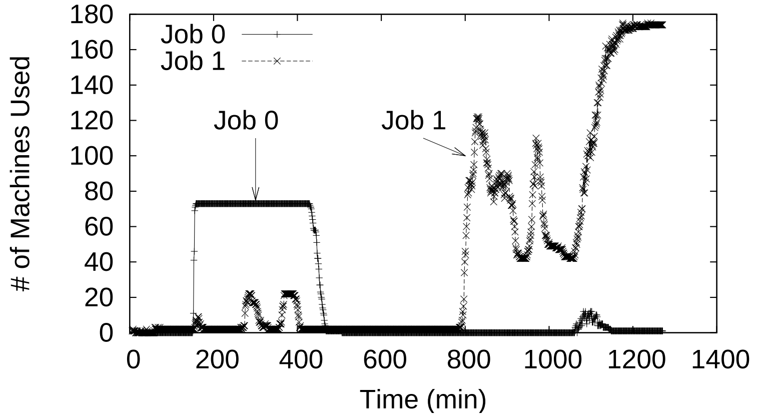

Figure 2.3: Usage of HP's utility computing environment by different animation rendering jobs. We show the number of machines each job uses.

system as a single group and globally broadcasting queries to all nodes.

2.4 The Basics of Moara

In this section, we first discuss how Moara end-nodes maintain data and how queries are structured. Then we discuss how Moara builds trees for individual groups.

2.4.1 Data and Query Model

Information at each node is represented and stored as (*attribute, value*) tuples. For example, a machine with CPU capacity of 3Ghz can have an attribute (CPU-Mhz, 3000). Moara has an agent running at each node that monitors the node and populates (*attribute, value*) pairs.

A query in Moara comprises of three parts: (*query-attribute*, *aggregation function*, *group-predicate*). The first field specifies the attribute of interest to be aggregated, while the second field specifies the aggregation function to be used on this data. We require this aggregation function to be partially aggregatable. In other words, given two partial aggregates for multiple disjoint sets of nodes, the aggregation function must produce an aggregate that corresponds to the union of these node sets [88, 109]. This admits aggregation functions such as enumeration, max, min,sum, count, or top-k. Average can be implemented by aggregating both sum and count.

The third field of the query specifies the group of machines on which the above aggregation is performed. If no group is specified, the default is to aggregate values from all nodes in the system. A *group-predicate* (henceforth called a "predicate") is specified as a boolean expression with `and` and `or` operators, over *simple* predicates of the following form: (*group-attribute op value*), where $op \in \{<, >, =, \leq, \geq, \neq\}$. Note that this set of operators allows us to implicitly support `not` in a group predicate. Any *attribute* that a Moara agent populates can be used as either *query-attribute* or *group-attribute*.

A simple query contains a simple predicate. For example, the simple predicate (ServiceX = true) defines all machines running ServiceX. Thus, a user wishing to compute the maximum CPU usage across machines where ServiceX is running will issue the following query: (CPU-Usage, MAX, (ServiceX = true)). Alternately, the user could use a *composite* predicate, e.g., (ServiceX = true `and` Apache = true). This composite query is defined with set operators $\cup$ and $\cap$.

Note that the query model can be easily extended so that instead of a *query-attribute*, a querier can specify any arbitrary program that operates upon simple *(attribute, value)* pairs. For example, a querier can specify a program that evaluates (CPU-Available > CPU-Needed-For-App-A) as *query-attribute*, to see how many nodes are available for the application A. Similarly, *group-predicate* can be extended to contain multiple attributes by defining new attributes. For example, we can define a new attribute *att* as (CPU-Available > CPU-Needed-For-App-A), which takes a boolean value of true/false. Then *att* can be used to specify a group. However, for this dissertation, we mainly focus on the techniques for efficiently answering the queries for given *group-predicates* and hence restrict query model to contain only simple attributes.

2.4.2 Scalable Aggregation

We describe here how Moara aggregates data for each group.

DHT trees: For scalability with large number of nodes, groups, and queries, Moara employs a peer-to-peer in-network aggregation approach that leverages the computing and network resources of the distributed infrastructure itself to compute results. These trees are used for spreading queries, and aggregating answers back towards the source node. In our architecture, a lightweight

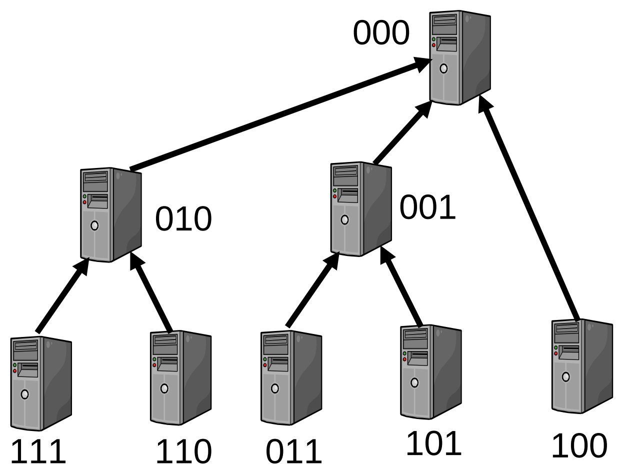

Figure 2.4: DHT tree for an ID with prefix 000

Moara agent runs at each server from which data needs to be aggregated. These agents participate in a structured overlay routing algorithm such as Pastry [93], Tapestry [111], or Chord [100]. These systems allow routing within the overlay, from any node to any other node, based on the IDs of these nodes in the system. Moara uses this mechanism for building aggregation trees called DHT trees, akin to existing systems [17, 16, 109]. A DHT tree contains all the nodes in the system, and is rooted at a node that maps to the ID of the group. For instance, Figure 2.4 shows the tree for an ID with prefix 000 using Pastry's algorithm with one-bit prefix correction. We choose to leverage a DHT, since it handles physical membership churn (such as failures and join/leave) very modularly and efficiently. Also, we can construct aggregation trees clearly, given a group predicate.

Basics of Resolving Queries: Given a simple query with predicate p, Moara uses MD-5 to hash the *group-attribute* field in p and derives a bit-string that stands for the group ID. The DHT tree for this ID is then used to perform aggregation for this query, e.g., Figure 2.4 shows the DHT tree for an attribute "ServiceX" that hashes to 000.

When a simple query is generated at any node in Moara, it is first forwarded to the root node of the corresponding DHT tree via the underlying DHT routing mechanism. The root then propagates it downwards along the DHT tree to the leaves. When a leaf receives a query, it evaluates the predicate p in the query (e.g., ServiceX=true). If the result is true, it replies to its parent the local value for the query attribute (e.g., CPU-Usage). Otherwise, it sends a null reply to its parent. An internal node waits to reply to its parent until *all* its children have replied or until a timeout occurs (using values in Section 2.8). Then, it aggregates the values reported by its children, including its own contribution if the predicate is satisfied locally, and forwards the aggregate to its parent. Finally, the root node replies to the original querying node with the aggregated value.

Moara Mechanisms: The above "global aggregation" approach has every node in the system receive every query. Hence, it is inefficient in resolving queries targeting specific groups. Moara addresses this via three mechanisms.

First, Moara attempts to prune out branches of the tree that do not contain any node satisfying the predicate p. We call this tree a pruned tree or a *group tree* for p. For example, in Figure 2.4, if nodes 111, 110, and 010 do not satisfy the predicate, then the root does not forward the query to 010. However, this raises a challenge – how do internal nodes know whether any of their descendants satisfy the predicate. For instance, if node 110 decides to install ServiceX and thus satisfies the predicate, the path from the root to this node will need to be added to the tree. Further, if the composition of a group changes rapidly, then the cost for maintaining the group tree can become higher than query resolution costs. Section 2.5 presents Moara's dynamic adaptation mechanism that addresses this dilemma.

Second, Moara reduces network cost and response latency by short-circuiting the group trees, thus reducing the number of internal tree nodes that do not satisfy the predicate. For instance, in Figure 2.4, if node 010 does not satisfy the predicate but node 110 does, then the former can be eliminated from the tree by having 110 receive queries directly from the root. Section 2.6 describes how this reduces the bandwidth cost of aggregating a group with m nodes in a system of N nodes, from $O(m \log N)$ to $O(m)$.

Third, Moara efficiently resolves composite queries involving multiple groups by rewriting the predicate into a more manageable form, and then selecting a minimal set of groups to resolve the query. For example, an intersection query (CPU-Util, avg, (floor=F1 `and` cluster=C12)) is best resolved by sending the query to only one of the two groups - either (floor=F1) or (cluster=C12) - whichever is cheaper. This design decision of Moara is detailed in Section 2.7.

2.5 Dynamic Maintenance

Given a tree for a specific group, Moara reduces bandwidth cost by adaptively pruning out parts of the tree, while still guaranteeing correctness via *eventual completeness*. Eventual completeness is defined as follows - when the set of predicate-satisfying nodes as well as the underlying DHT overlay do not change for a sufficiently long time after a query injection, a query to the group will eventually return answers from all such nodes. For now, we assume that the dynamism in the system is only due to changes in the composition of the groups ("group churn"); we will describe how our system handles node and network reconfigurations (churn in system) later in Section 2.8.

To resolve queries efficiently, Moara could prune out the branches of the corresponding DHT tree that do not contain any nodes belonging to the group. However, to maintain completeness of the query resolution, Moara can perform such aggressive pruning only if it maintains up-to-date information at each node about the status of branches at that node. For groups with high churn in membership relative to the number of queries (e.g., CPU-Util $<$ 50), maintaining group status at each node for all its branches can consume high bandwidth - broadcasting queries system-wide may be cheaper. For relatively stable groups however (e.g., (sliceX = true) on PlanetLab), proactively maintaining the group trees can reduce bandwidth and response times. Instead of implementing either of these two extreme solution points, Moara uses a distributed adaptation mechanism that, at each node, tracks the queries in the system and group churn events from children for a group predicate and decides whether or not to spend any bandwidth to inform its parent about its status.

Basic Pruning Mechanism: Each Moara node maintains a binary local state variable *prune* for each group predicate. If *prune* for a predicate is true (PRUNE state), then the branch rooted at this node can be pruned from the DHT tree while querying for that predicate. Whenever a node goes from PRUNE to NO-PRUNE state, it sends a NO-PRUNE message to its parent; the reverse transition causes a PRUNE message to be sent. When the root or an internal node receives a query for this predicate, it will forward the query to only those of its children that are in NO-PRUNE state.

Note that it is incorrect for an internal node to set its state for a predicate to PRUNE based merely on whether it satisfies the predicate or not. One or more its descendants may satisfy the predicate, and hence the branch rooted at the node should continue to receive any queries for this predicate. Further, an internal or a leaf node should also consider the churn in the predicate satisfiability before setting the *prune* variable. For example, suppose the predicate is (CPU-Util $<$ 50) and a leaf node's utilization is fluctuating around 50% at a high rate. In this case, the leaf node will be setting and unsetting *prune* variable, leading to a large number of PRUNE/NO-PRUNE messages.

Due to the above reasons, we define the *prune* variable as a variable depending on two additional local state variables—*sat* and *update*. *sat* is a binary variable to track if the subtree rooted at this node should continue receiving queries for the predicate. Thus *sat* is set to 1 (SAT) if either the local node satisfies the predicate or any child node is in NO-PRUNE state.

update is a binary state variable that denotes whether the node will update its *prune* variable or not. So, when *update* = 1 (UPDATE state), the node will update the *prune* variable; but, when *update* = 0 (NO-UPDATE state), the node will cease to perform any updates to the *prune* variable irrespective of any changes in the local satisfiability, or any messages from its children. In other words, a node does not send any PRUNE or NO-PRUNE messages to its parent when it is in NO-UPDATE state. So, to ensure correct operation,

Procedure 1 Updating *sat* variable for each predicate

```
Initial Value: $sat \leftarrow 0$
Procedure Call: whenever there is a local attribute change or an update from
  any child
  $cnt \leftarrow 0$
  for each child do
    if there is no state associated with this child regarding the given predicate
    then
      // by default, a parent does not maintain any state on its children
      // Also, states can be garbage-collected after a period of inactivity
      $cnt$++
    else if child is in NO-PRUNE state then
      $cnt$++
    end if
  end for
  if the predicate is locally satisfied then
    $cnt$++
  end if
  if $cnt > 0$ then
    $sat = 1$
  else
    $sat = 0$
  end if
```

a node can move into NO-UPDATE state only after setting $prune = 0$. This guarantees that its parent will always send the queries for the predicate to this node. Formally, we maintain the following invariants:

$$\begin{aligned} update = 1 \text{ AND } sat = 1 &\Longrightarrow prune = 0 \\ update = 1 \text{ AND } sat = 0 &\Longrightarrow prune = 1 \\ update = 0 \ &\Longrightarrow prune = 0 \end{aligned}$$

The transition rules for the state machine at each node is illustrated in Figure 2.5. Note that a node sends a status update message to its parent whenever it moves from PRUNE to NO-PRUNE state or vice-versa. This state machine ensures the following invariant – *each node in the system performs at least one of the following: (a) sends status updates upwards to its parent, or (b) receives all queries from its parent.* This invariant suffices to guarantee eventual completeness because after the group stops changing, any node that satisfies the predicate will be in SAT state. Therefore, the node and its ancestors will all be in NO-PRUNE state, and thus the node will receive the next query.

Procedure 1, 2, and 3 show pseudo-code on how Moara evaluates each variable.

Adaptation Policy: To decide the transition rules for the *update* state variable, Moara employs an adaptation mechanism that allows different policies. Our goal is to use a policy that minimizes the overall message cost, i.e., sum of both update and query costs. In Moara, each node tracks the total number of *recent* queries and local changes it has seen (in the tree) - we will explain

Procedure 2 Updating *update* variable for each predicate

```
Initial Value: $update \leftarrow 0$ // in the beginning, a node receives every query
Procedure Call: whenever there is a new query received or a sat variable
  change
  if $2 \times q_n < c$ then
    $update \leftarrow 0$
  else if $2 \times q_n > c$ then
    $update \leftarrow 1$
  end if
```

Procedure 3 Updating *prune* variable for each predicate

```
Procedure Call: whenever there is a change in either update or sat
  if $update == 1 \&\& \ sat == 1$ then
    $prune \leftarrow 0$
  else if $update == 1 \&\& \ sat == 0$ then
    $prune \leftarrow 1$
  else if $update == 0$ then
    $prune \leftarrow 0$
  end if
```

recentness soon. Each node keeps two query counts - q_n, the number of queries recently received by the system while the node is in NO-SAT state, and q_s, the number of recent queries received by the system while it was in SAT state. The node also keeps track of the number of times the *sat* variable toggled between 0 and 1, denoted as c.

A node in NO-UPDATE state would exchange a total of $B_{NU} = 2 \times (q_n + q_s)$ messages with its parent (two per query), while a node in UPDATE state would exchange $B_{UP} = c + 2 \times q_s$ messages (one per change, and two per query). Thus, to minimize bandwidth, the transition rules are as follows: (1) a node in UPDATE state moves to NO-UPDATE if $B_{NU} < B_{UP}$, i.e., $2 \times q_n < c$; (2) a node in NO-UPDATE state moves to UPDATE if $B_{NU} > B_{UP}$, i.e., $2 \times q_n > c$. In order to avoid flip-flopping around the threshold, we could add in hysteresis, but our current design performs well without it.

Remembering Recent Events: Each node in Moara maintains a recent "window" of events for the counters mentioned above (q_n, q_s, and c). We use a window of k_{UPDATE} events if the node is in UPDATE state, and a window of $k_{NO-UPDATE}$ events if it is NO-UPDATE. In practice, we found that $k_{UPDATE} = 1$, $k_{NO-UPDATE} = 3$ works well, and we use these values in our implementation. For illustration purposes though, Figure 2.6 depicts the state machine for $k_{UPDATE} = k_{NO-UPDATE} = 1$. In this case, notice that whenever a node: (i) is in the PRUNE state and observes a change in *sat* ($c = 1, q_n = 0$), it switches to (NO-UPDATE, SAT); (ii) is in (NO-UPDATE, NO-SAT) and receives a query ($q_n = 1, c = 0$), it switches to UPDATE.

One corner issue with the above approach is that when a node is in the PRUNE state, it does not receive any more queries and thus cannot accurately track q_n. Note that this does not affect the correctness (i.e., eventual com-

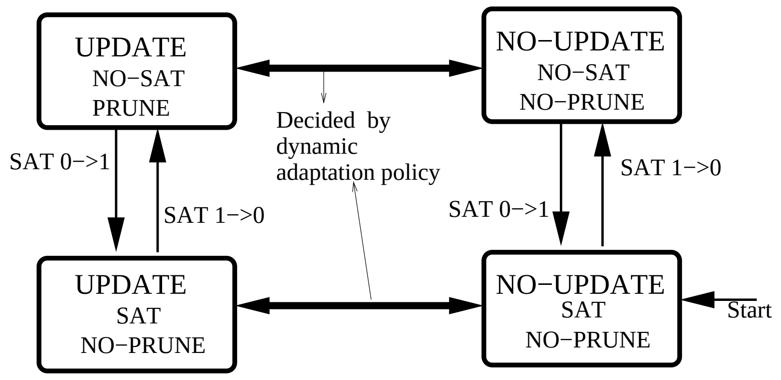

Figure 2.5: State machine for dynamic adaptation mechanism

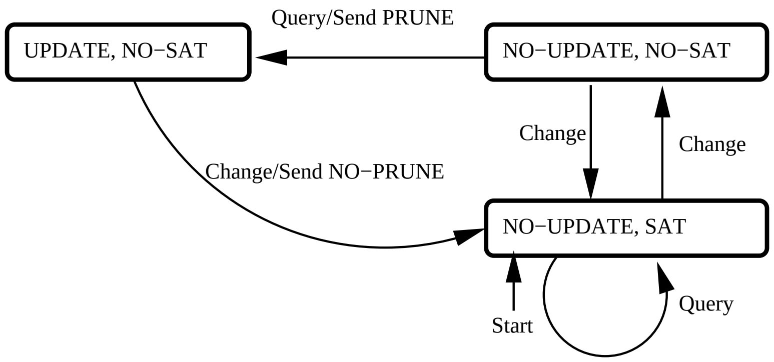

Figure 2.6: State changes at a Moara node for $k_{UPDATE} = 1$ and $k_{NO-UPDATE} = 1$. With these values, (UPDATE, SAT) is not reachable, thus is not shown here.

pleteness) of our protocol but may cause unnecessary status update messages. To address this, the root node of an aggregation tree in Moara assigns a sequence number for each query and sends that number piggybacked along with the queries. Thus, any node that receives a query with sequence number s is able to track q_n using the difference between s and its own last-seen query sequence number. In addition, our implementation suffers only minimally since we use small k_{UPDATE} values. For instance, for $k_{UPDATE} = 1$, when a node in (UPDATE, SAT) undergoes a local change, it immediately switches to NO-UPDATE, and sends no more messages to its parent.

State Maintenance: By default, each node does not maintain any state, which is considered as being in NO-UPDATE state. A node starts maintaining states only when a query arrives at the node. Without dynamic maintenance, merely maintaining pruned trees for a large number of predicates (e.g., a tree for each slice in the PlanetLab case or a tree for each job in the data center) could consume very high bandwidth in an aggregation system. With dynamic maintenance, pruning is proactively performed for only those predicates that are of interest at that time. Once queries stop, nodes in the aggregation tree start moving into NO-UPDATE state with any new updates from their children and hence stop sending any further updates to their parents.

We note that a node in NO-UPDATE state for a predicate can safely garbage-collect state information (e.g., predicate itself, recent events information, etc) for that predicate without causing any incorrectness in the query resolution. So, once a predicate goes out of interest, eventually no state is maintained at any node and no messages are exchanged between nodes for that predicate. Several policies for deciding when to garbage-collect state information are possible: we could 1) garbage-collect each predicate after a timeout expires, 2) keep only the last k predicates queried, 3) garbage-collect the least frequently queried predicate every time a new query arrives, etc. However, studying these policies is beyond the scope of this dissertation. We also note that we do not consider DHT maintenance overhead. In addition, note that global aggregation trees are implicit from the DHT routing and hence require no separate maintenance overhead.

Finally, since Moara maintains state information for each predicate, it could be more efficient if we aggregated different predicates. For example, predicates such as CPU-Util > 50, CPU-Util > 60, and CPU-Util > 70 could be aggregated as one predicate, CPU-Util > 50, so that Moara could maintain only one tree. This design choice requires careful study on the tradeoff between the state maintenance overhead and the bandwidth overhead incurred by combining different trees with the same attribute. This is outside of the scope of this dissertation, since we focus on the tradeoff of the bandwidth overhead based on the query rate and the group churn rate.

2.6 Separate Query Plane

Given a tree that contains m predicate-satisfying nodes, using the pruned DHT trees of the previous section may lead to $O(m \log N)$ additional nodes being involved in the tree. These extra nodes would typically be internal tree nodes that are forwarding queries down or responses up the tree, but which do not satisfy the predicate themselves. This section proposes modifications to the protocol described in Section 2.5 in order to reduce the traffic through these internal nodes.

Our idea is to bypass the internal nodes, thus creating a *separate query plane* which involves mostly nodes satisfying the predicate. This optimizes the tree that we built (Section 2.5) further by eliminating unnecessary internal nodes. This reduces the tree to contain only $O(m)$ nodes, and thus resolves queries with message costs independent of the number of nodes in the system. Note that this technique has similarities to adaptations of multicast trees (e.g., Scribe [17]), but Moara needs to address the challenging interplay between dynamic adaptation and this short-circuiting.

To realize a separate query plane, each node uses the states, constraints and transitions as described in Section 2.5. In addition, each node runs operations

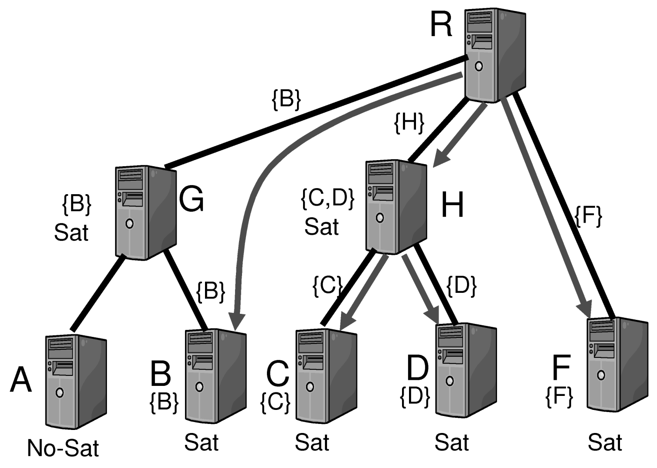

Figure 2.7: Separate Query Plane for *threshold*=1. We assume all nodes are in UPDATE mode. Each node's *qSet* is shown next to it, and *updateSet* on the link to its parent.

using two locally maintained sets: (i) *updateSet* is a list of nodes that it forwards to its parent; (ii) *qSet* is a list of children or descendant nodes, to which it forwards any received queries. We consider first, for ease of exposition, modified operations only for nodes in the UPDATE state. When a leaf node in UPDATE state begins to satisfy the tree predicate, it changes to SAT state as described in Section 2.5 and sets its *UpdateSet* to contain its ID. In addition, when sending a NO-PRUNE message to its parent, it also sends the *updateSet*. Each internal node in turn maintains its *qSet* as the union of the latest received *updateSet*s from all its children, adding its own ID (IP and port) if the tree predicate is satisfied locally. The leaf nodes do not need to maintain *qSet*s since they do not forward queries.

Finally, each internal node maintains its *updateSet* by continually monitoring if $|qSet| < threshold$, where *threshold* is a system parameter. If so, then *updateSet* is the same as *qSet*, otherwise *updateSet* contains a single element that is the node's own ID regardless of whether the predicate is satisfied locally or not. Whenever the *updateSet* changes at a node and is non-empty, it sends a NO-PRUNE message to its parent along with the new *updateSet* informing the change. Otherwise, it sends a PRUNE message.

The above operations are described assuming that all nodes are in UPDATE state. When a node is NO-UPDATE state, it maintains *qSet* and *updateSet* as described above, but does not send any updates to its parent. For correctness, a node moving from UPDATE to NO-UPDATE state sends its own ID along with the NO-PRUNE message to its parent so that it receives future queries.

If parameter *threshold*=1, the above mechanisms produce the pruned DHT tree described in Section 2.5, while *threshold* > 1 gives trees based on a separate query plane. This is because with *threshold*=1, an internal node that receives an *updateSet* from any of its children will pass along to its parent an *updateSet* containing its own ID, even if the predicate is not satisfied locally. However, with *threshold* > 1, the only internal nodes that do not satisfy the predicate locally but receive queries, are ones that are maintaining a *qSet* of size $\geq$ *threshold*. Such nodes are required to receive queries so that they can be forwarded to its descendants. However, the tree bypasses several other nodes that do not satisfy the predicate, thus obtaining bandwidth savings. Specifically, an internal node that has $|qSet| <$ *threshold* and does not satisfy the predicate, does not include its own ID in the *updateSet*, and thus does not receive queries.

Having a high value of *threshold* in the system bypasses several internal nodes in the tree. However, this comes at the expense of a higher update traffic since any *updateSet* changes need to be communicated to the parent. Figure 2.7 shows an example with *threshold*=1.

Adaptation and SQP: Our SQP design with *updateSet* and *qSet* variables at nodes, as described above, allow us to easily use the adaptation policy rules described in Section 2.5. In this case, q_n at a node is the number of queries received by the system when that node's *updateSet* does not contain its ID (similar to NO-SAT state) and q_s is the number of queries received at other times. The number of changes c is the number of changes to the *updateSet* variable. With these definitions, a node can use same adaptation policies as described in Section 2.5. One exception is the use of the query sequence number: for correct calculation of q_n at a bypassed node, each node piggybacks its last seen sequence number alongside all its status update messages to its parent.

Overhead analysis: For a group with m nodes, we analyze the overhead for forwarding a query in the separate query plane, assuming all nodes are in UPDATE state. First, notice that all leaf nodes in this tree satisfy the predicate - if some leaf did not, then it would be pruned out by the above rules. Second, the tree has the maximum links when all m predicate-satisfying nodes are at the leaves of this tree. This means that since *threshold* > 1, no internal node (other than the root node) in the tree has fewer than 2 children - if it did, it would be bypassed by the above rules. However, no tree with m leaves and internal node degree > 2 has more than m internal nodes. Thus, the total number of nodes, other than the root, receiving the query is $\leq 2 \cdot m = O(m)$, independent of system size.

2.7 Composite Queries

So far, we have described how to build and maintain a *single* tree corresponding to one simple predicate. We now describe how a query with a composite predicate is satisfied. Specifically, we first expand on the multiple possible trees, one tree per simple predicate in the composite query, that such a query entails (Section 2.7.1). Then, we explain how Moara plans a given query (Section 2.7.2), and how it selects a low-cost groups of nodes to execute a given composite query (Section 2.7.3).

2.7.1 Maintaining Multiple Trees

Section 2.5 explains the maintenance of trees for simple predicates, starting from the time a predicate is first encountered. If this predicate does not reappear again in subsequent queries in the system, then all nodes in the tree will eventually move to NO-UPDATE state (due to group churn events), and thus there will be no load, either query or update, along the tree. Thus, Moara trees become silent and incurs zero bandwidth cost if not used, obviating the need to explicitly delete trees for simple predicates. Furthermore, Moara does not maintain trees for composite queries, since these might be exponentially large in number - instead, it decides which simple predicate trees (existing or not) will be selected to execute a given composite query. This decision process is described next.

2.7.2 Composite Query Planning

Consider the following composite query: "find the average free memory across machines where service X and Apache are running". Suppose we have one group tree for (ServiceX=true) and another tree for (Apache=true). A naïve way to resolve the query would be to query both trees in parallel. However, we observe that bandwidth can be saved, without compromising completeness of answers, by (1) sending the query to any *one* of the trees (because it is an intersection query), and (2) choosing the tree that incurs a lower query cost.

Based on this observation, Moara answers arbitrary nested queries involving `and` and `or` boolean expressions across simple predicates by selecting a small *cover*. A cover for a given composite query Q is defined as a set of groups (selected from among simple predicates inside Q) which together contain all nodes that satisfy the composite predicate in Q. Thus, we only need to send Q to a cover to obtain a complete answer.

We can compute a cover for a query Q by exploring the boolean expression structure recursively as follows:

- cover(Q="A") = {A} if A is a predefined group.
- cover(Q="A `or` B") = cover(A) $\cup$ cover(B).
- cover(Q="A `and` B") = cover(A), cover(B), or (cover(A) $\cup$ cover(B)).

For example, for a query with expression ((A `and` B) `or` C), the above rules derive {A,C}, {B,C}, and {A,B,C} as possible covers. We call such covers as *structural* covers since we infer them from the structure of the boolean expression.

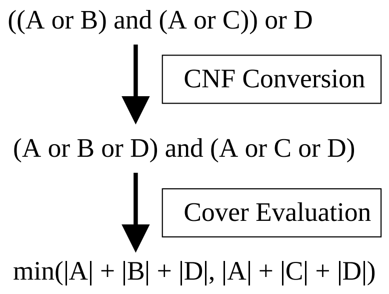
((A or B) and (A or C)) or D
CNF Conversion
(A or B or D) and (A or C or D)
Cover Evaluation
min(|A| + |B| + |D|, |A| + |C| + |D|)

Figure 2.8: Example query processing

Once the query originating node calculates the cover for a given query Q, the composite query is forwarded to the roots of trees corresponding to each group in the cover, the answers from these trees are aggregated, and finally returned to the querying node. Notice that it is possible for some node(s) to receive multiple copies of the query, if they are present in multiple trees which appear in the cover for Q. Such nodes reply with the attribute value to only *one* of the trees they are present in, eliminating duplicate answers. This requires nodes to remember the query ids (based on sender IP and sequence number). Such information is cached for 5 minutes in our Moara implementation.

To further save on bandwidth, we would like to select a low-cost cover. This is done by minimizing both the number of groups in the selected cover, as well as the total cost of querying this cover. We explore below three ways of deriving a low-cost cover: (1) *structural optimizations*, which rewrite the nested query to select a low-cost structural cover consisting of simple predicates that already appear within the query, (2) *estimates of query costs* for individual trees, and (3) *semantic optimizations*, which take into account semantic information obtained from users or query attributes.

2.7.3 Query Optimization: Finding Low-Cost Covers

Given a composite query, Moara first transforms it into a Conjunctive Normal Form (CNF) expression using distributive laws of `and` and `or` operators. A CNF form is a two level expression of `and`'s across a series of `or` terms.

It is important to notice that in the CNF form of a composite predicate for query Q, each series of `or` terms is a possible cover - this is due to the same reason as our intersection optimization explained earlier. Thus, if Moara can evaluate the query cost of each of these structural covers (as a sum of the query costs for all sets in the cover), then it can select the minimal cost cover for executing the query Q. We will describe query cost calculation soon, but before that we give an example of the query rewriting as well as proof sketch of why the CNF form gives the *minimal-cost* cover for a composite predicate.

Figure 2.8 shows an example transformation. Consider a query targeting ((A `or` B) `and` (A `or` C)) `or` D. Moara first transforms the expression to the

Semantic information	(A `or` B)	(A `and` B)
$A \cap B = \phi$	{A,B}	{}
$A \cap B \neq \phi$ and $A \supseteq B$ and $A \subseteq B$ $\Rightarrow$ (A=B)	{A}	{A}
$A \cap B \neq \phi$ and $A \supseteq B$ and $A \nsubseteq B$	{A}	{B}
$A \cap B \neq \phi$ and $A \nsupseteq B$ and $A \subseteq B$	{B}	{A}
$A \cap B \neq \phi$ and $A \nsupseteq B$ and $A \nsubseteq B$	{A,B}	{A},{B}

Table 2.2: Semantic info to reduce cover sizes

Relation between pair of groups	Description	Example pair of groups
Intersection (without inclusion)	Two groups intersect properly	(CPU-Util < 50), (CPU-Util > 20)
Discontinuous Intersection	The intersection is not continuous	(CPU-Util < 50), (CPU-Util $\neq 20$)
Equivalence	Two groups are identical	(CPU-Util < 50), (CPU-Util < 50)
Inclusion	One group is a subset of another	(CPU-Util < 50), (CPU-Util < 20)
Disjointedness	Two groups do not intersect	(CPU-Util < 50), (CPU-Util > 80)

Table 2.3: Defining operators of groups that allow relation inference between two groups

equivalent CNF: (A `or` B `or` D) `and` (A `or` C `or` D). Moara chooses one cover between the two structural covers - either $\{A, B, D\}$ or $\{A, C, D\}$, whichever has a lower cost.

If query cost estimates for individual groups are up-to-date and available at the tree roots, we can prove by contradiction that our structural optimizations produce a cover that is minimum in cost. Suppose that the given CNF expression is $E = A_1$ `and` A_2 `and` $\ldots$ `and` A_n, where each term A_i is an `or` of positive literals and hence a structural cover for E. Assume the contrary, i.e., suppose there exists a structural cover C with a lower cost than our covers. In each term A_i of expression E, if we substitute the literals from set C with 0, the expression should evaluate to 0 (since C is a structural cover). However, since A_i's are `and`-ed, there has to be *some* A_j that evaluates to 0 in this substitution. Thus, all groups in this A_j have to be a part of C. However, this is a contradiction since A_j is a cover with cost no more than C.

Estimating Query Costs for Trees: In order to enable low-cost cover calculation, the root node of each tree for a simple predicate continually maintains the query cost for that tree. The query cost is fetched by the querying node and used in the low-cost cover calculation described above. Within the tree, the cost for each query is simply $2 \times np$, where np is the number of nodes in NO-PRUNE state. The values of np are aggregated continually up the tree. Each internal node stores this count for its own subtree, modifies the count according to its own state, and piggybacks this information atop all updates and query responses to its parents. Although this lazy updating of the counts means the query costs may be stale at times, this only affects communication overhead, but not the correctness of the response.

Using Semantic Optimizations: If semantic information is available about the groups, then Moara can further optimize the communication costs by choosing a better cover. We explore two kinds of semantic information in our system: (i) information from description of the group, and (ii) user supplied semantic information. For example, consider two groups A and B defined as follows: $A = \{\text{nodes with memory} < 2G\}$ and $B = \{\text{nodes with memory} < 1G\}$. Then, we can infer from these definitions that $B \subseteq A$. In Table 2.3, we list relations between two groups that Moara infers by analyzing the operations that define those pair of groups. Once we have semantic information either inferred from the description of the groups or supplied by a user, Moara applies the optimizations detailed in Table 2.2 to obtain a low-cost cover. As another instance, Moara implicitly supports `not` operator by observing complement relations in the specified groups (the last row in Table 2.3), and the following optimizations:

- (A `or` B) `and` (A `or` C) = A, if C = `not` (B)
- (A `or` C) `and` B = A `and` B, if C = `not` (B)
- (A `or` B) `and` C = A `and` `not` (B), if C = `not` (B)

2.8 Implementation and Evaluation

We have built a prototype of Moara using SDIMS [109] and FreePastry [93]. All other Moara protocols, described in Section 2.4 through Section 2.7, are built atop these systems. Here, we discuss our implementation details and evaluation methodology.

Moara Front-End: The Moara front-end is a client-side interface of Moara. It includes an interactive shell, a query parser, and a query optimizer. Through the interactive shell, a user can submit SQL-like aggregation queries to Moara. The query parser parses the queries, and the query optimizer determines the groups that need to be queried through the algorithm described in Section 2.7. Once the front-end determines the groups to be queried, it generates a *sub-query* for each group. Each sub-query is resolved exactly the same way as a normal query, except that the front-end waits until it receives all the results from sub-queries, aggregates the results returned by the sub-queries, and returns the final aggregate to the user.

Reconfigurations: To handle reconfigurations, we leverage the underlying FreePastry mechanism for failure detection and neighbor set repair. When a node gets a new parent for a predicate, it sends its current state information (e.g., *updateSet*) for that predicate to the new parent. Also, when a node is waiting for a response from a child and if the underlying DHT notifies that the child has failed or is not reachable, then the node will proceed assuming a NULL response from the child. In addition to this, Moara also implements a time-out mechanism (in waiting for a child's reply to a query) to ensure that all queries are responded to independent of FreePastry's timeout values for failure detection.

Evaluation Environments: We use simulation, Emulab, and PlanetLab, and choose a suitable environment to evaluate each of our design choices. We use simulation exclusively for measuring bandwidth consumption in a large-scale environment. We use Emulab and PlanetLab to mainly measure the latency in realistic environments, namely, a medium-scale data center (Emulab) and a wide-area infrastructure (PlanetLab).

For each design choice (group-based aggregation, dynamic maintenance, separate query plane, and composite query processor), we choose the evaluation environments that are most suitable. First, we evaluate group-based aggregation on Emulab and PlanetLab, since group-based aggregation is designed to reduce both latency and bandwidth consumption. Second, we evaluate dynamic maintenance and separate query plane using simulation, since both mechanisms are designed for bandwidth optimization and have wide choices of parameters. However, we evaluate the separate query plane on Emulab as well to measure the latency. Lastly, we evaluate our composite query processor on Emulab, since it only affects latency.

Workload: The workload is characterized by two factors - group churn rate and query rate. First, since a group is defined over a particular attribute, the group churn rate depends on how dynamic the attribute is (e.g., a group of (OS = Linux) is likely to be static, while a group of (CPU-util $< 60\%$) is likely to be dynamic). Second, the query rate depends on the usage of Moara and is expected to vary widely. For example, a data center operator might typically query a group once an hour on a day, but several times a minute on days with high workloads or unscheduled downtimes. Thus, we parameterize these factors and present the performance of Moara over the parameter range.

2.8.1 Simulation Results

We perform simulation experiments to measure the bandwidth overhead of Moara's dynamic tree maintenance and separate query plane. Our simulations are performed with the FreePastry simulator environment, simulating up to 16,384 nodes. Each node maintains an attribute A with value $\in \{0, 1\}$. All queries are simple queries for (A, SUM, $A = 1$), which counts the number of nodes where A is set to 1.

Dynamic Maintenance: To study the dynamic maintenance mechanism under different workload types, we stress the system by injecting two types of events - query events and group churn events - at different ratios. For example, a query:churn ratio of 0:500 represents an extreme type of workload where there

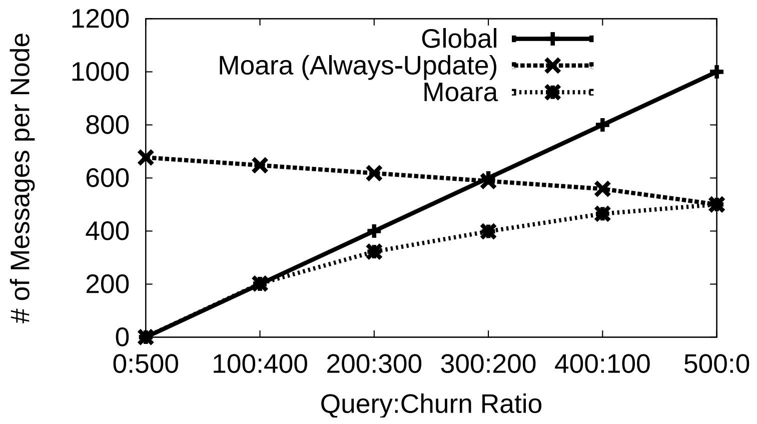

Figure 2.9: Bandwidth usage with various query-to-churn ratios

is high group churn, but no queries at all. On the other hand, the query:churn ratio of 500:0 represents the other extreme where there is high query rate, but no group churn. Each group churn event selects m nodes at random, and toggles the value of their attribute A. The value of m determines the "burst size" of attribute churn. We fix the total number of events to 500, and randomly inject query or group churn events at the chosen ratio. All data points are averaged over 3 runs.

Figure 2.9 shows the average number of messages per node in Moara under various query:churn ratios, in a system of 10,000 nodes with m = 2000-sized group churn events. In addition to Moara, we also plot the number of messages generated by two other static approaches that lie at the opposing extremes. These are: 1) the *Global* approach, where no group trees are maintained and queries are sent to all the nodes on the DHT trees, and 2) *Moara (Always-Update)* approach, where a tree is aggressively maintained by having each child send an update to its parent on each attribute churn event.

The Global approach is inexpensive when there are fewer queries in the system, since it avoids the overhead of tree maintenance. On the other hand, with a high-query:low-churn ratio, Moara (Always-Update) performs well because it always maintains group trees and hence incurs lower traffic than Global approach. The plots show that Moara meets or lowers the message overhead in comparison to either of these extreme design choices, at all values of query:churn ratios. When group churn is high, Moara suppresses attribute churn events from propagating to other nodes. With more queries than group churn events, Moara reduces query cost by maintaining trees aggressively. Thus, Moara is able to adapt to various workload patterns. In the above experiment, we use k_{UPDATE}=1 and $k_{NO-UPDATE}$=3. Here, we study the sensitivity of the performance for different values for these knobs. Figure 2.10 plots the average number of messages per node in a system of 500 Moara nodes under a range of query:churn ratios for different threshold values. Although we have tried a wide range of values

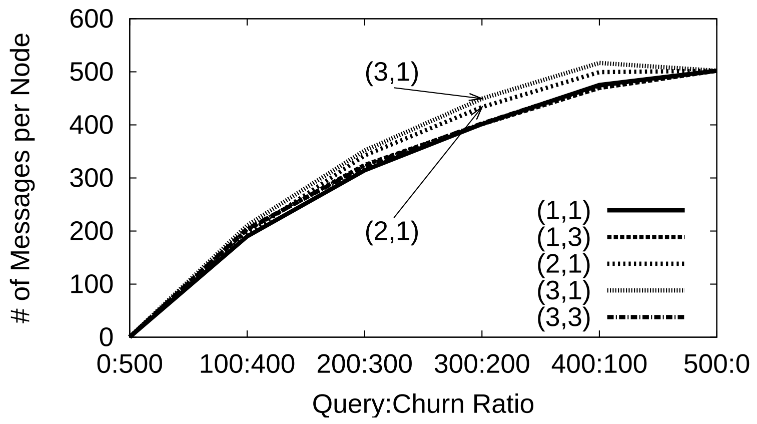

Figure 2.10: Bandwidth usage with various (k_{UPDATE}, $k_{NO-UPDATE}$). Although we have tried a wide range of values in our simulation, we only show a few representative pairs for clarity of presentation.

(each up to 10), we only show a few representative pairs that are sufficient to show the conclusion, in order to prevent the plot from being too crowded. When there are very few query events in the system (compared to churn events), different threshold values perform similarly. However, when the number of queries is high (e.g., # Queries=400), large k_{UPDATE} values (e.g., $(3,1)$ and $(2,1)$) coupled with small $k_{NO-UPDATE}$ values lead to slightly more overall messages. This is because with larger k_{UPDATE} and smaller $k_{NO-UPDATE}$, more Moara nodes stay in UPDATE state - thus, each child updates its parent more often, even with small churn rates. Overall, we observe that the sensitivity of the performance to different thresholds is very small. For all other experiments, we use the default values of k_{UPDATE}=1 and $k_{NO-UPDATE}$=3.

Separate Query Plane: In Figure 2.11, we plot the query cost against the number of nodes in the system for different threshold values and different group sizes. Note that the *threshold* value of 1 implies the absence of a separate query plane, while higher threshold values create a separate query plane (refer to Section 2.6). For this experiment, we do not introduce any group churn during the experiment. We perform 1,000 queries and compute the average of the query cost. Even though there is no group churn, there are updates sent by nodes to their parents as they move into UPDATE state with the first query message. We count those messages as the update cost.

Figure 2.11 shows that without the separate query plane (*threshold*=1), the query cost increases logarithmically as the total system size is raised. However, while maintaining a separate query plane (*threshold*>1), the query cost reaches a constant value and stays flat, independent of the number of nodes in the system. While increasing the value of *threshold* decreases query cost, it can lead to more update messages as discussed in Section 2.6. In Figure 2.12, we plot the query costs for different *threshold* values as a percentage of the query

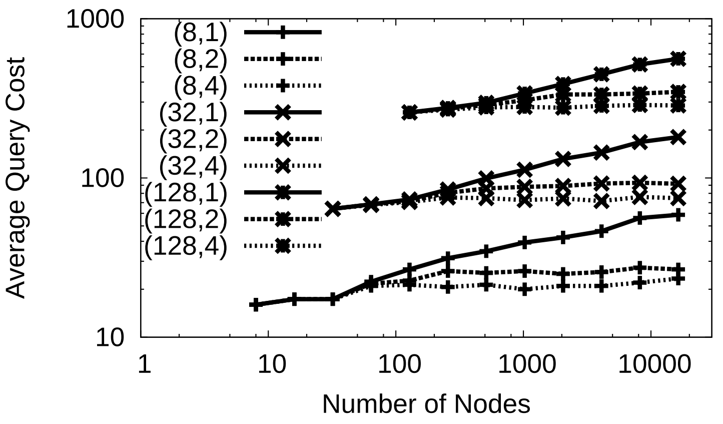

Figure 2.11: Bandwidth usage with (threshold > 1) and without the separate query plane (threshold=1) for different group sizes. Each line represents a (group size, threshold) pair

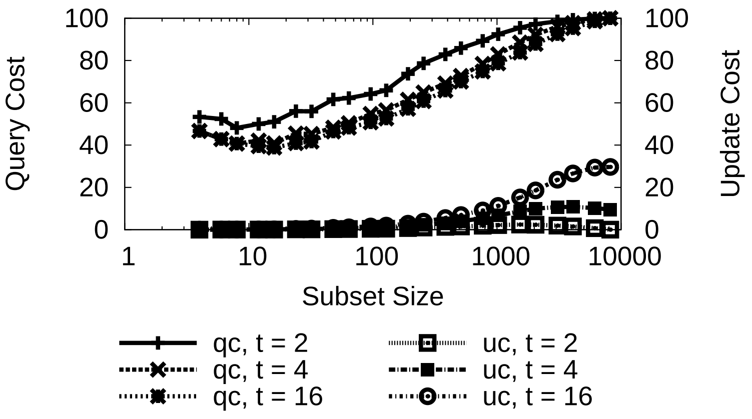

Figure 2.12: Query costs (qc) and update costs (uc) of the separate query plane in a 8192-node system

cost for *threshold*=1 and also plot the percentage increase in the update costs in comparison to *threshold*=1. From these two plots, we observe that (1) with small groups and large total nodes (e.g., 8192 total nodes with group size=8 or 32), using a query plane saves more than 50% bandwidth in query costs, and (2) while using a higher value of threshold does reduce bandwidth, the savings are marginal beyond a threshold of 2 and can incur higher update costs at large group sizes.

2.8.2 Emulab Experiments

In this section, we study both the latency and communication overhead of Moara under a real deployment scenario in Emulab, that emulates a medium-scale data center. Specifically, we evaluate three different workloads. First, we study performance of Moara when querying groups of static attributes (e.g., OS =

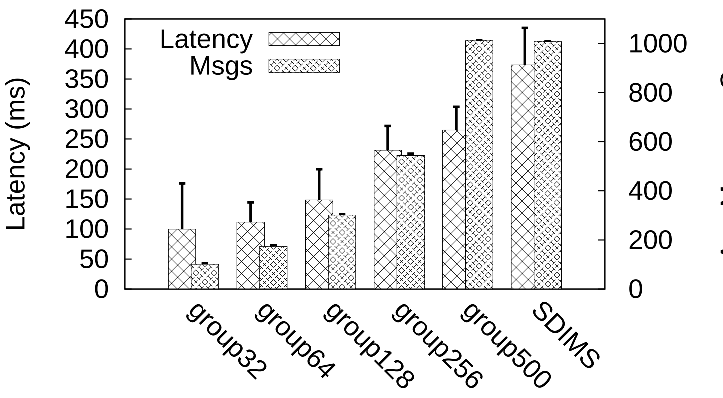

Figure 2.13: Latency and bandwidth usage with static groups

Linux). We vary the size of groups and show the benefits of using Moara. Second, we study Moara with groups defined over dynamic attributes (e.g., CPU-util $< 60\%$). We stress Moara by varying the frequency of changes. Third, we study composite queries with varying numbers of groups per query.

Methodology: We create a network of 50 machines on a 100 Mbps LAN and instantiate 10 instances of Moara on each machine, thus emulating a 500 node Moara system. Each experimental run is started with one bootstrap node, followed by a batch of 100 new instances joining after intervals of 10 seconds each. After the last join, we wait an additional 5 minutes to warm up before initiating queries and group churn from a Moara node. Since we are mainly interested in per-query latency and bandwidth consumption, we fix the query rate and repeat the same query multiple times. As previously, each node maintains one binary attribute A. Our default query is a count, providing the number of nodes with A=1. All data points are the average of 3 runs.

Static Groups: Figure 2.13 compares the performance of Moara (with separate query plane) w.r.t. both latency and bandwidth. We vary the group sizes and query 100 times for each experiment. In addition, we compare this performance against an approach where a single global tree is used system-wide - this is labelled as the *SDIMS* approach in the plot. As we can see from the figure, Moara's latency and bandwidth scale with the size of the group. The savings are the most significant for small groups (e.g., set32 which has 32 nodes), where the savings compared to the SDIMS approach are up to 4X in latency and 10X in bandwidth. The latency is reduced due to the use of separate query plane because of short-circuiting long chains of intermediate nodes.

Dynamic Groups: We study the effect of group churn due to attribute-value changes at individual nodes. We considered a group of 100 nodes, with group churn controlled by two parameters *churn* and *interval*. Every *interval* seconds, we randomly select *churn* nodes in the group to leave, and *churn* nodes outside the group to join.

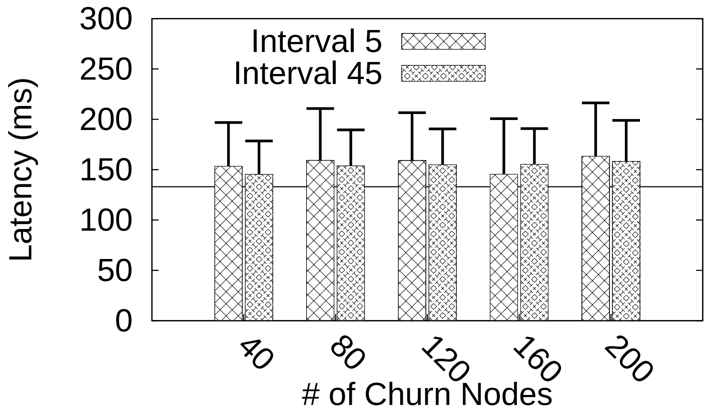

Figure 2.14: Average latency of dynamically changing groups. The horizontal line shows the average latency with a static group of the same size.

Figure 2.14 shows the effect on query latency, of different *churn* values (x-axis) for two different *interval* values. Queries are inserted at the rate of one query per second, and the data points are averages of 100 queries per run. The plot shows that Moara's query latency is not affected significantly by group churn - (1) even when we increase the group churn rate by a 9-fold factor from *Interval*=45 to *interval*=5, Moara experiences only a small increase in latency, and (2) the latency stays low, and around 150 ms even when the entire group membership changes every 5 seconds (*interval*=5, *churn*=200).

Figure 2.15 provides an insight into the workings of Moara under the above workload, for *interval*=5, *churn*=160. Notice that the spikes in query latency occur once every 5 seconds, around the time that the group churn batch occurs. However, notice that (1) the peak latency stays within 300 ms, and (2) Moara query latency stabilizes very quickly after each group churn batch, typically within 1-2 seconds. These plots thus show that Moara is highly resilient to dynamism due to rapidly occurring attribute-value changes.

Composite Queries: The experiments so far have focused on single groups in Moara. Here, we microbenchmark the performance of Moara on composite queries. Assuming $S_1, S_2, \ldots, S_n$ are simple single predicate groups, we study three types of composite queries: (1) Intersection queries of the form $S_1 \cap S_2 \cap \ldots \cap S_n$, for different values of n; (2) Union queries of the form $S_1 \cup S_2 \cup \ldots \cup S_n$, for different values of n; and (3) Complex queries, which are structured as $T_1 \cap T_2 \cap \ldots \cap T_m$, where each T_i is a union of multiple groups. These experiments suffice to characterize Moara's performance since the query optimization reduces all query expressions to one of the three. Each basic group S_i consists of 50 nodes selected at random. The complex expression we use[1] is $T_1 \cap T_2 \cap T_3$, and each T_i is a union of n basic groups for different values of n. Figure 2.16

[1]We found that the number of T_i's has little effect on latency because Moara queries only one of all T_i's.

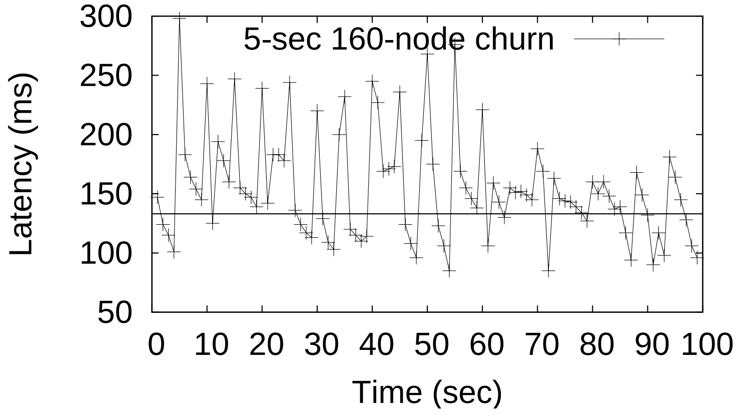

Figure 2.15: Latency over time with a dynamically changing group. The horizontal line shows the average latency with a static group of the same size.

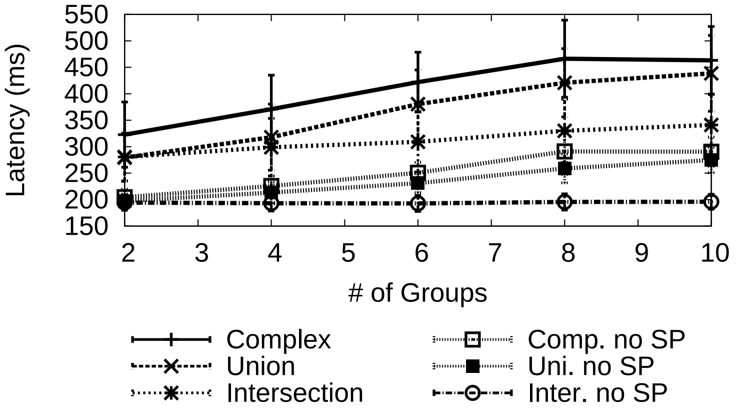

Figure 2.16: Latency with composite queries

plots the latency for above three types of queries with different values of n. For composite queries, recall that Moara first sends size probes to root nodes of group trees, in order to make a query optimization decision. Thus, we plot not only the total latency of a Moara query, but also the latency excluding the time to finish the size probes. Each data point is averaged over 300 queries.

First, notice that the average completion times of all queries, including queries with up to 10 groups, is less than 500 *ms*. For intersection queries, the completion times excluding time for size probes (plot line “Inter. no SP”) do not depend on the size of the expression. This is because Moara selects only one of these groups to propagate the query. Although size probes are sent in parallel, the latency for size probes increases slightly since Moara waits until the slowest probe response arrives. For union queries, the total completion time of a query rises gradually with the size of the expression, as Moara needs to contact all groups (two “Union” plots). Finally, the completion time for complex queries is only slightly more than that of union queries, since Moara’s query

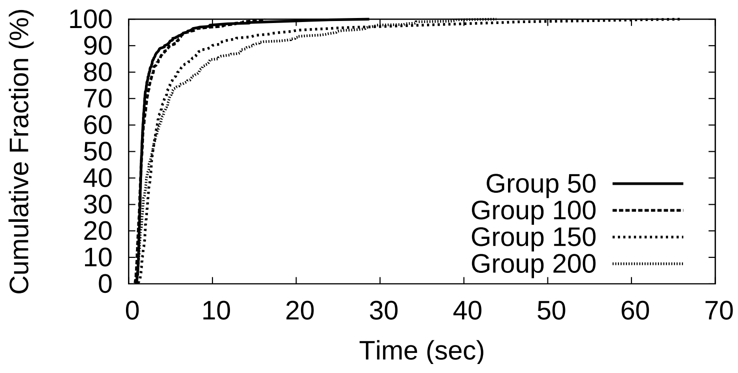

Figure 2.17: PlanetLab Latency

optimization selects only one of T_i's. The additional latency is caused by two factors: (a) the time taken for size probes is higher as we have to query the sizes for larger number of groups, and (b) a complex set expression adds more overhead at each node, because each node evaluates the entire complex expression to determine if it satisfies it or not (this step could be further optimized).

2.8.3 PlanetLab Experiments

Methodology: We deploy Moara atop 200 PlanetLab nodes, which span several continents. Each PlanetLab node runs one instance of Moara. The instances are started sequentially, the system is given 5 minutes to warm up, and then a series of queries is injected from a Moara front-end running on a local machine. In order to study the behavior of Moara's query latency in-depth, we perform experiments on only one group at a time, but for different sizes of this group. Each experiment involves a total of 500 queries injected 5 seconds apart. All plotted data points are the average of 3 runs. We do not timeout on queries, in order to obtain complete answers.

Query Response Latency: Figure 2.17 plots the cumulative fraction of replies received as a function of time since query injection, on four different-sized groups. The plot shows the responsiveness of Moara in a wide-area setting - even with as many as 100 nodes in the group, the median answer is received back within 1-2 seconds, while 90% of the answers are received within 5 seconds.

Moara versus Centralized Aggregation: Figure 2.18 compares Moara against a centralized approach which maintains no trees but has the Moara front-end directly query all nodes in parallel regardless of whether they satisfy the given predicate or not (labelled "Central"). The response for a query from this centralized aggregator is considered complete when the centralized aggregator has received a response from every node regarding the query. The figure plots the cumulative fraction of replies received as a function of time since query injection. This plot illustrates that the comparison between the centralized aggregator and Moara is intuitively akin to the comparison of "the tortoise and

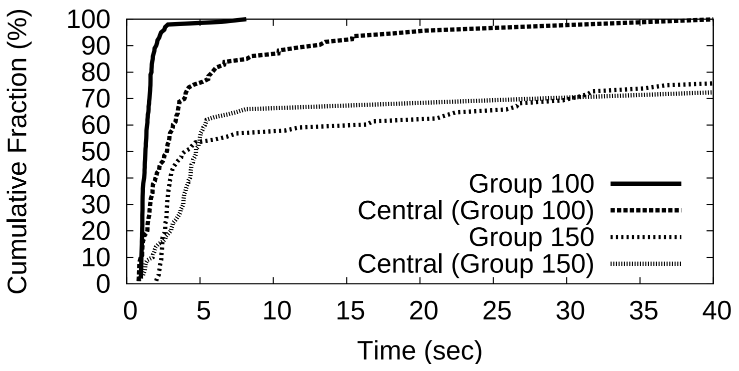

Figure 2.18: Moara vs. Centralized Aggregator

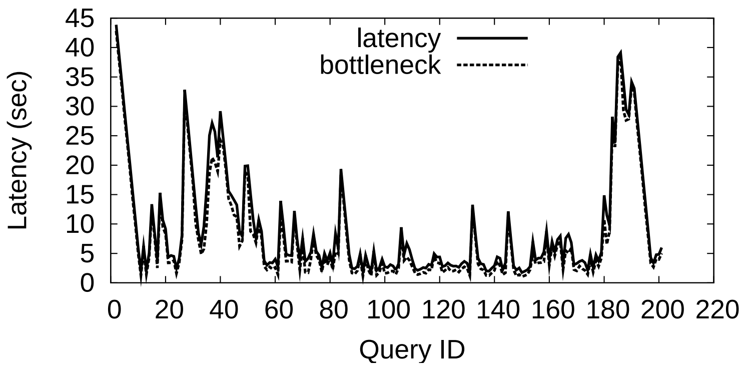

Figure 2.19: PlanetLab Bottleneck Latency

the hare". In other words, for both groups of size 100 and 150, we notice that the centralized aggregator obtains initial replies faster than Moara, but then it slows down waiting for the remainder of the query answers from nodes.

Figure 2.19 further explains why Moara's overall completion time is shorter than the centralized aggregator with smaller groups. We plot the total completion latency for a Moara query in a 200-node group, along with the latency on a single bottleneck link in the Moara tree. This bottleneck link is obtained via offline analysis, and by picking the largest round-trip-time among all parent-child pairs in the tree. This plot shows that it is a *single* bottleneck link that contributes to the latency of Moara. Moara is faster overall in obtaining a large fraction of replies, because it avoids bottlenecks that are not part of the queried group. In comparison, the centralized aggregator is subject to being slowed down by *all* nodes that suffer from bottlenecks, mainly the slowest bottleneck.

2.9 Related Work

Management solutions have existed for a long time in the areas of operating systems [63], the Internet [97, 99], and distributed systems, e.g., in AFS [41], Amoeba [101], Condor [64], Globus [32], and others [69]. However, in today's context, dealing with complex, dynamic, and large-scale systems presents a new set of challenges. This has been earmarked by the CRA [20] and several leading industry researchers [98] as a grand challenge for the next few years. This is especially relevant in systems such as PlanetLab [81], data centers and Grids [28, 27, 87], as well as emerging systems, such as NSF GENI [76].

The management operations of interest to us include distributed monitoring, aggregation, and querying. Most commercial tools for these operations such as HP OpenView, IBM Tivoli, and CA's Unicenter are centralized in nature. On PlanetLab there are several management tools in use, such as vxargs, PSSH, Stork, CoTop, and others [84]. CoMon [80] collects 5 minute CPU, memory, and network data, with aggregation per-slice. None of the mentioned tools addresses scalability and expressive queries for distributed systems.

Several academic efforts such as Ganglia [70], PIER [43], Astrolabe [88], and MON [62] propose distributed systems for aggregating data in large systems, but they either do not support complex queries as a first-class problem, or are too expensive for complex queries.

Vanilla DHTs by themselves, e.g., Pastry [93], Chord [100], Tapestry [111], Kademlia [71], Kelips [36], etc., do not support complex queries. PIER [43] is a distributed database system supporting recursive SQL-style queries on the several tables distributed across nodes in a large system. PIER does not leverage in-network aggregation, the querying node gets all records and performs aggregation locally.

TinyDB [66] is a querying and aggregation system for sensor networks whose goals bear similarities to ours, and it supports complex queries. TinyDB's applicability is limited to wireless sensor networks with multi-hop connectivity; utilizing its techniques in our setting would contact a large number of nodes for any complex query. Also in sensor networks, Synopsis diffusion techniques aggregate data across a DAG without double-counting [75]. Yao *et al.* have also proposed query processing in sensor networks via sketches and other techniques [102, 110].

Existing hierarchical solutions for data aggregation include, for example, Astrolabe [88] and Ganglia [70]. Our goals are most similar to that of Astrolabe, which is a querying system that supports SQL-style aggregation queries on geographically distributed clusters. Astrolabe provides a generic aggregation abstraction that also forms the basis for SDIMS [109]; whereas SDIMS leverages DHTs to construct multiple trees and scale with both nodes and the number of metrics, Astrolabe uses a single static tree and hence has limited scalability with the number of metrics. Astrolabe implements this by building a MIB that is distributed and is structured as a zone-based hierarchy, continuously updating itself via gossiping. Unlike Astrolabe, Moara support *on-demand* actions [62] based on individual queries, and low latency combined with good accuracy for these queries. MON [62] uses on-demand overlays as well, but does not support expressive queries. Ganglia [70] uses a single hierarchical tree and local multicasts to collect the data; they focus mainly on collecting the entire data to a single central site without performing any in-network computation on the data.

NetProfiler [79] forms groups and aggregates metrics over those groups. Groups are defined by a particular characteristic: machines running Windows, machines in a subnet 128.83.X.X, etc. The scenarios considered by NetProfiler are covered by Moara.

Huebsch *et al.* [44] present a way to optimize global aggregation queries by sharing computations. They propose a method drawn from linear algebra and other heuristics to discover k queries which can be used to answer n queries where $k << n$. Moara shares a similar goal of optimizing aggregation queries, but Moara's optimization is for multiple subset aggregation trees rather than a global aggregation tree.

The bulk of work done under optimizing continuous queries over streaming data, e.g., [30, 58, 103] addresses centralized solutions that support more dynamism than a database. Our focus is on on-demand, one-time queries.

Algorithmic work on collecting aggregate properties in large distributed systems includes estimating the system size [57, 40] and gossip-based aggregation [38, 49, 51]. However, unlike Moara, none of this work addresses expressive queries.

Finally, providing strong consistency guarantees on aggregated results in large distributed systems is a hard problem. Similarly to previous aggregation systems such as Astrolabe [88] and SDIMS [109], Moara focuses on providing high availability and guarantees only weak eventual consistency guarantees. As stated by Narayanan *et al.* [74] and Bawa *et al.* [6], it is impossible to guarantee that a read-only query sees a snapshot at a single time across the entire system (called *snapshot validity*). Thus, PIER [43] guarantees "dilated reachable snapshot", which is a snapshot that consists of reachable nodes at the dilated time. Seaweed [74] guarantees that the aggregated result includes all the nodes that were available for sufficient time to execute the query.

2.10 Summary

We have presented the motivation, use cases, design, and evaluation of Moara, an on-demand group-based monitoring system. Moara addresses the challenges of scale and dynamism by implementing three techniques — dynamic group tree maintenance, separate query plane construction, and a query optimization for complex queries. Our evaluation using simulations and deployments has shown that Moara is effective in accurately answering single-group queries as well as multi-group complex queries within hundreds of milliseconds across hundreds of nodes, and with low per-node bandwidth consumption.

Chapter 3

ISS: Intermediate Storage System

This chapter presents the motivation, requirements, design, and evaluation of ISS (Intermediate Storage System). ISS solves the problem of providing on-demand non-interfering replication of intermediate data as it is generated. It addresses the challenge of scale in terms of the number of machines and the amount of intermediate data. It addresses the challenge of dynamism in bandwidth availability. We detail these challenges and how ISS addresses them in this chapter.

3.1 Motivation

Dataflow programming frameworks such as MapReduce [23], Dryad [46], Pig [77], and Hive [29] are gaining popularity for large-scale parallel data processing. For example, organizations such as A9.com, AOL, Facebook, The New York Times, Yahoo!, and many others use Hadoop, an open-source implementation of MapReduce, for various data processing needs [85]. Dryad is currently deployed as part of Microsoft's AdCenter log processing [25]. These frameworks run in a single-site data center/cloud, which may be internally hierarchical, e.g., organized as racks.

In general, a dataflow program consists of multiple stages of computation and a set of communication patterns that connect these stages. For example, Figure 3.1 shows an example dataflow graph of a Pig program. A Pig program is compiled into a sequence of MapReduce jobs, thus it consists of multiple Map and Reduce stages. The communication pattern is either all-to-all (between a Map stage and the next Reduce stage) or one-to-one (between a Reduce stage and the next Map stage). Dryad allows more flexible dataflow graphs, though we do not show an example in this dissertation.

Thus, one common characteristic of all the dataflow programming frameworks is the existence of *intermediate data* produced as an output from one stage and used as an input for the next stage. On one hand, this intermediate data shares some similarities with the intermediate data from traditional file systems (e.g., temporary .o files) – it is short-lived, used immediately, written once and read once [4, 107]. On the other hand, there are new characteristics – the blocks are distributed, large in number, large in aggregate size, and a compu-

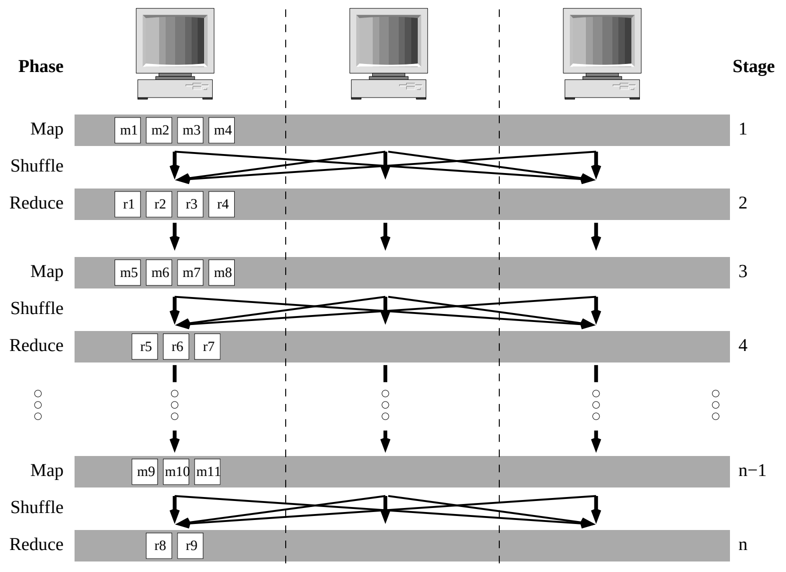

Figure 3.1: An example of Pig executing a linear sequence of MapReduce stages. The Shuffle phase involves all-to-all data transfers, while local data is used between each Reduce and the next Map.

tation stage cannot start until all its input intermediate data has been generated by the previous stage. This large-scale, distributed, short-lived, computational-barrier nature of intermediate data firstly creates network bottlenecks because it has to be transferred in-between stages [23]. Worse still, it prolongs job completion times under failures (as we show in Section 3.4.2).

Despite these issues, we observe that the intermediate data management problem is largely unexplored in current dataflow programming frameworks. The most popular approach to intermediate data management is to rely on the local filesystem as in Hadoop and Pig. Data is written locally on the node generating it, and read remotely by the next node that needs it. Failures are handled by the frameworks themselves without much assistance from the storage systems they use. Thus, when there is a failure, affected tasks are typically re-executed to generate intermediate data again. In a sense, this design decision is based on the assumption that intermediate data is temporary, and regeneration of it is cheap and easy.

Although this assumption and the design decision may be somewhat reasonable for MapReduce with only two stages, it becomes unreasonable for more general multi-stage dataflow frameworks, as we detail in Section 3.4.2. In a nutshell, the problem is that a failure can lead to expensive *cascaded re-execution*; some tasks in *every stage from the beginning* have to be re-executed sequentially up to the stage where the failure happened. This problem shows that efficient and reliable handling of intermediate data can play a key role in optimizing the execution of dataflow programs.

Reported experiences with dataflow frameworks in large-scale environments indicate that transient and permanent failures are prevalent, and will only exacerbate as more organizations process larger data with multiple stages. For example, Google reports 5 average worker deaths per MapReduce job in March 2006 [22], and at least one disk failure in every run of a 6-hour MapReduce job with 4,000 machines [68]. Yahoo! reports their Web graph generation (called *WebMap*) has grown to a chain of 100 MapReduce jobs [78]. In addition, many organizations such as Facebook and Last.fm report their usage of MapReduce, Pig, and Hive processing hundreds of TBs of data already with a few TBs of daily increase [108].

Thus, it is our position that we must design a new storage system that treats intermediate data as a first-class citizen. We believe that a storage system (as opposed to the dataflow frameworks) is the natural and right abstraction to efficiently and reliably handle intermediate data, regardless of the failure types. In the following sections, we discuss the taxonomy of the intermediate data management solution space, characteristics of this type of data, the requirements for a solution, the applicability of candidate solutions, and finally our design of ISS, the intermediate storage system.

3.2 Taxonomy

Figure 3.2 shows the taxonomy of the intermediate data management solution space. ISS provides a solution for a large-scale dynamic environment, where there are failures and bandwidth dynamism (the upper right region). Small-scale applications such as compilers typically run on a single machine, i.e., a static environment where failures are rare in the lifetime of an application run. Thus, the local filesystem is usually enough to handle intermediate data (the bottom left region). Some applications need to handle a large volume of intermediate data, but run in a static environment. For example, many 2-stage MapReduce programs might generate PBs of intermediate data, but rarely encounter failures since their execution times only last a few minutes. In this case, one can use the current MapReduce framework without any replication of intermediate data (the upper left region). Finally, some MapReduce programs run in a dynamic environment with failures and unpredictable bandwidth availability such as virtualized clouds (e.g., Amazon EC2). However, if they only generate a low volume of intermediate data, a simple replication mechanism with no interference minimization techniques (discussed in Section 3.6 and 3.7) suffices to provide availability of intermediate data, since a low volume does not cause much interference (the bottom right region).

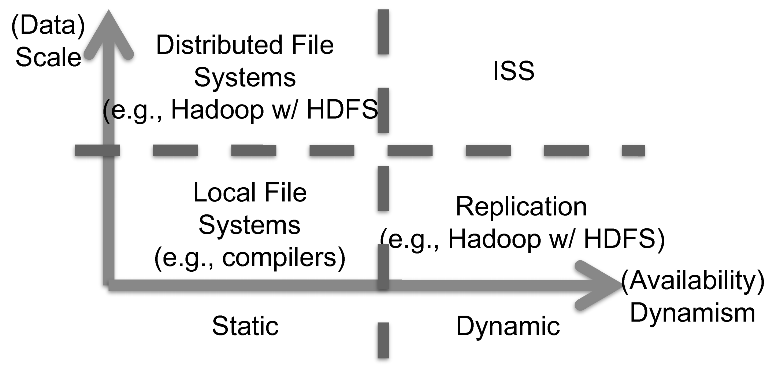

Figure 3.2: Taxonomy of the Intermediate Data Management Problem

3.3 Background: MapReduce

Since much of our discussion in this chapter involves MapReduce, we briefly summarize how MapReduce works. The goal of our discussion here is not to have a comprehensive introduction to MapReduce, but rather to provide a primer.

Overview of MapReduce The MapReduce framework is a runtime system that utilizes a cluster of machines, often dedicated to the framework. The framework takes two input functions, *Map()* and *Reduce()*, written by a programmer, and executes each function in parallel over a set of distributed data files. A distributed file system such as GFS [31] and HDFS [39] is used to store input and output data. There are currently two implementations of MapReduce — the original MapReduce from Google that is not released to the public, and an open-source implementation called Hadoop from Yahoo!.

Three Phases of MapReduce There are three phases that every MapReduce program execution can be divided into. They are Map, Shuffle, and Reduce. Each phase utilizes every machine dedicated to the MapReduce framework. We summarize each phase below.

1. Map: The Map phase executes the user-provided Map function in parallel over the MapReduce cluster. The input data is divided into chunks and stored in a distributed file system, e.g., GFS or HDFS. Each Map task reads some number of chunks from the distributed file system and generates intermediate data. This intermediate data is used as the input to the Reduce phase. In order to execute the Reduce function in parallel, the intermediate data is again partitioned and organized into a number of chunks. These chunks are stored locally on the nodes that generate them.
2. Shuffle: The Shuffle phase moves the intermediate data generated by the Map phase among the machines in the MapReduce cluster. The communi-

cation pattern is all-to-all as shown in Figure 3.1. Because of this all-to-all nature of communication, this phase heavily utilizes the network, and is often considered as the bottleneck [23].

3. Reduce: The Reduce phase executes the user-provided Reduce function in parallel over the MapReduce cluster. It stores its output in the distributed file system. This output can be the final output if there is only one MapReduce job. However, it can also be the intermediate output for the next MapReduce job, if the user wants to run a chain of MapReduce jobs as in Yahoo!'s *WebMap* [78].

The most important aspect of these three phases in regards to intermediate data is that the intermediate data is stored locally in the Map phase, transferred remotely in the Shuffle phase, and read in the Reduce phase.

3.4 Why Study Intermediate Data?

In this section, we discuss some salient characteristics of intermediate data, and outline the requirements for an intermediate data management system.

3.4.1 Characteristics of Intermediate Data

Persistent data stored in distributed file systems ranges in size from small to large, is likely read multiple times, and is typically long-lived. In comparison, intermediate data generated in cloud programming paradigms has uniquely contrasting characteristics. Through our study of MapReduce, Dryad, Pig, etc., we have gleaned three main characteristics that are common to intermediate data in all these systems. We discuss them below.
Size and Distribution of Data: Unlike traditional file system data, the intermediate data generated by cloud computing paradigms potentially has: (1) a large number of blocks, (2) variable block sizes (across tasks, even within the same job), (3) a large aggregate size between consecutive stages, and (4) distribution across a large number of nodes.
Write Once-Read Once: Intermediate data typically follows a write once-read once pattern. Each block of intermediate data is generated by one task only, and read by one task only. For instance, in MapReduce, each block of intermediate data is produced by one Map task, belongs to a region, and is transmitted to the unique Reduce task assigned to the region.
Short-Lived and Used-Immediately: Intermediate data is short-lived because once a block is written by a task, it is transferred to (and used immediately by) the next task. For instance, in Hadoop, a data block generated by a Map task is transferred during the Shuffle phase to the block's corresponding Reduce task.

Topology	1 core switch connecting 4 LANs (5 nodes each)
Bandwidth	100 Mbps
# of Nodes	20
Input Data	2GB/Node
Workload	Sort

Table 3.1: Emulab Experimental Setup Referred to as ENV1

Topology	1 core switch connecting 4 LANs (20 nodes each)
Bandwidth	100 Mbps Top-of-the-Rack Switch 1 Gbps Core Switch
# of Nodes	80
Input Data	2GB/node
Workload	Sort

Table 3.2: Emulab Experimental Setup Referred to as ENV2

The above three characteristics morph into major challenges at runtime when one considers the effect of failures. For instance, when tasks are re-executed due to a failure, intermediate data may be read multiple times or generated multiple times, prolonging the lifetime of the intermediate data. In summary, failures lead to additional overhead for generating, writing, reading, and storing intermediate data, eventually increasing job completion time.

3.4.2 Effect of Failures

We discuss the effect of failures on dataflow computations. Suppose we run the dataflow computation in Figure 3.1 using Pig. Also, suppose that a failure occurs on a node running task t at stage n (e.g., due to a disk failure, a machine failure, etc.). Note that, since Pig (as well as other dataflow programming frameworks) relies on the local filesystem to store intermediate data, this failure results in the loss of all the intermediate data from stage 1 to $(n-1)$ stored locally on the failed node. When a failure occurs, Pig will reschedule the failed task t to a different node available for re-execution. However, the re-execution of t cannot proceed right away, because some portion of its input is lost by the failed node. More precisely, the input of task t is generated by all the tasks in stage $(n-1)$ including the tasks run on the failed node. Thus, those tasks run on the failed node have to be re-executed to regenerate the lost portion of the input for task t. In turn, this requires re-execution of tasks run on the failed node in stage $(n-2)$, and this cascades all the way back to stage 1. Thus, some tasks in every stage from the beginning will have to be re-executed sequentially up to the current stage. We call this *cascaded re-execution.* Although we present this problem using Pig as a case study, any dataflow framework with multiple stages will suffer from this problem as well.

Figure 3.3 shows the effect of a single failure on the runtime of a Hadoop

Topology	1 core switch connecting 4 LANs (20 nodes each)
Bandwidth	1 Gbps
# of Nodes	80
Input Data	2GB/node
Workload	Sort

Table 3.3: Emulab Experimental Setup Referred to as ENV3

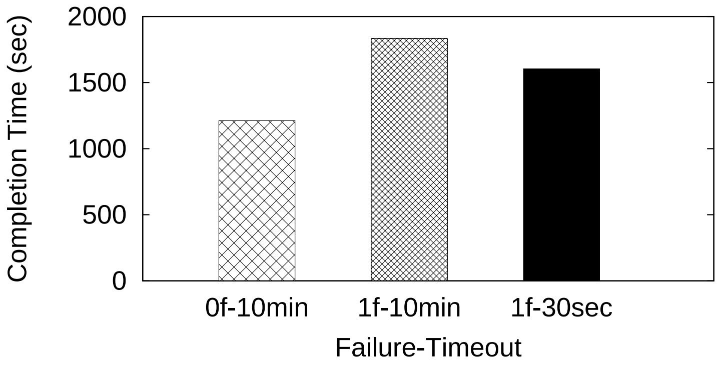

Figure 3.3: Effect of a Failure on a Hadoop Job (ENV1)

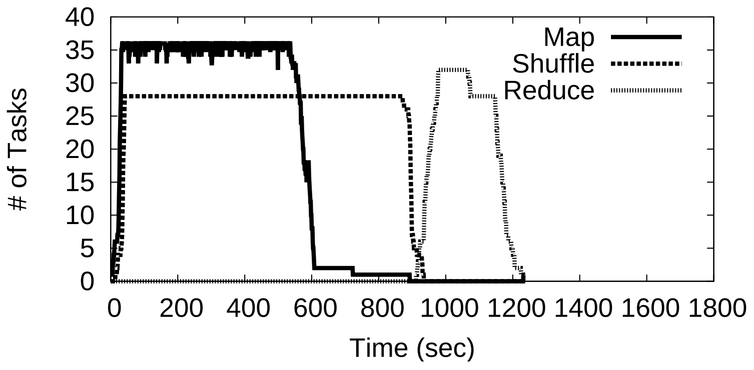

Figure 3.4: Behavior of a Hadoop Job (ENV1)

job (i.e., a two-stage job). The failure is injected at a random node immediately after the last Map task completes. The leftmost bar is the runtime without failures. The middle bar shows the runtime with 1 failure, when Hadoop's node failure detection timeout is 10 minutes (the default) – *a single failure causes a 50% increase in completion time.* Further reducing the timeout to 30 seconds does not help much – the runtime degradation is still high (33%).

To understand this further, Figures 3.4 and 3.5 show the number of tasks over time for two bars of Figure 3.3 (0f-10min and 1f-30sec). Figure 3.4 shows clearly the barrier – Reduce tasks do not start until the Shuffles are (almost) done around t=925 sec. We made several observations from the experiment of Figure 3.5: (1) a single node failure caused several Map tasks to be re-executed

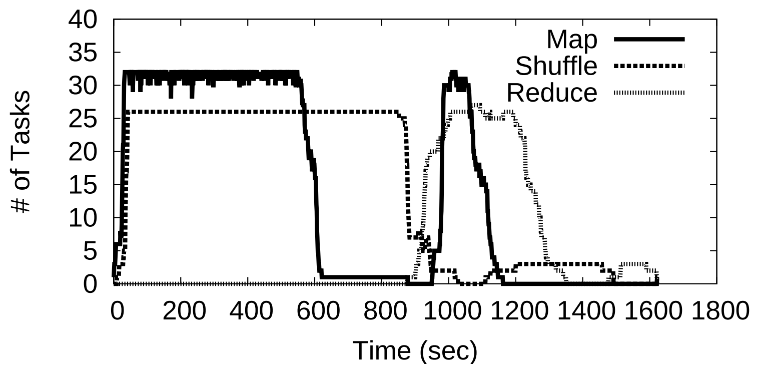

Figure 3.5: Behavior of a Hadoop Job under 1 Failure (ENV1)

(starting t=925 sec), (2) a renewed Shuffle phase starts after these re-executed Maps finish (starting t=1100 sec), and (3) Reduces that were running on the failed node and that were not able to Shuffle data from the failed node, get re-executed as well towards the end (t=1500 sec). While this experiment shows cascaded re-execution within a single stage, we believe it shows that in multi-stage dataflow computations, a few node failures will cause far worse degradation in job completion times.

3.4.3 Requirements

Based on the discussion so far, we believe that the problem of managing intermediate data generated during dataflow computations, deserves deeper study as a first-class problem. Motivated by the observation that the main challenge is dealing with failure, we arrive at the following two major requirements that any effective intermediate storage system needs to satisfy: *availability* of intermediate data, and *minimal interference* on foreground network traffic generated by the dataflow computation. We elaborate below.

Data Availability: A task in a dataflow stage cannot be executed if the intermediate input data is unavailable. A system that provides higher availability for intermediate data will suffer from fewer delays for re-executing tasks in case of failure. In multi-stage computations, high availability is critical as it minimizes the effect of cascaded re-execution (Section 3.4.2).

Minimal Interference: At the same time, data availability cannot be pursued over-aggressively. In particular, since intermediate data is used immediately, there is high network contention for foreground traffic of the intermediate data transferred to the next stage (e.g., by Shuffle in MapReduce) [23]. An intermediate data management system needs to minimize interference on such foreground traffic, in order to keep the job completion time low, especially in the common case of no failures.

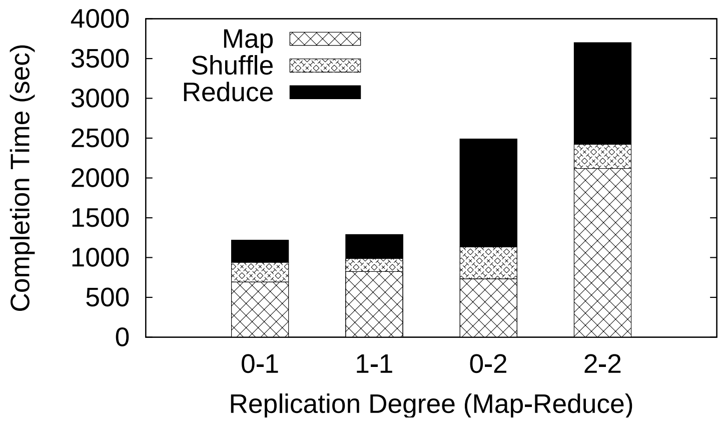

Figure 3.6: Using HDFS (ENV1): Varying Replication Degree (i-j) for Output of Map (i) and Reduce (j)

3.5 Exploring the Design Space

Current dataflow frameworks store intermediate data locally at the outputting node and have it read remotely. They use purely *reactive* strategies to cope with node failures or other causes of data loss. Thus, in MapReduce, there is no mechanism to ensure intermediate data availability. The loss of Map output data results in the re-execution of those Map tasks, with the further risk of cascaded re-execution (Section 3.4.2).

In contrast, we recognize that can be satisfied with any distributed file system that replicates data. The unanswered question is: how much interference the replication process will cause to the foreground job completion time. Thus, a natural approach to satisfying both requirements is to start with an existing distributed file system, determine how much interference it causes, and reason about how one can reduce the interference.

3.5.1 Replication Overhead of HDFS

As the first step, we experimentally explore the possibility of using a distributed file system especially designed for data-intensive environments. We choose HDFS, which is used by Hadoop to store the input to the Map phase and the output from the Reduce phase. We modify Hadoop so that HDFS can store the intermediate output from the Map phase.

Figure 3.6 shows four bars, each annotated i-j, where i is the replication degree within HDFS for Map output (i=0 being the default local write-remote read) and j the replication degree for Reduce output. When one incorporates HDFS to store Map data into HDFS, there is only a small increase in completion time (see 1-1 vs. 0-1). This is because the only additional overheads are HDFS metadata that point to the local copy of the output already stored at the Map node.

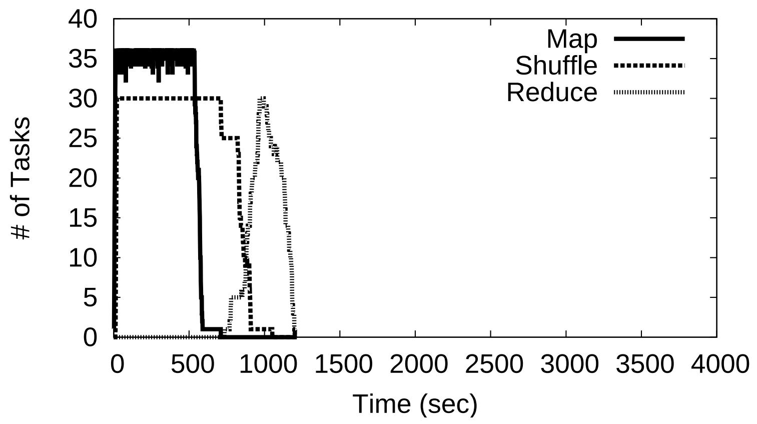

Figure 3.7: Map Replication: 0, Reduce Replication: 1 (ENV1)

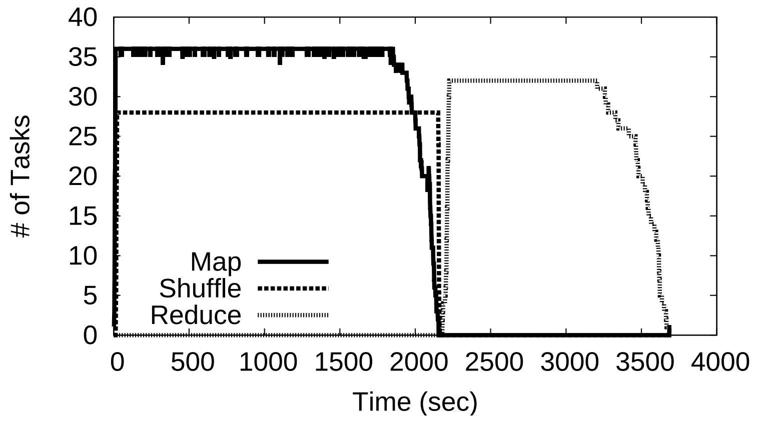

Figure 3.8: Map Replication: 2, Reduce Replication: 2 (ENV1)

Increasing the Reduce replication degree to 2, on the other hand (see 0-2 vs. 0-1) doubles the job completion time[1]. Further, replicating Map output increases the completion time by a factor of about 3 compared to the default (see 2-2 vs. 0-1). To delve into this further, we compare the timeline of tasks run by Hadoop without replication in Figure 3.7 and with replication in Figure 3.8. We observe that the Map runtime increases by a factor of over 3, Shuffle runtime by a factor of 2, and Reduce runtime by a factor of around 2.

Hence, we conclude that using HDFS as-is will not work due to interference.

3.5.2 Background Replication with TCP-Nice

One way to reduce the interference is to use a background transport protocol beneath HDFS (such as TCP-Nice [105] and TCP-LP [59]), so that we could replicate intermediate data without affecting foreground traffic. Thus, we discuss this possibility qualitatively here. We focus our discussion around TCP-

[1]This plot seems to indicate why the Pig system (built atop Hadoop) uses a default replication degree of 1 for Reduce.

Nice, a well-known background transport protocol. However, we believe our discussion below is generally applicable to any background transport protocol.

TCP-Nice allows a flow to run in the "background" with little or no interference to normal flows. These background flows only utilize "spare" bandwidth unused by normal flows. This spare bandwidth exists because there is local computation and disk I/O performed in both Map and Reduce phases. Thus, we could put replication flows in the background using TCP-Nice, so that they would not interfere with the foreground traffic such as Shuffle.

The biggest drawback with this approach is that TCP-Nice (as well as any background transport protocol) is designed as a general transport protocol. This means that it does not assume (and is thus unable to utilize) the knowledge of applications using it and the environments in which it is operating. For example, TCP-Nice does not know which flows are foreground and which are not. Thus, TCP-Nice gives background flows lower priority than any other flow in the network. This means that a background replication flow will get a priority lower than Shuffle flows, as well as other flows unrelated to the dataflow application, e.g., any ftp or http traffic going through the same shared core switch of a data center.

Moreover, TCP-Nice minimizes interference at the expense of network utilization. [2] This is because a background flow reacts to congestion by aggressively reducing its transfer rate. Thus, applications cannot predict the behavior of TCP-Nice in terms of bandwidth utilization and transfer duration. This is not desirable for intermediate data replication, where timely replication is important.

Finally, TCP-Nice relies on a rate-control mechanism to avoid interference. However, replication is a process that is more than just data transfer. There are other opportunities (e.g., replica placement, data selection, etc.) that one can explore in addition to rate-control, as we demonstrate from the next section.

3.6 Three Hypotheses for Eliminating Interference

Our next step is to determine what factors contribute to the replication overhead and reason about how to reduce the overhead. To this end, we have comprehensively examined how HDFS replicates data, which environment dataflow programming frameworks target, and how to exploit application-specific knowledge in replication. After this examination, we have arrived at three hypotheses. The first hypothesis is HDFS-specific. However, the other two hypotheses are generally applicable to any system design. The hypotheses and the reasoning behind them follow below.

[2] In fact, the original paper on TCP-Nice [105] makes clear that network utilization is not a design goal.

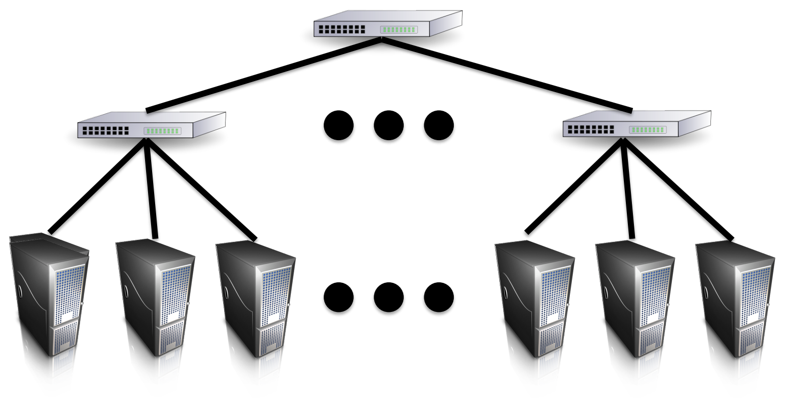

Figure 3.9: A 2-Level Topology Example

1. **Asynchronous replication can help:** We have observed that HDFS replication works synchronously and reports data as stored only when replication is complete. This leads to Map tasks blocking for the HDFS replication to complete. Thus, if we replicate intermediate data asynchronously, Map tasks will proceed without waiting for the replication to complete.

2. **The replication process can exploit the inherent bandwidth heterogeneity of data centers:** Typically, the dataflow programming frameworks target data centers, where the network topology is hierarchical, e.g., 2-level with top-of-the rack switches and a core switch [19]. Figure 3.9 shows an example. We mainly consider this 2-level architecture because it supports the scale of up to 8K nodes, which is sufficient for the current scale of dataflow programming frameworks [78, 108]. However, our discussion below holds for any number of levels and arbitrary network topologies.

   In a hierarchical topology, the bottleneck is the core switch because it is shared by many racks and machines. Thus, there is inherent heterogeneity in bandwidth — inter-rack bandwidth is scarce compared to intra-rack bandwidth. This is especially true in the Shuffle phase of MapReduce. Since the communication pattern among the nodes is all-to-all in the Shuffle phase, the core switch is heavily utilized during this phase while top-of-the-rack switches are under-utilized. The replication process can exploit this behavior by replicating only within each rack and avoiding the data transfer through the core switch.

3. **Data selection can help:** As we discuss in Section 3.4.2, the problem of cascaded re-execution is caused by the loss of intermediate data. However, if we examine this phenomenon more closely, the exact cause is the loss of intermediate data that is *consumed locally*.

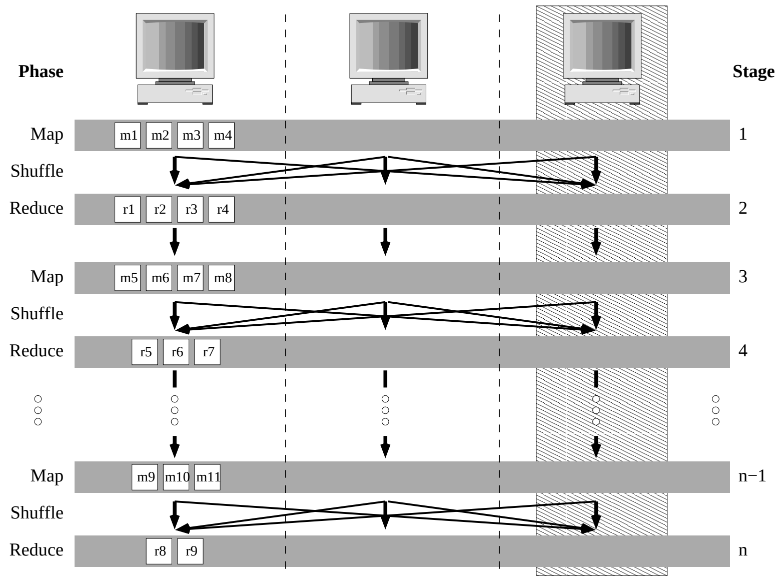

Figure 3.10: A Failure Example

Figure 3.10 illustrates this point. If there is a machine failure at the Reduce phase (e.g., stage 4), affected Reduce tasks can be restarted if they can still fetch the intermediate data generated by the Map tasks that ran on the same machine (indicated by straight-down arrows). Other necessary pieces of intermediate data can be fetched from all the other machines.

Thus, replication can benefit from this observation by only replicating locally-consumed data. Especially in Map stage, the amount of intermediate data that needs to be replicated can potentially be reduced significantly this way. However, since Reduce outputs are always locally consumed, this technique will be of little help in reducing the replication interference of Reduce data.

Qualitatively, these three hypotheses appear to help reducing the interference. However, the question is not *if* these techniques will help, but *how much*. Thus, we experimentally explore how much each technique can help in reducing the interference next.

3.7 Hypotheses Validation

In this section, we validate the hypotheses in Section 3.6 experimentally and show how much reduction of interference we can achieve with them.

3.7.1 Asynchronous Replication

We have noticed that HDFS replication works synchronously and reports data as stored only when replication is complete. This leads to Map tasks blocking

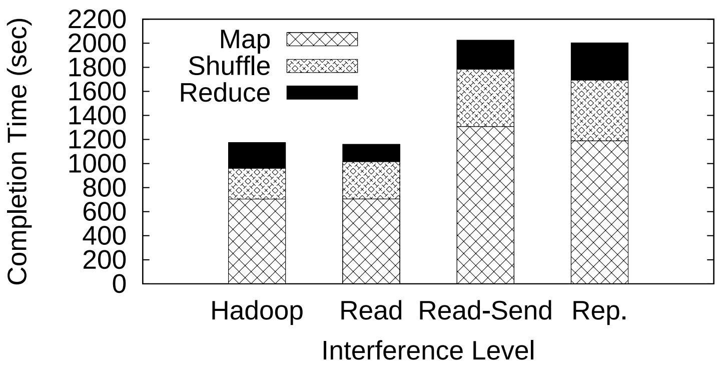

Figure 3.11: Asynchronous Replication with HDFS (ENV1)

for the HDFS replication to complete. Hence, we modify HDFS and experiment further with asynchronous replication.

Figure 3.11 shows average completion times of MapReduce in four experimental settings. The purpose of the experiment is (1) to examine the performance of MapReduce when the intermediate output is asynchronously-replicated using HDFS, and (2) to understand where the sources of interference are. Thus, the four bars are presented in the order of increasing degree of interference.

In the left-most experiment (labeled as *Hadoop*), we use the original Hadoop that does not replicate the intermediate data, hence there is no interference due to replication. In the right-most experiment (labeled as *Rep.*), we use HDFS to asynchronously replicate the intermediate data to a remote node. Although the degree of interference is less than synchronous replication, performance still degrades to the point where job completion time takes considerably longer.

The middle two experiments help to show the source of the performance hit by breaking down HDFS replication into its individual operations. In the second experiment (labeled as *Read*), we take only the first step of replication, which is to read the Map output. This incurs a local disk read. In the next experiment (labeled as *Read-Send*), we use HDFS to asynchronously replicate the intermediate data *without* physically writing to the disks. This involves a local disk read and a network transfer, but no disk writes.

When we only read the intermediate data, there is hardly any difference in the overall completion time (*Hadoop* vs. *Read*). However, when the replication process starts using the network (*Read* vs. *Read-Send*), there is a significant overhead that results in doubling the completion time. This is primarily due to the increase in the Shuffle phase. [3] The increase in the Map finish time in *Read-Send* is also due to the network interference, since some Maps need to fetch their inputs from remote nodes. Finally, we notice that the interference of disk writes is very low (*Read-Send* vs. *Rep.*).

[3]The plot shows the finish time of each phase, but does not show the initial start time for Shuffle and Reduce; the phases in fact overlap as seen in Figure 3.4 and 3.5.

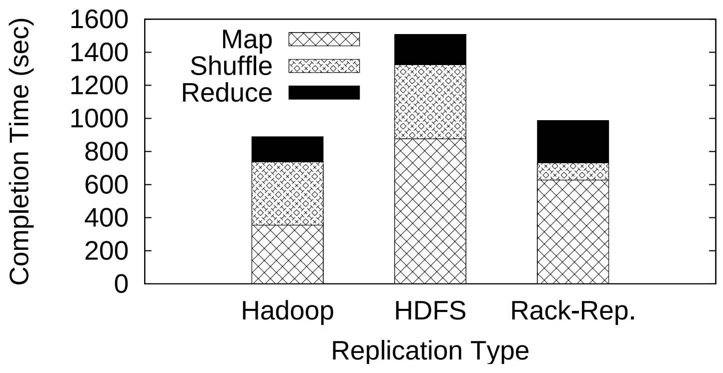

Figure 3.12: Rack-Level Replication (ENV2)

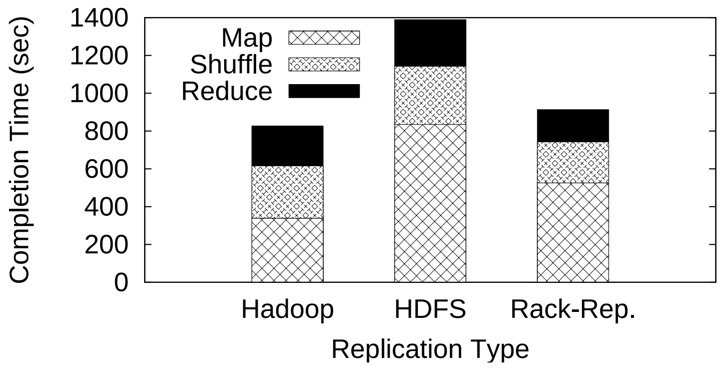

Figure 3.13: Rack-Level Replication (ENV3)

3.7.2 Rack-Level Replication

By default, HDFS places the first copy on the local machine, and the next copy on a machine located in a remote rack. However, as we have reasoned in Section 3.6 there is potential bandwidth availability within a rack. Thus, we need to quantify how much reduction of interference one can achieve by replicating within a rack.

Figures 3.12 and 3.13 show the results for two different settings. We have actually performed our experiments in a few more settings in order to quantify the benefit of rack-level replication in various scenarios, but the results were similar.[4] In all experiments, we use 80 nodes (4 LANs and 20 nodes/LAN), but the bandwidth settings are different as specified. The left-most bar (labeled as *Hadoop*) shows the result with the original Hadoop. It does not perform any replication of the intermediate data (the Map outputs). The middle bar (labeled as *HDFS*) uses HDFS to replicate the intermediate data. The right-most bar (labeled as *Rack-Rep.* replicates the intermediate data within the same rack.

All three plots show that we can achieve significant reduction in interference

[4]More specifically, we have used 20 machines spread over 4 LANs, 60 machines in one LAN, and 80 machines in one LAN, with various bandwidth combinations using 100Mbps and 1Gbps switches.

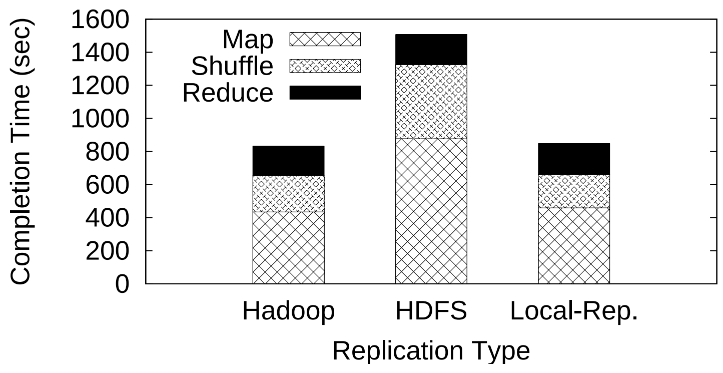

Figure 3.14: Locally-Consumed Data Replication (ENV2)

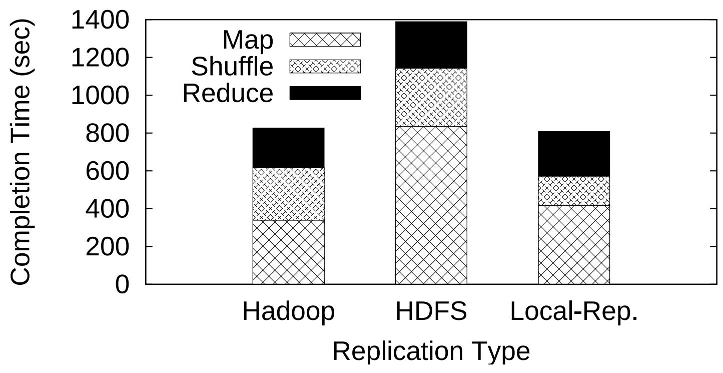

Figure 3.15: Locally-Consumed Data Replication (ENV3)

just by replicating within the same rack. The actual completion time varies depending on the configuration, but the increase only ranges from 70 seconds to 100 seconds. In contrast, when using HDFS for replication, completion times were increased nearly twice as long.

As we discuss in Section 3.6 and 3.7.1, this reduction is possible because 1) the network is the main source of interference, and 2) often times there is idle bandwidth within each rack.

3.7.3 Locally-Consumed Data

The third possibility for reducing the interference was to only replicate locally-consumed data. As we discuss in Section 3.6, this is possible because the Shuffle phase transfers intermediate data in an all-to-all manner, which leads to natural replication.

Figures 3.14 and 3.15 show the results. Both plots show that there is very little overhead when we replicate the locally-consumed data. This is due to the amount of data that needs to be replicated. If we replicate only the locally-consumed data, the amount of replicated data is reduced by $\frac{1}{N}$ assuming uniform partitioning, where N is the number of total machines.

Name	Description
int iss_create(String pathName)	Creates a new intermediate file
int iss_open(String pathName, int off, int len)	Opens an existing intermediate file
int iss_write(int fd, char[] buf, int off, int len)	Writes the content of a buffer to an intermediate file at the given offset, and replicates it in the same rack
int iss_read(int fd, char[] buf, int off, int len)	Reads the content of an intermediate file to a buffer from the given offset
int iss_close(int fd)	Closes an intermediate file

Table 3.4: API Extension for ISS (Written in POSIX-Style)

However, this technique is only effective for reducing the Map outputs. This is because Reduce outputs are always consumed locally, and every piece of data has to be replicated.

3.8 ISS: Intermediate Storage System

The results in Section 3.7 showed that a rack-level replication mechanism that asynchronously replicates locally-consumed data allows a storage system to satisfy the requirements of intermediate data availability and minimal interference. However, the question of how to implement this mechanism still remains. Thus, we address this question by discussing the design of our system, ISS (Intermediate Storage System).

3.8.1 Interface and Design Choices

ISS is implemented as an extension to HDFS. Concretely, we add an implementation of a rack-level replication mechanism that asynchronously replicates locally-consumed data (i.e., a mechanism with all three hypotheses from Section 3.6). ISS takes care of all aspects of managing intermediate data, including writing, reading, replicating, and shuffling. Thus, a dataflow programming framework that uses ISS does not need to perform the Shuffle phase manually, since ISS seamlessly transfers the intermediate data from writers (e.g., Map tasks) to readers (e.g., Reduce tasks). ISS extends the API of HDFS as we summarize in Table 3.4.

There are some important design choices in ISS as we discuss shortly. They are tailored toward programming convenience for dataflow programming frameworks. We elaborate our design choices below.

1. All files are immutable. Once a file is created and closed, it can only be opened for read operations. Thus, the protocol for a writer is create-write-close, and the protocol for a reader is open-read-close.

2. The file becomes visible immediately after it is created.

3. There can be only one writer at a time, but multiple readers are allowed over a duration.

4. When a reader opens a file, it blocks until the file is closed by the writer of that file (if it is opened and being written by the writer at the same time) and the file chunk indicated by the offset and length fields is completely copied to the local disk.

3.8.2 MapReduce Example

To illustrate how a dataflow programming framework can utilize ISS, consider MapReduce as an example. In MapReduce, each Map task can create intermediate files using iss_create(). The Hadoop's master machine, which is in charge of scheduling and coordinating MapReduce tasks, can keep track of all the file names that Map tasks create. Since a similar process is necessary for the Shuffle phase in the current MapReduce, this is not a significant departure from the current implementation. However, since the Shuffle phase is automatically performed by ISS, we can reduce the overhead and complexity of the Shuffle process from the master. Each Map task then proceeds with iss_write() to write the intermediate files. In the meantime, each Reduce Task can learn all the file names through the master, then uses iss_open() to wait until the files are completely copied to the local disk. Note that this step replaces the Shuffle phase. After fetching the files, each Reduce task can proceed with iss_read() as it would with a local file.

3.9 Evaluation

We evaluate performance characteristics such as behavior under failures and replication completion time. In Section 3.7, we have already presented how our asynchronous rack-level selective replication mechanism performs.

3.9.1 Performance Under Failure

As we have discussed in Section 3.4.2, the original Hadoop re-executes Map tasks again when it encounters failures in the Reduce phase. This is illustrated in Figure 3.16 (although we have shown the result of a failure injection experiment in Section 3.4.2, we show a similar result in a different setting in this section for the sake of discussion).

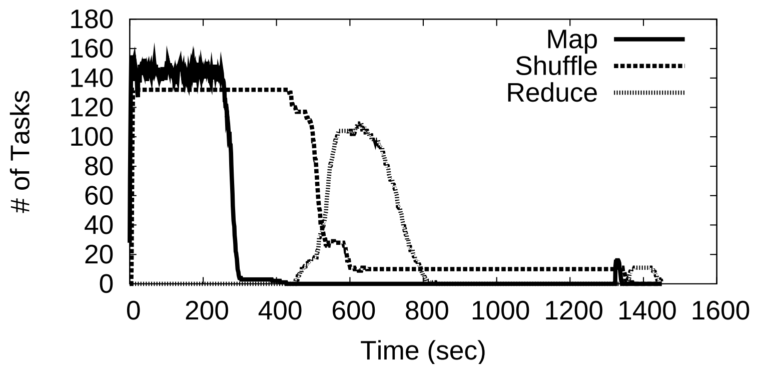

Figure 3.16: Hadoop with One Machine Failure Injection (ENV3)

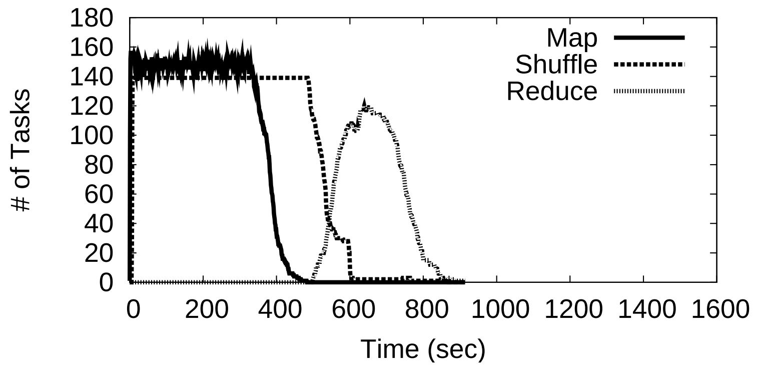

Figure 3.17: Expected Behavior of Hadoop Augmented with ISS (ENV3)

In Figure 3.16, a machine failure is injected at around time $t = 500$ second in the beginning of the Reduce phase. This failure goes undetected until the failed machine's heartbeat timeout expires (the default timeout value is 10 minutes). Thus, at around time $t = 1100$ second, we can see that Map tasks are re-assigned to a different machine and re-executed, shown by the re-surge of the bold line.

This behavior would have been different if the intermediate data generated by the failed node had been available at some other node. In fact, the effect of failure would have been almost non-existent because of "speculative execution" implemented in Hadoop. In a nutshell, speculative execution detects any task that is making slow progress compared to other tasks, and redundantly executes the detected slow task on a faster machine. Thus, if the intermediate data is available, speculative execution can identify tasks that were running on the failed machine as slow tasks and redundantly execute them on different machines. However, the speculative execution does not help if the intermediate data is not available as in Figure 3.16, since even the speculated tasks need the intermediate data.

Figure 3.17 demonstrates this behavior. We emulate what would happen with ISS, speculative execution, and a machine failure in the following ways.

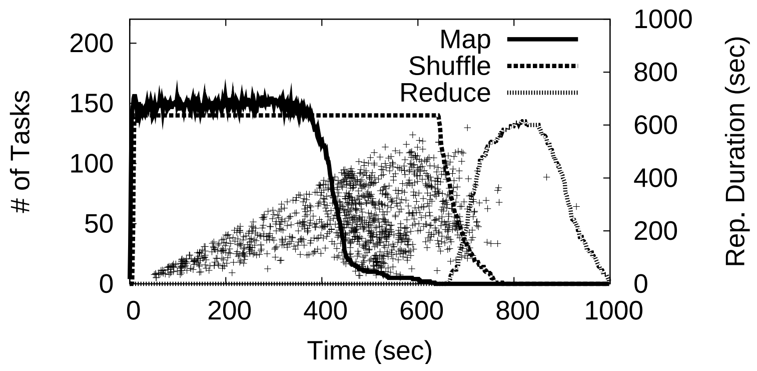

Figure 3.18: Replication Completion Time (ENV2)

First, we kill tasks that are running on one machine in order to emulate a machine failure at time $t = 800$ second. This is different from the failure injection experiment performed for Figure 3.16, since unlike actual machine failures, Hadoop detects task failures immediately and reacts to the failures. Second, we still let the intermediate data accessible from other machines. Third, since Hadoop immediately re-executes tasks upon their failures, we kill tasks at the time ($t = 800$ second) when speculative execution might detect and redundantly execute those tasks.

We observe that the completion time increases from 914 seconds in Figure 3.17 to 1450 seconds in Figure 3.16, showing approximately 59% of slowdown (or 37% speedup). Compared to the average completion time of Hadoop without any failure shown in both Figure 3.13 and Figure 3.15, the completion time of Hadoop with ISS under one failure increases only approximately 10% in Figure 3.17. However, the completion time of Hadoop without ISS under one failure in Figure 3.16 increases approximately 75%, which is quite significant.

3.9.2 Replication Completion Time

Replication completion time is an important metric since it shows the "window of vulnerability", i.e., the period of time during which ISS cannot provide data availability for an intermediate data block. This is shown in Figures 3.18 and 3.19. In both plots, we plot the time taken by each block (size: 128MB) to be completely replicated (shown by crosses) along with the timeline of a foreground MapReduce job (shown by lines). We have chosen to show the performance of asynchronous rack-level replication mechanism that replicates all the intermediate data generated by the Map tasks, in order to picture the complete process of replication. We show one specific run for each of two settings, ENV2 and ENV3. We omit the discussion about completion time comparison, since we have discussed it already in Section 3.7.

We observe a general trend that the replication time takes longer for each block towards the end of the Shuffle phase. This is due to lower bandwidth

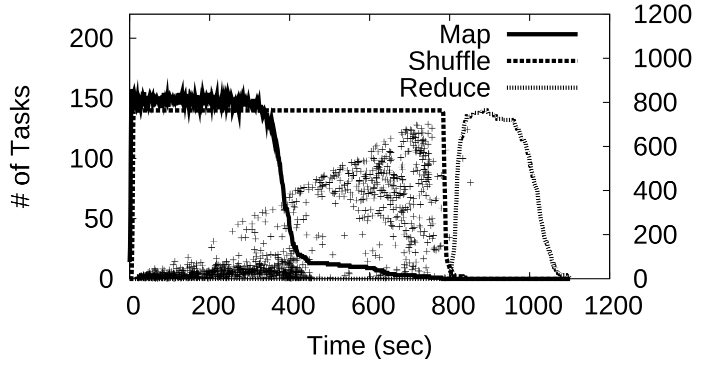

Figure 3.19: Replication Completion Time (ENV3)

availability, since the network is heavily utilized during the Shuffle phase. In Figure 3.18, the last replicated block finishes at around time $t = 950$ second, shortly before the Reduce phase ends. In contrast, in Figure 3.19, the last replicated block finishes at around time $t = 850$ second, shortly after the Shuffle phase ends. This difference is due to the bandwidth available within a rack. ENV2 uses 100 Mbps switches, while ENV3 uses 1Gbps switches. However, we can see that even in a bandwidth-scarce environment such as ENV2, the replication finishes before the Reduce phase.

3.10 Summary

We have shown the need, presented requirements, and a design of an intermediate storage system (ISS) that treats intermediate storage as a first-class citizen for dataflow programs. We have shown experimentally that the existing approaches are insufficient satisfying the requirements of data availability and minimal interference. We have also shown that our asynchronous rack-level selective replication mechanism is effective, and almost eliminates all the interference. Our failure injection experiments show that Hadoop without ISS can slowdown the performance by approximately 59% compared to Hadoop with ISS. Replication completion time shows that asynchronous rack-level replication can be done quickly even in a bandwidth-scarce environment.

Chapter 4

On-Demand Worker-Centric Scheduling

This chapter presents the motivation, background, design and evaluation of worker-centric scheduling strategies for data-intensive applications in Grids. Worker-centric scheduling strategies are necessary due to the scale of data ranging from terabytes to petabytes that current data-intensive applications need to handle, as well as the dynamic nature of resource availability in Grids. We detail these challenges and how worker-centric scheduling strategies address them in this chapter.

4.1 Motivation

Data-intensive Grid applications are the applications that run on distributed Grid sites and are characterized by their access of large amounts of data sets. In attempting to minimize the execution time for such applications, schedulers of the Grid application are hampered by the sheer size of the data sets involved. While these data sets are mostly read-only and predefined, their size ranges from several terabytes to petabytes [1]. Examples of such data-intensive Grid applications can be found in many scientific domains such as Physics, Earth science, and Astronomy, e.g., [72, 96].

At run time, this large scale of the data sets makes it impractical to replicate all the data at every execution site, where the term "site" refers to a cluster of client machines ("workers"). Instead, the typical approach to structuring such a data-intensive Grid application (i.e., the "job") is to partition the execution code into several small "tasks", and to divide up the data into several disjoint pieces, each of which we call a "file". Thus, each task requires a specific subset of the files that constitute the job data, and a site begins the execution of a given task by retrieving all those required files.

When running a data-intensive Grid application across a collection of several sites, one of the most challenging problems is the design of a (global) Grid scheduling algorithm. Specifically, since the cost of data transfer is a major bottleneck for the execution time [72, 94, 86, 12], the main goal of the (global) scheduling algorithm becomes assigning tasks to sites in such a way as to reduce the frequency and amount of data transfer [94, 86, 12]. Fortunately, many data-intensive Grid applications exhibit *locality of interest, i.e.,* a file is often accessed by multiple tasks and also, a set of files that are accessed by one task are also likely to be accessed together by other tasks [45] (note: we will also use *data-sharing* whenever appropriate).

Our analysis of *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [72, 96]) (explained in detail in Section 4.3.1) also shows the locality of interest in data-intensive Grid applications. There is a significant number of files accessed by multiple tasks (Figure 4.2) and there is a large number of tasks that access the same set of files during their execution (Figure 4.3). This locality of interest gives an opportunity to reduce the numbers of both redundant file transfers and file replicas, and is present in wide variety of applications including data mining, image processing, genomics [94], and spatial processing applications which consist of tasks that process overlapping regions [72].

Previously, locality of interest has been exploited for scheduling and workflow planning in Grid data-intensive applications. Casanova *et al.* [12], Ranganathan *et al.* [86] and Santos-Neto *et al.* [94] successfully demonstrated the benefits of their locality-aware schedulers over traditional schedulers. However, the scheduler design in all the mentioned papers is *task-centric*, i.e., the global scheduler assigns a task to a worker, without considering whether or not the worker can start executing the task immediately after the task assignment.

We observe that such task-centric scheduling suffers from two major issues when dealing with data-intensive applications. First, there is a possibility of unbalanced task assignments, resulting in some sites being overloaded with tasks. Second, conditions at a site during scheduling time of a task may be different from the conditions at the site during execution of the task, because each task usually waits in the site's (or worker's) task queue for a while.

We argue that an alternative *worker-centric* scheduling [106, 92], where a scheduling decision to a worker is made only when the worker can start executing the task immediately, is amenable to approaches that exploit locality in file accesses, and addresses both of these issues. In worker-centric scheduling, the times of task assignment to a worker are determined solely by the worker's preference based on its local criteria, e.g., by using policies based on local CPU load, site queue length, time of the day, etc. The task execution begins as soon as the task arrives at the worker. The scheduling problem then becomes the one of designing a global scheduler that assigns the best possible as-yet-unscheduled task to the "best" worker, based on such characteristics as the files already present at the worker's site, and the data required by the unscheduled tasks.

Worker-centric scheduling is on-demand — its scheduling decisions are based on the most up-to-date state of the infrastructure at the time immediately preceding actual task executions. On the other hand, tack-centric scheduling suffers because often times it makes decisions based on the information that will be stale by the time of actual task executions.

There are two options for implementing worker-centric scheduling strategies - either (1) workers could *pull* tasks from a task repository associated with the global scheduler, when the worker's local policies allow it to do so; or (2) the global scheduler could *push* tasks out to workers, depending on the worker's preference. We consider only the pull variant ((1) above) since it is simpler and more practical. Henceforth in this chapter, whenever we use the term "worker-centric", we will be referring to *only* the pull variant of the worker-centric algorithm.

We present the first (to the best of our knowledge) worker-centric scheduling strategies that implicitly exploit the locality of interest in data-intensive Grid applications. We then demonstrate the advantages of worker-centric scheduling over task-centric scheduling for data-intensive Grid applications through experiments. In our worker-centric strategies, each worker requests a task from the global scheduler when convenient to the worker. Upon receiving this request, the global scheduler iterates over the list of as-yet-unscheduled tasks and finds the best task to assign to the worker. The "best" task could be selected according to a variety of metrics, which we discuss later in detail.

We propose three different metrics that consider the different aspects of locality of interest in data-intensive Grid applications, and aim to: (1) maximize the chance of reusing the data, and (2) to minimize the number of file transfers. Our simulation results with *Coadd* confirm that worker-centric scheduling gives better performance than task-centric scheduling in many scenarios. We select *Coadd* for all our experiments in this dissertation because (1) it is difficult to obtain Grid application traces, and (2) *Coadd* is a real Grid application used by several research organizations [72, 96] and it shows many typical characteristics of data-intensive Grid applications. Thus, we believe that our results will hold for many other data-centric Grid applications.

It is important to note that our Grid model is general, and *not* intended to specifically target production Grids such as Grid2003 [28]. Rather, we use the term "Grid" as a generic model, where a set of cooperating sites (a cluster of workers) can be used to execute a job (which consists of tasks sharing read-only data). Also, our scheduling strategies focus only on scheduling data-sharing tasks within a single large job (application), instead of multiple disconnected jobs injected into the system by different users. However, for realistic evaluation, we do simulate the presence of background jobs running concurrently with our main Grid job in our experiments in Section 4.5.

4.2 Taxonomy

Figure 4.1 shows the taxonomy of the scheduling solution space. In this figure, dynamism comes from resource availability, e.g., CPU, memory, disk, and network bandwidth. Scale comes from the amount of data and the number of tasks. Worker-centric scheduling strategies provide a solution for a large-scale dynamic environment (the upper right region). Task-centric scheduling strategies are not suitable for this environment due to the reasons discussed in Section 4.3.5. How-

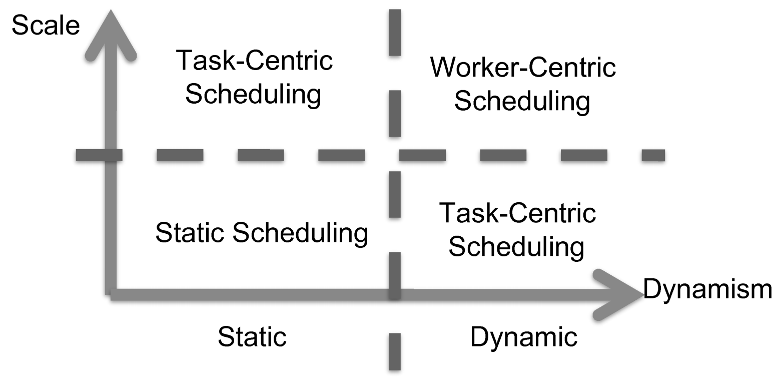

Figure 4.1: Taxonomy of the Scheduling Solution Space

ever, task-centric scheduling strategies provide solutions for a large-scale static environment and a small-scale dynamic environment. This is due to the fact that premature scheduling decisions cannot be made in a static environment (the upper left region), and that unbalanced task assignment is not a significant issue in a small-scale environment (the bottom right region). Finally, if the environment is small-scale and static, one can always pre-compute the perfect schedule beforehand without much overhead (the bottom left region).

4.3 Background and Basics

In this section, we motivate the scheduling problem by presenting the characteristics of data-intensive applications. We then elaborate on the two types of schedulers mentioned: task-centric and worker-centric. Lastly, we discuss scheduling issues for data-intensive applications.

4.3.1 Characteristics of Data-Intensive Applications

We discuss characteristics of data-intensive applications here to motivate the problem. As a real example, we use one particular application, *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [72, 96]) in our discussion.

In general, tasks in a data-intensive application access a large set of files, thus data transfer time significantly affects the entire execution time (i.e. data-intensive applications are network-bound [21, 94]). In addition, the tasks have a high degree of data-sharing among them, which gives an opportunity to reuse data in local storage [12, 21, 72, 86, 94].

For example, *Coadd* is a spatial processing application that has 44,000 tasks accessing 588,900 files in total. It is reported by Meyer *et al.* [72] that when it was run on Grid3 [28] with over 30 sites and 4,500 CPUs, it took roughly 70 days to complete. One of the reasons for the observed long completion time was the

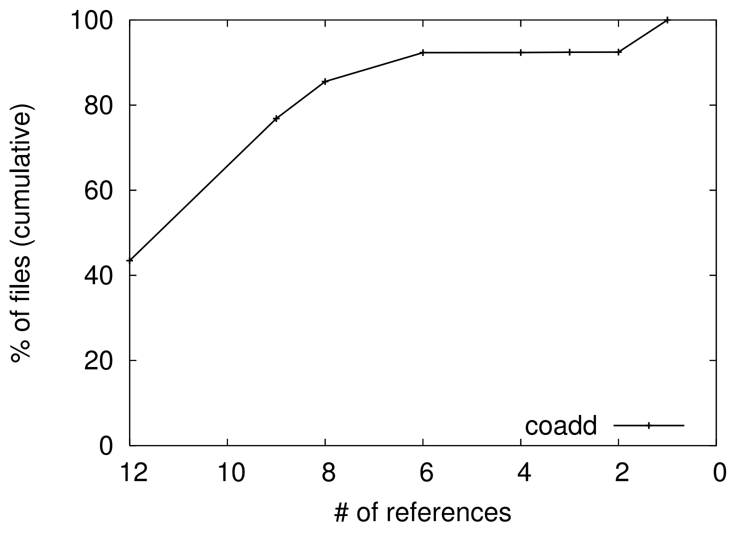

Figure 4.2: Coadd file access distribution. Note that the x-axis is in decreasing order, so each point in the CDF represents the minimum number of files accessed.

large number of files necessary for each task. Meyer *et al.* [72] state that these characteristics would also be expected in other spatial processing applications.

Our analysis of *Coadd* indeed confirms the characteristics of data-intensive applications. In *Coadd*, each task accesses a different number of files ranging from 36 to 181, and approximately 124 files on average. Moreover, roughly 90% of files are accessed by 6 or more tasks, as shown in Figure 4.2. If we assume that each file is fixed at 5MB as in [72], then the total size of all the files is roughly 2.8TB, and each of 44,000 tasks potentially requires 620MB of data transfer on average and up to 905MB in the worse case for each execution. Considering the number of tasks and size of data transfers, it is desirable to reduce the redundant file transfers.

To show locality in *Coadd*, we first pick 1,000 sample pairs of files (say, A and B) accessed by *Coadd* tasks. We then plot the ratio between the actual number of tasks accessing both files, and the expected number of tasks accessing the same files. Figure 4.3 shows the result. The former (the actual number, say, C) is directly counted from our *Coadd* workload, and the later (the expected number) is derived from $\frac{a}{T} \times \frac{b}{T} \times T$, where T is the total number of tasks, and a, b are the numbers of tasks accessing A and B, accordingly. The Y-axis shows $C/(\frac{a}{T} \times \frac{b}{T} \times T)$. As we can see, the values are much larger than 1, which means that the number of tasks that access the same pair of files is much larger than statistically expected.

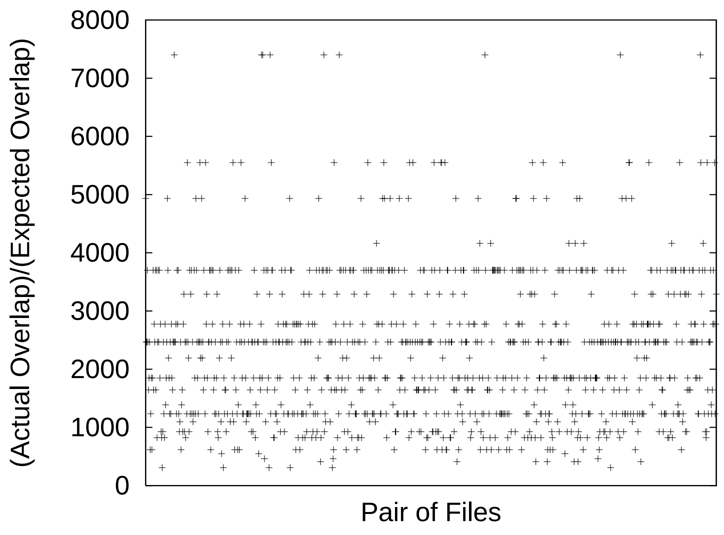

Figure 4.3: Locality of interest in Coadd.

4.3.2 System Model

Before comparing task-centric to worker-centric solutions, we present our system model. We assume that:

1. A *job* is defined as an application composed of multiple parallel *tasks*. Each task does not need to communicate with other tasks in order to proceed (i.e., a job is a Bag-of-Tasks [94]). However, tasks do share read-only files (data). These files are provided a priori along with the job specification.
2. There are multiple sites. Each site has at least one computation server or *worker* (and possibly multiple workers), and one data server to store data locally. We further assume that there is only one *data server* (or *local storage*) per site. If there are multiple data servers at a site, we consider all these data servers as combined storage. Storage size at a site is limited.
3. The data server of a site receives all file requests from the workers in the same site, and sends batch file requests for the missing files to the external file server. The data server processes requests one by one. This is more efficient than simultaneous requests, given the bandwidth limits.
4. Each task issues exactly one batch file request.
5. A worker starts executing a task by transferring all the files necessary for the task to the local data storage. After the transfer is over, the worker begins the actual computation of the task.
6. There is one external (global) scheduler that contains information about all tasks and gives tasks out on-demand to workers. Also, there is an

external file server that has all the files necessary for all tasks, and hands them out to data servers on-demand.

7. Intra-site communication costs are negligible compared to inter-site communication costs.

8. In order to simplify our exposition, we will henceforth assume that all files are equally-sized. However, all our algorithms can be easily extended to variable sized files, by modifying the considered metrics to reflect the data size rather than the number of files.

We use the following two terms throughout this chapter:

- *Makespan* [82] is the total execution time of the job in consideration. This is the main metric for performance measurement.

- *Utilization* of worker A is defined as, *(total computation time of A) / (total execution time of A)*.

- A task and a local storage (i.e. the data server at a site) are said to *overlap* with each other, when at least one file necessary for the task is already present in the local storage. We use the term, *overlap cardinality*, to indicate the number of overlapping files.

The main goals for a scheduling algorithm are then to: (1) reduce the makespan, (2) reduce the number of files transferred to sites, and (3) increase the utilization at workers.

4.3.3 Task-Centric and Worker-Centric Schedulers

We elaborate two types of schedulers, namely, task-centric schedulers and worker-centric schedulers. Figures 4.4 and 4.5 show illustrations of worker-centric and task-centric scheduling. In essence, this categorization is based on whether or not a scheduling strategy considers immediate task execution of a worker after a task assignment.

Concretely, a scheduler is *worker-centric*, if the task assignment to a worker is done when the worker can start executing the task immediately. As mentioned before, we consider only the pull-based variant of worker-centric scheduling and the term "worker-centric" refers to this pull-based variant of worker-centric scheduler throughout this chapter. This variant has each worker *pull* a task from a task repository associated with the global scheduler, when its *local policies* allow it. These local policies may be a function of CPU load, free RAM space, time of day, etc. For instance, a site could have a policy that Grid jobs are executed only over night or at a specific time of the day. Another policy might state that a site could execute Grid jobs only when the average CPU load has been below a specified threshold for a while. This architecture is similar to a server-client architecture - a worker requests a task to the scheduler, and the

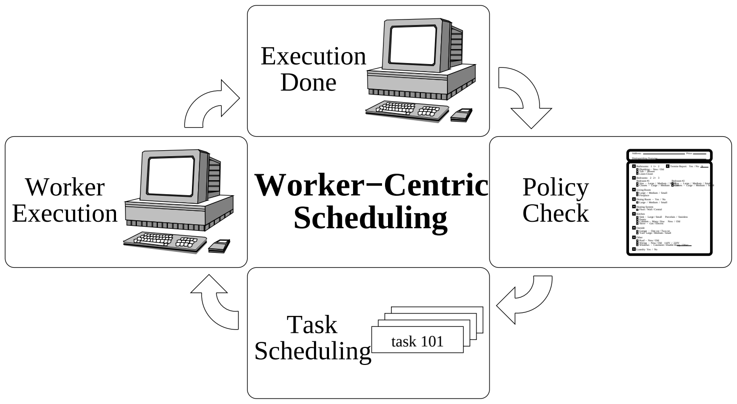

Figure 4.4: An illustration of Worker-Centric Scheduling

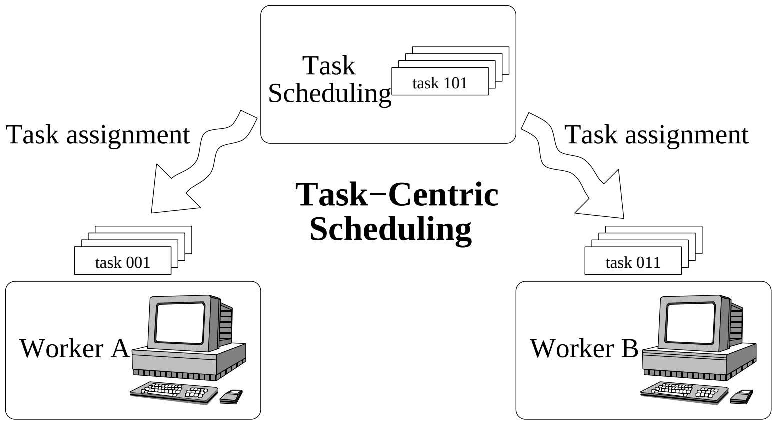

Figure 4.5: An illustration of Task-Centric Scheduling

scheduler finds the "best" task for the worker according to a set of metrics and local policies of the worker. One example of this type of worker-centric strategies is the traditional *workqueue* algorithm, which dispatches a task in FIFO order to an idle worker [18].

On the contrary, a scheduler is *task-centric*, if a task assignment is done without considering whether or not the worker can execute the task immediately. For a given set of tasks and a set of workers, the global scheduler chooses the best match (based on its certain metrics other than immediate task execution) between workers and tasks, and assigns each task to the best worker. Each worker has a task queue and executes the tasks in the queue one by one; an empty queue means the corresponding worker is not executing tasks for that job. Typical metrics used by schedulers are CPU load, network bandwidth, data overlap, etc. For example, scheduling strategies in [86] and storage affinity-based schemes [21] are task-centric.

Since our focus in this dissertation is to show the effectiveness of worker-centric scheduling in exploiting locality compared to task-centric scheduling, we do not discuss various policies of worker-centric scheduling further. In Section 4.5, we first evaluate our task-centric and worker-centric strategies using a simple policy called *always available* - a worker requests a task from job X immediately after it finishes the previous task from the same job X. Later, to consider the effect of slowdown due to background CPU load, we experimentally study the effect of local jobs at individual workers (which might be submitted by local users or through other schedulers) - these background jobs run concurrently with tasks of the Grid job under consideration.

4.3.4 Scheduling Issues for Data-Intensive Applications

Several previous studies have identified that reusing data in local storage gives a dramatic performance improvement for data-intensive applications [12, 72, 86, 94]. Among others, studies by Ranganathan *et al.* [86] and Santos-Neto *et al.* [94] propose various task-centric scheduling strategies for data-intensive applications. Their studies suggest that making scheduling decisions based on data reuse indeed improve performance over other scheduling strategies that consider various different metrics altogether. Broadly, both types of strategies calculate and use the overlap cardinality (either the number of files or bytes) between all possible task-site pairs, in order to make the scheduling decisions.

The reason why schedulers considering overlap cardinality work better is intuitive. As we state in Section 4.3.1 and show in Figure 4.2, (a) data transfer time significantly affects the entire execution time of a data-intensive application, and (b) tasks have a high degree of data-sharing among themselves. This strategy also works well in the real world because data location is relatively static and easy to obtain compared to dynamic metrics such as network bandwidth and CPU loads [94].

4.3.5 Problems of Task-Centric Scheduling and Possible Solutions

We observe two problems from task-centric scheduling strategies. These problems are significant because data replication and task replication [86, 94] never address the second problem, although the first problem can be avoided by both mechanisms.

1) **Unbalanced Task Assignments**: As mentioned by Ranganathan *et al.* [86], task-centric scheduling with data reuse has the problem of overloading certain sites with popular files. Since the overlap cardinality is the primary metric when assigning a task, workers with popular files may be assigned more tasks than the workers with less popular files. Since this problem is inherent in task-centric scheduling, other mechanisms need to be used to avoid the problem, e.g., data replication [86] and task replication [94].

With data replication, the system keeps track of the popularity of each file. If a file's popularity exceeds the pre-determined threshold, it is replicated to other sites. Thus, data replication helps to distribute the load of sites with popular files [86].

Task replication can also help to distribute the unbalanced load caused by popular files. With task replication, the scheduler first distributes its tasks according to the overlap cardinality. Once the initial assignment is done, the scheduler waits until at least one worker becomes idle. Then it picks a task already assigned to a worker and replicates it to the idle worker. If one of the workers finishes the task, the other worker cancels the task. The process is repeated whenever there is an idle worker. This strategy, called *storage affinity*, is proposed and evaluated by Santos-Neto *et al.* [94]. They show that a task-centric scheduler with data reuse and task replication performs better than other scheduling strategies with dynamic information such as CPU loads and available bandwidth.

2) **Long Latency between scheduling and execution**: Task-centric scheduling typically has long latency between scheduling and execution. The following two reasons cause this problem - (1) Since each worker accepts tasks passively from the scheduler and stores received tasks in its queue, there is latency between task assignment time and the actual execution time. (2) Since storage at a site is limited in size, some files required by a task may have been replaced by other required files between the scheduling and execution times of the task.

Therefore, it is possible that a worker was assigned a task because it had some files needed by the task, but at the time of execution, the worker might no longer have some of those files. This "premature scheduling decision" can cause performance degradation with small storage sizes as we show in Section 4.5.

4.3.6 Advantages of Worker-Centric Scheduling

In comparison to the above approach, worker-centric scheduling does not suffer from the unbalanced task assignment problem because a worker requests a new task to the scheduler only when its local policies allow it to execute a task. This means that it is not necessary to have other mechanisms to resolve the issue. Therefore, a worker-centric scheduler only needs to consider its scheduling metric, which leads to a simpler scheduler design.

In fact, both data replication and task replication are orthogonal mechanisms to improve performance in worker-centric schedulers. Thus, they might help the performance of worker-centric schedulers, but are not necessary. However, task-centric schedulers *require* other mechanisms because unbalanced task assignment caused by popular files actually *hurts* the performance of task-centric schedulers [86].

In addition, worker-centric scheduling has short latency between scheduling and execution compared to task-centric scheduling. This arises because w.r.t. a worker, this is a *just-in-time* scheduling policy. Each worker executes a task

```
while(forever):
    req = GetNextRequest()
    if taskQueue is empty:
        wait for a task
    for each task t in taskQueue:
        CalculateWeight(t)
    t = ChooseTask(n)
    ReturnRequest(t)
```

Figure 4.6: Pseudo-code of the basic algorithm. The global scheduler performs this algorithm whenever a worker requests a task.

as soon as the task has arrived at the worker. Thus, it does not suffer from the premature scheduling decisions.

In Section 4.4, we focus on worker-centric scheduling strategies and propose various metrics that consider data-reuse. We also show in Section 4.5 that worker-centric scheduling without additional mechanisms can achieve better performance in many scenarios than task-centric scheduling with additional mechanisms.

4.4 New Worker-Centric Scheduling Algorithms

In this section, we present our new worker-centric scheduling algorithms that attempt to exploit locality by considering data-reuse during scheduling.

4.4.1 Basic Algorithm

Our basic algorithm is shown in Figure 4.6. It is a worker-centric algorithm, with one global scheduler and multiple sites, each containing multiple workers. Upon receiving a request from a worker, the global scheduler calculates the weight of each as-yet-unscheduled task (*CalculateWeight()*) and chooses the best task to assign to the requesting worker (*ChooseTask()*). Notice that worker requests are processed sequentially. *CalculateWeight()* and *ChooseTask()* take into account the set of files already at the worker's site, and the set of files required by the worker, thus attempting to exploit locality. These are detailed next.

As mentioned in Section 4.3.2, for simplicity of exposition, we restrict our discussion to tasks that share equally-sized files. However, our algorithms can easily be extended to varied file sizes by merely considering a "file block" (instead of a file) as a unit of sharing among tasks.

4.4.2 *CalculateWeight()*

CalculateWeight() calculates a weight for each each task in order to exploit the locality of file access. This weight can be calculated via one of three possible metrics - *Overlap*, *Rest*, and *Combined.* Before further discussion, we need to define the following terms and conditions:

- T: the set of all unscheduled tasks that the scheduler currently has in its queue.
- F_t: the set of overlapping files between task t and the data storage at the site of the requesting worker.
- $|t|$: the total number of files required by task t.
- r_i: the number of past references of the file i at the local storage (i.e. data server) of the requesting worker, i.e., the number of previously completed tasks at the site that accessed file i.
- Task t is said to be *better* than task t', when $CalculateWeight(t) > CalculateWeight(t')$

Now we consider three metrics that could be used by the scheduler.

1. *Overlap*: This metric is the overlap cardinality (discussed in Section 4.3.2). It counts the number of files that are needed by the given task and are already present in the local storage of the requesting worker. Thus, $|F_t|$ is the overlap cardinality. Intuitively, the goal of this metric is to maximize the chance of reusing the data already stored in the local storage of the requesting worker. As mentioned before, this metric is the primary metric of task-centric scheduling strategies in the previous studies.
2. *Rest*: This metric is the inverse of the number of files that need to be transferred in order to execute the given task, i.e., $rest_t = \frac{1}{|t|-|F_t|}$. Intuitively, the goal of this metric is to minimize the number of files that need to be transferred. This is a complement of *overlap* metric conceptually.
3. *Combined*: For this metric, each data server keeps for each file the number of past references, i.e., the number of previously completed tasks at the site that have accessed the file. It combines these past references and *rest* using an equation defined as follows. We define ref_t to be the total references of all the overlapping files of task t at the worker's site, i.e., $ref_t = \sum_{i \in F_t} r_i$. Now, let $totalRef$ be the sum of all ref_t over all t in T (w.r.t. the requesting worker's site), i.e., $totalRef = \sum_{t \in T} ref_t$. Also, let $totalRest$ be the sum of all $rest_t$ over all t in T, i.e., $totalRest = \sum_{t \in T} rest_t$. Then, $combined_t = \frac{ref_t}{totalRef} + \frac{totalRest}{rest_t}$. Intuitively, this metric attempts to exploit locality of file access, and thus minimize both the number of files that need to be transferred as well as to prefer workers that accessed the same files in the past.

4.4.3 *ChooseTask()*

Since the scheduler greedily assigns a task to a worker based on the value of *CalculateWeight()*, there is some possibility of sub-optimal assignments. One reason for this is the sequential nature of such worker-centric scheduling. For example, suppose worker h is a better candidate to execute task t than worker h', but worker h' requests a task right before worker h requests a task. In this case, the scheduler will assign task t to worker h' rather than h. This can happen quite often especially for data-intensive applications - since file transfer time is usually long after a task assignment, the global scheduler can receive a number of requests from different workers during the transfer. So it is possible that a better worker comes by while the previously-assigned worker has not even started processing, i.e., it is still awaiting the file transfer to complete.

To take these types of scenarios into account, we use randomization when choosing a task through *ChooseTask(n)*. *ChooseTask(n)* then executes two steps. First, it chooses a set, T_n, of the best n tasks among all tasks (i.e., tasks with n largest values calculated by *CalculateWeight()*), where n is a parameter. Second, it chooses one task among the best n tasks with a probability proportional to the *CalculateWeight()* values. Thus the probability of choosing task t is,
$P_t = \frac{CalculateWeight(t)}{\sum_{k \in T_n} CalculateWeight(k)}$.

If $n \geq 2$, this is a randomized approach. If $n = 1$, this is a deterministic approach that greedily chooses the best task. Notice that this procedure, in combination with *CalculateWeight()*), attempts to implicitly exploit the locality of file access.

4.4.4 Reducing Communication Cost

In order to make the scheduling decision for a requesting worker, we assumed above that global scheduler has all the necessary information about files currently stored at the requesting worker's site, namely, (1) names of files that the data server is currently storing, and (2) the reference count for each of these files. In other words, we assumed that the global scheduler implicitly maintains a *reference table*, as shown in Figure 4.7. In this table, there is one column per file in the job, and one row per site in the Grid. Each entry (i, j) specifies "reference count" for file j at site i. The reference count denotes the past references of file j at site i and also shows the presence of file j at site i.

There are two efficiency sub-problems that need to be addressed: how to maintain this table efficiently, and how to keep it updated with minimal network bandwidth overhead. The first sub-problem is addressed by having the global scheduler maintain a local hash table per site (row in the reference table), containing the names of files currently stored at that site along with their reference counts. File names are the keys for this data structure. Notice that lookup, insertion and deletion into this hash table are each $O(1)$ on expectation.

File ID / Site ID	file0	file1	file2
site0	N / A	12	6
site1	N / A	N / A	N / A
site2	10	1	5
site3	8	N / A	1
site4	N / A	N / A	1

Figure 4.7: A reference table example. Each entry contains a reference counter. We use N/A to indicate that the entry is not present for the sake of demonstration.

The bandwidth problem is addressed by piggybacking each task-requesting message, from a worker to the global scheduler, with the set of file names that have been replaced at the data server of the worker's site *since the last request from the same site*, i.e., the list of names of files that were eliminated from the site's data server since its last request. The global scheduler deletes these file names from the hash table for that site. Then, once it makes the requested scheduling decision for the worker, the new files required by the assigned task are inserted into this hash table and the corresponding reference counts are initialized to 1. For all other files that are already present at the site and required by the assigned task, the global scheduler increments corresponding reference counts by 1. In this way, the communication between the worker to the global scheduler is reduced to only once per request no matter how many files are added and/or deleted from the site's data server.

This approach is very efficient for our considered cases. In spite of file-sharing across tasks, each task in our observed data-intensive applications typically accesses a relatively small number of files compared to the total number of files for a given application. For example, in the *Coadd* traces, no task accesses more than 181 files out of a total of 588,900, in spite of data-sharing. This also means that at most 181 files are replaced between two consecutive requests. Thus, assuming file names are 4 bytes each, the additional information piggybacked along with a worker request is at most 724 bytes in size, which is reasonably small.

4.4.5 Complexity

If $|T|$ is the number of currently waiting tasks, and $|I|$ is the maximal number of files required by any task, then the total communication complexity of our algorithm arises out of the per-request piggybacked information as described in the previous section - this is $O(|I|)$ per task assigned to a worker. Similarly, the computation complexity is $O(|I| + |T| \times |I|)$ per task assigned to a worker, with

Unit computation cost	1,000 MFLOPS
capacity of each data server	6,000 files
number of workers per site	1
number of sites	10
file size	25 MB

Table 4.1: Default parameters for experiments

the first term accounting for the hash table operations, and the second one for the scheduler's operation itself. This is $O(|T| \times |I|)$, and more efficient than task-centric strategies used by Ranganathan *et al.* [86] and Santos-Neto *et al.* [94], which compare all pairs of tasks and sites. Their complexity is $O(|T| \times |I| \times |S|)$ (where $|S|$ is the total number of sites), even assuming the use of a hash table similar to that described in the previous section. Our approach is more efficient because we do not assume any knowledge (a priori or otherwise) about sites other than the requesting worker's.

4.5 Evaluation

In this section, we present our evaluation of worker-centric scheduling strategies and discuss the results.

4.5.1 Simulation Overview

To demonstrate the advantages of worker-centric scheduling over task-centric scheduling, we implement our basic algorithm with three metrics on the SimGrid simulator [60]. For comparison, we also implement *storage affinity* [94], a task-centric scheduling with data reuse and task replication.

We vary five main parameters in our experiments - (1) capacity of each data server, (2) number of workers per site, (3) computation time, (4) number of sites, and (5) file size. The default values for these parameters are summarized in Table 4.1, and used in our experiments unless otherwise noted. However, we vary each of these 5 parameters in our experiments to see the effects of different values. Throughout the experiments, the computation time of each task is linear to the number of files (i.e., *(number of files) * (unit computation cost)*).

Our main workload is *Coadd* (Sloan Digital Sky Survey southern-hemisphere coaddition [72, 96]). As mentioned before, *Coadd* is a spatial processing application that has 44,000 tasks accessing 588,900 files in total. We use only the first 6,000 tasks of *Coadd* to finish our experiments in a reasonable amount of time. A total of 53,390 files are accessed by these 6,000 tasks. More workload characteristics are shown in Table 4.2. Although we only use the first 6,000 tasks, our workload characteristics remain similar to Figure 4.2.

Total number of files	53,390
Max number of files needed by a task	101
Min number of files needed by a task	36
Average number of files needed by a task	78.4327

Table 4.2: Characteristics of *Coadd* with 6,000 tasks

	Avg (MB/s)	Std dev
Topology 0	4.418	5.416
Topology 1	4.631	6.734
Topology 2	3.858	2.599
Topology 3	3.432	1.432
Topology 4	3.932	2.778

Table 4.3: Average bandwidth and standard deviation between a site and the file server

4.5.2 Simulation Environment

Network Configuration: We use 5 different topologies, each with 90 sites, generated with Tiers topology generator [24]. Tiers is a structural topology generator that generates hierarchical cluster topologies. We use Tiers because it is well-supported by SimGrid, the simulator we use in our experiments. Only a subset of 90 sites are used in each experiment. For each topology, there are one global scheduler and one global file server which stores all the files. At each site, there are 30 workers and 1 data server. All 30 workers and the data server in a site share outgoing links to the global scheduler and the file server. Intra-site communication cost (cause by bandwidth and latency) is negligible. Inter-site communication cost is determined by underlying network links generated by Tiers. Each path between two sites consists of multiple network links, and the bandwidth and latency of each of these links determine the inter-site communication cost. Table 4.3 summarizes the average and standard deviation of bandwidth values between a site to the file server for each topology. Each worker's computation capacity (in MFLOPS) is chosen randomly from top500 list [104] and is uniformly divided by 100, since most of the 500 machines are too powerful. Each experiment is performed with 5 different topologies and the results are averaged over the 5 runs.

Background Jobs: We perform our experiments with background jobs as well as without background jobs. We use background jobs to evaluate the performance of different strategies in the presence of competing applications running on each site. Since a site is typically shared by different schedulers and local users, this gives us a more realistic setting.

We simulate background jobs through varying each worker's CPU load. A worker is always executing a task for the Grid job in question, but in addition it is also running background jobs. The background jobs thus slow down the execution of the task at the worker. The load due to these background jobs is simulated as follows: at each worker, once every 5 minutes, the background CPU load is picked as a floating-point number uniformly at random between 0 to 100. This becomes the worker's background load over the next 5 minutes. Considering that the total job execution time in our simulations is *O(tens to hundreds of days)*, we consider the granularity of 5 minutes to be fine-grained enough to capture dynamics of background jobs.

4.5.3 Algorithms

We compare the following 6 different algorithms. The first algorithm is task-centric; the rest are worker-centric.

1. *task-centric storage affinity* : The task-centric scheduling with data reuse and task replication [94]. This is a deterministic algorithm.
2. *overlap* : Our basic algorithm with the *overlap* metric. This is a deterministic algorithm.
3. *rest* : Our basic algorithm with the *rest* metric. $n = 1$ for *ChooseTask(n)*. This is a deterministic algorithm.
4. *combined* : Our basic algorithm with the *combined* metric. $n = 1$ for *ChooseTask(n)*. This is a deterministic algorithm.
5. *rest.2* : Our basic algorithm with the *overlap* metric. $n = 2$ for *ChooseTask(n)*. This is a randomized algorithm.
6. *combined.2* : Our basic algorithm with the *overlap* metric. $n = 2$ for *ChooseTask(n)*. This is a randomized algorithm.

We have tried different values of n for *ChooseTask()*, but only 1 and 2 give good results. Thus, we only show the results of $n = 1$ and 2.

4.5.4 Capacity Per Data Server

Figure 4.8 shows the makespan (i.e. total execution time) of each algorithm with different capacities of 3,000, 6,000, 15,000, and 30,000 files in the presence of background jobs. We do not present the results without background jobs since the performance characteristics are similar. Randomized algorithms, *rest.2* and *combined.2*, perform the best in all cases, which confirms that it avoids sub-optimal scheduling decisions described in Section 4.4.3. *Storage affinity* has a negative performance impact with smaller capacities because of premature scheduling decisions as discussed in Section 4.3.5. However, the performance becomes comparable to worker-centric scheduling as the storage size increases.

Figure 4.8 also shows the importance of considering the number of files that actually need to be transferred. Among the worker-centric strategies, *overlap*

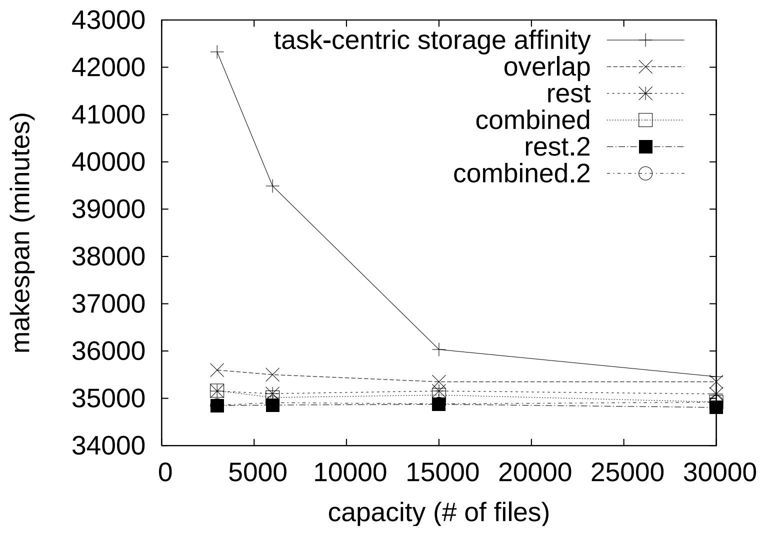

Figure 4.8: Makespan of each algorithm with different capacities of 3,000, 6,000, 15,000, and 30,000 files (with background jobs).

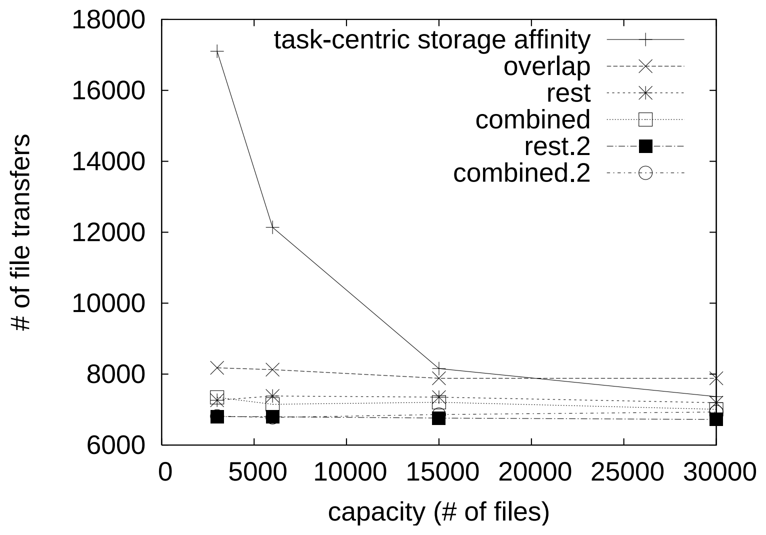

Figure 4.9: File transfers of each algorithm with different capacities of 3,000, 6,000, 15,000, and 30,000 files (with background jobs).

performs worse than other metrics because it does not explicitly consider the number of file transfers, while other metrics do. As we can see in Figure 4.9, *overlap* usually has higher number of file transfers than other metrics. Overall, the randomized algorithms appear to perform the best (i.e., *rest.2* and *combined.2*).

The makespan of each metric in worker-centric scheduling shows steady behavior because the working set of a *Coadd* task is not big. As is shown in Table 4.2, a task needs 101 files at most, and roughly 78 files on average. Thus, a storage with 3,000 files can actually give similar performance as a storage

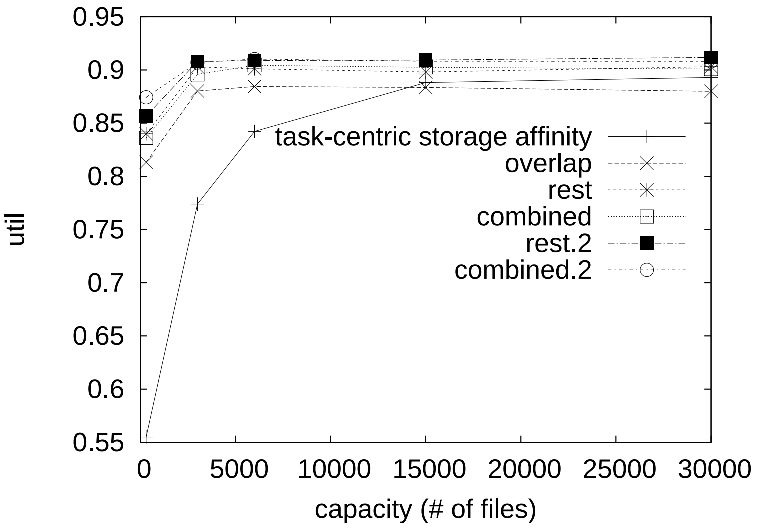

Figure 4.10: Average utilization at worker, with different capacities of 3,000, 6,000, 15,000, and 30,000 files (with background jobs)

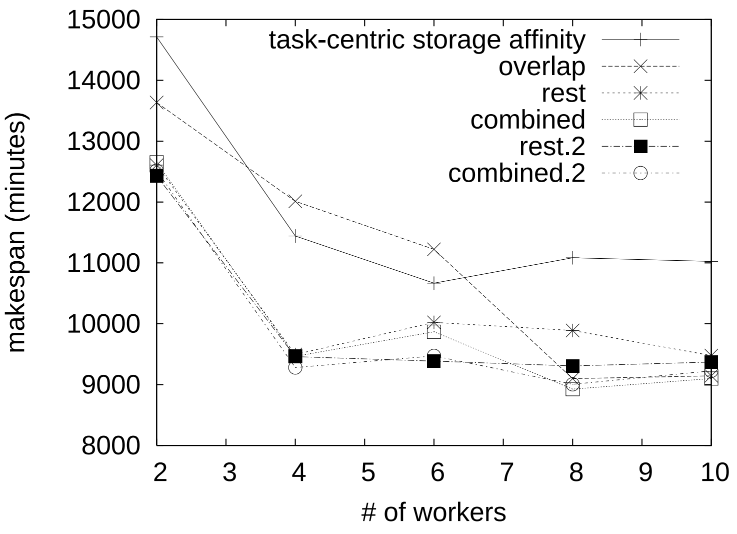

Figure 4.11: Makespan with different numbers of workers at a site (without background jobs)

with, say, 10,000 files.

Figure 4.10 shows the average utilization of each worker (accounting for both the main Grid job and the background jobs). For *task-centric storage affinity*, the low utilization with the capacity of 3,000 files means that the greedy approach requests files more often than other strategies. This behavior shows (1) that randomized decisions can be better than taking what looks as the "best" decision at some particular time and, again, that (2) the *task-centric storage affinity* suffers from premature scheduling decisions.

Due to the lack of space, we do not present the utilization results without

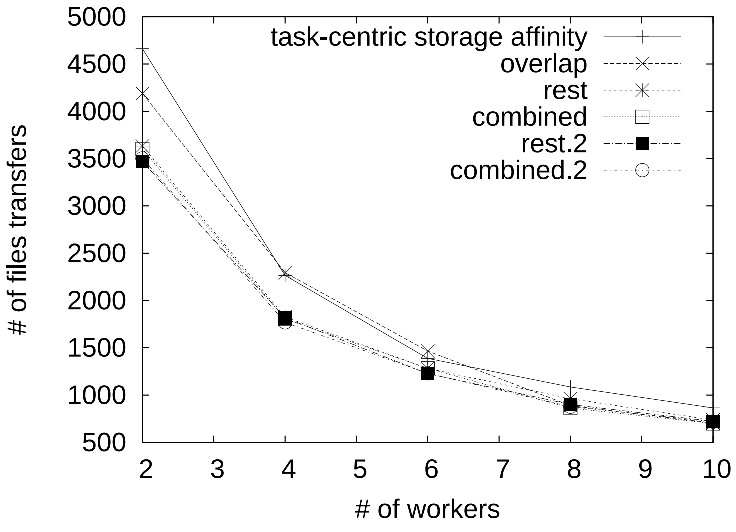

Figure 4.12: Average number of file transfers per worker with different numbers of workers at a site. We only show the results without background jobs, since the presence of background jobs does not show any different behavior.

background jobs here. However, the utilization of each worker with background jobs is slightly higher than that of each worker without background jobs. There are two factors contributing to this result. The first factor is obviously background jobs running on each worker. The second factor is that it takes more time for a worker to finish a task with background jobs. Thus, the utilization goes higher with background jobs.

4.5.5 Number of Workers Per Site

Figure 4.11 shows the makespan of each algorithm with different numbers of workers at a site. *combined.2* performs the best mostly, which shows that minimizing file transfers as well as considering past references helps to reduce the makespan. Overall, worker-centric scheduling metrics perform well with smaller numbers of workers, but *storage affinity* performs well with larger numbers of workers. Also, randomized algorithms that consider the number of file transfers perform better than others.

The makespan of each algorithm flattens as the number of workers increases. In some cases, the performance is worse with more workers (in Figure 4.11)! We can understand the reason behind this behavior with two factors that contribute to the makespan. First, as the number of workers increases at a site, the contention at the data server of the site increases. Since the data server processes each request one by one so as to minimize the redundant file transfers (as mentioned in Section 4.3.2), this contention is unavoidable. This factor has a negative impact on the makespan (i.e. increases it). On the contrary, as the number of workers increases, the number of files that can be shared by

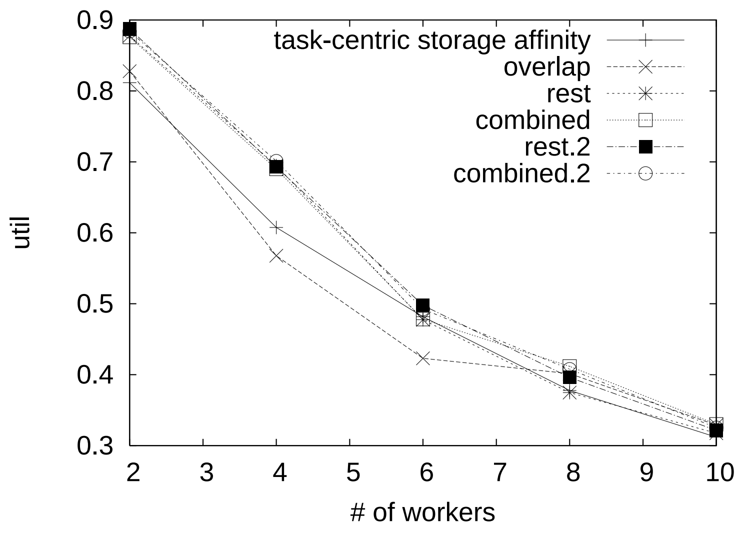

Figure 4.13: Average worker utilization, with different numbers of workers at a site. We only show the results without background jobs, since the presence of background jobs does not show any different behavior.

	waiting time (hrs)	transfer time (hrs)	# of file transfers
2 workers	3.59	30.35	3998.5
4 workers	40.32	45.45	2086.5
6 workers	98.35	33.85	1335.17
8 workers	75.93	18.81	906.38

Table 4.4: Result of the *rest* metric at a site with 2 workers, 4 workers, 6 workers, and 8 workers. All numbers are averages per worker. Note that *rest* shows the worst makespan with 6 workers at a site.

the workers also increases. This factor has a positive impact on the makespan. The interaction of these two factors results in different behaviors of different algorithms.

To validate the reason, Figure 4.12 shows the number of file transfers per worker and Figure 4.13 shows the corresponding utilization. It shows that the average number of file transfers per worker decreases as the number of workers increases. Thus, it shows that good file-sharing is achieved intra-site as the number of workers increases. In addition, Table 4.4 shows the result of the *rest* metric at one particular site with 2, 4, 6, and 8 workers. It shows (1) average waiting time that a file request spends at the data server's waiting queue, (2) transfer time that it takes to transfer all the files from the external file server to the data server, and (3) associated number of file transfers.

In the case of 2 workers in Table 4.4, the contention at each data server and the file server is very low compared to other settings, simply because there are fewer workers. Thus, the waiting time and the transfer time are rather small even though the number of file transfers is high.

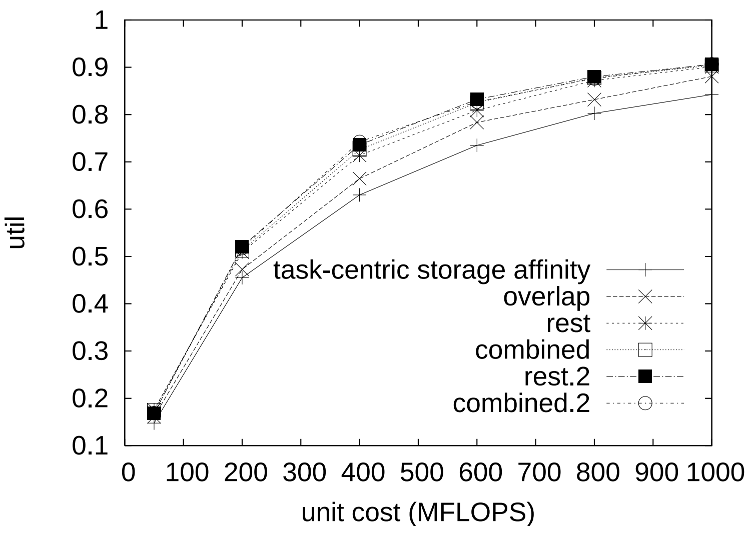

Figure 4.14: Average utilization per worker for different unit computation costs of 50, 200, 400, 600, 800, and 1,000 MFLOPS

We can reason why the performance is sometimes worse with more workers with the data of 4 workers, 6 workers, and 8 workers. If we look at the data in this range, both the average number of file transfers and the average transfer time decrease as the number of workers increases, but the average waiting time peaks at 6 workers. This means that the reduced transfer time is not enough to compensate the increased competition at the data server for *rest* with 6 workers at a site. For the same reason, other algorithms sometimes exhibit a worse makespan with more workers.

4.5.6 Effect of Computation Time

With our default parameter values in Table 4.1, the average utilization per worker is usually more than 90%, which means that each worker spends most of its time on computation. Thus, we perform an experiment with smaller values of unit computation time in order to understand how different computation-to-communication ratios affect the behavior of each strategy. As mentioned before, the computation time of each task is linear to the number of files that it needs to process, i.e., *(computation time) = (number of files) * (unit computation cost)*. We vary the unit computation cost in this experiment.

Figure 4.14 and Figure 4.15 show that our experiment covers a wide range of communication-to-computation ratio. As shown in Figure 4.14, the utilization of each worker (i.e., *(total computation time of the worker) / (total execution time of the worker)*) varies from roughly 0.2 to 0.9. Also, Figure 4.15 shows that the file transfer time (i.e., communication time) takes from roughly 50% to almost 100% of the entire makespan. Thus, our experiment covers a wide range of communication-to-computation ratio, and still captures the characteristic of

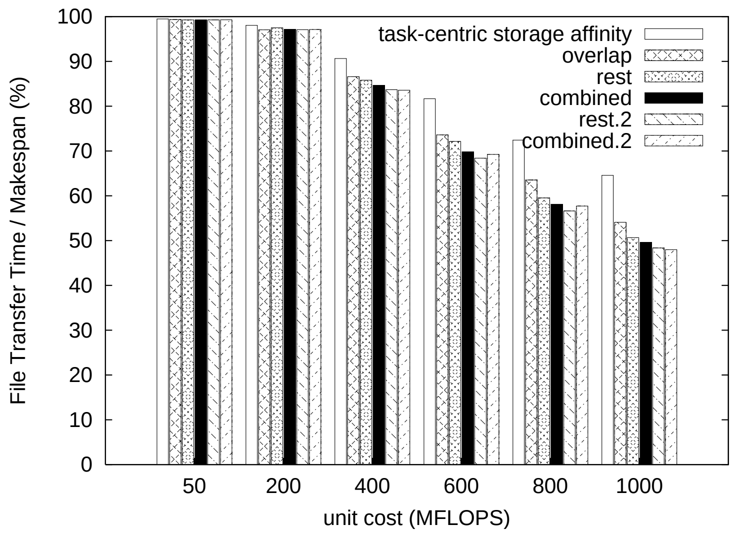

Figure 4.15: Total file transfer time compared to makespan, for different unit computation costs of 50, 200, 400, 600, 800, and 1,000 MFLOPS

long communication time in data-intensive applications. Although Figures 4.14 and 4.15 show the results without the presence of background jobs, the overall behavior remains similar even with background jobs. Note that file transfer time does not directly contribute to worker utilization as in Figures 4.14 and 4.15. The reason is because computation is parallelized, and hence, most workers are busy with doing computation even when the file server transfers files. This explains a seemingly inconsistent behavior of Figures 4.14 and 4.15, in which the file transfer time takes roughly 50% with the unit computation cost of 1,000 MFLOPS in Figure 4.15, even when the average utilization of each worker is roughly 90% in Figure 4.14.

Figure 4.16 shows the makespan (with background jobs) of each algorithm in percentile scale using task-centric storage affinity as a baseline comparison. We do not present the results without background jobs since they exhibit similar behaviors. Overall, we observe that the performance trend remains similar across different strategies even with various computation-to-communication ratios. Worker-centric strategies perform better than the task-centric storage affinity in terms of makespan. In the best case, worker-centric *rest* takes roughly 28% less makespan time than *task-centric storage affinity*. Also, the gap between task-centric storage affinity and other strategies generally becomes wider as the unit computation cost decreases. This is an expected behavior since file transfer time becomes more dominating in total execution time as the unit computation cost decreases.

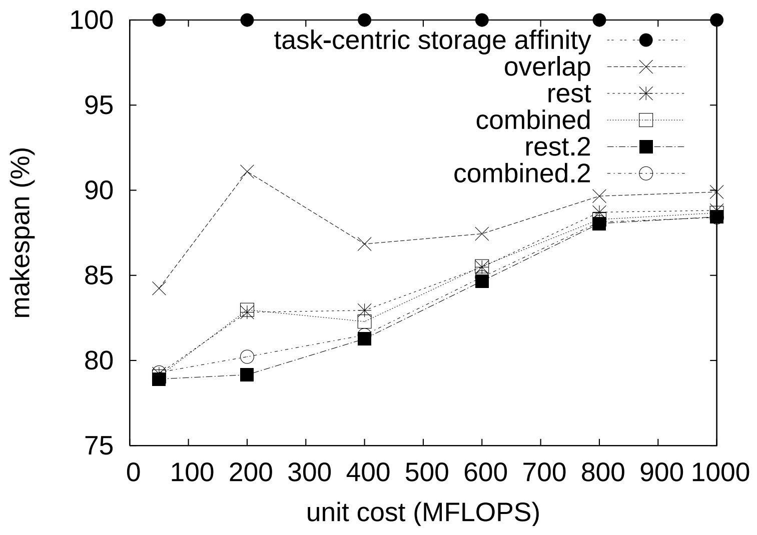

Figure 4.16: Makespan (percentile) of each algorithm with different unit computation costs of 50, 200, 400, 600, 800, and 1,000 MFLOPS (with background jobs)

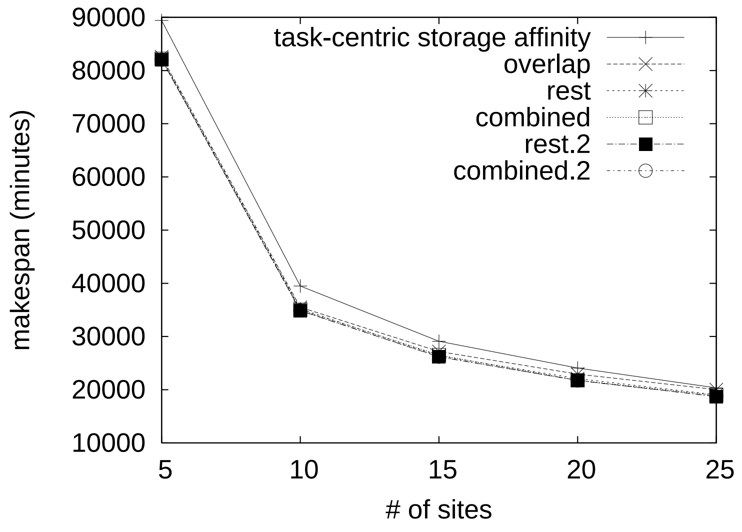

Figure 4.17: Makespan with different numbers of sites (with background jobs)

4.5.7 Number of Sites

Figure 4.17 shows the makespan of each algorithm with different numbers of sites and Figure 4.18 shows the number of file transfers accordingly. Generally, the makespan of each algorithm reduces as the number of sites increases, as expected. *combined.2* performs the best, which again confirms that minimizing file transfers as well as considering past references helps to reduce the makespan. In the best case, *combined.2* takes roughly 17% less makespan time than *task-centric storage affinity*. Randomized algorithms perform better than

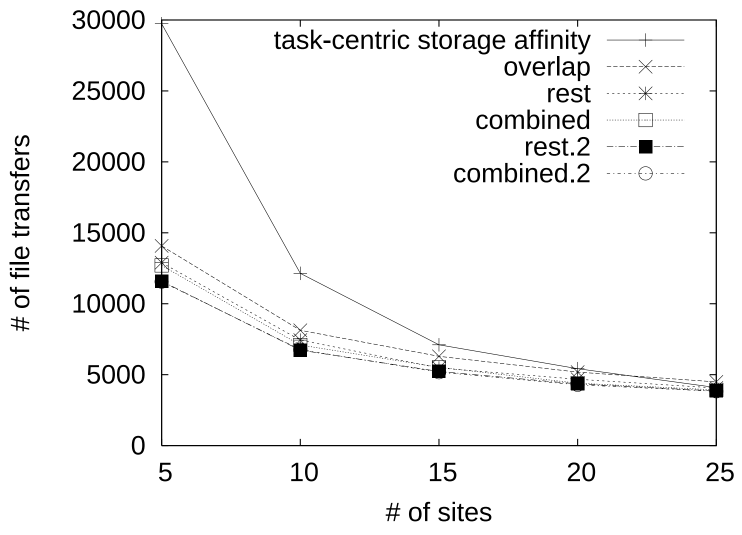

Figure 4.18: Number of file transfers with different numbers of sites (with background jobs)

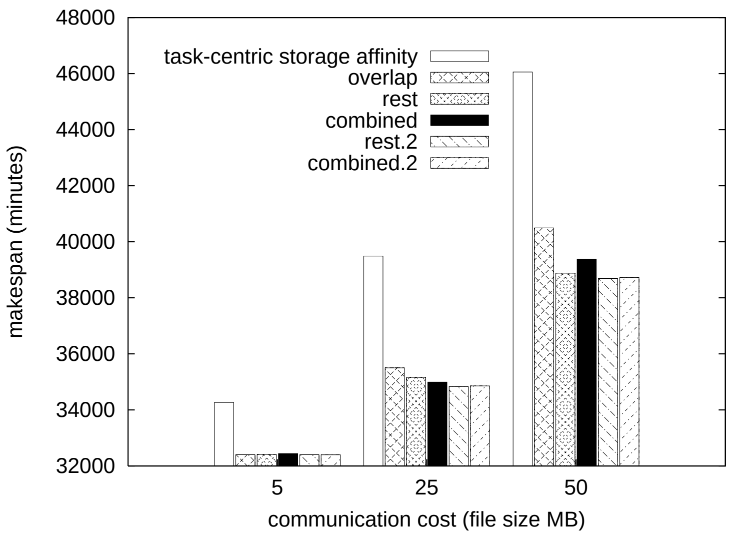

Figure 4.19: Makespan with different file sizes (with background jobs)

deterministic algorithms, which again shows that it avoids sub-optimal scheduling decisions described in Section 4.4.3.

4.5.8 File Size

Figure 4.19 shows the makespan of each algorithm with different file sizes. We choose small (5MB), middle (25MB), and large (50MB) file sizes. The makespan grows almost linearly as the file size grows. Since all algorithms consider files as the primary metric, various file sizes do not result in dramatically different behaviors. *combined.2* shows the best performance just like many other scenarios shown before. The general behavior remains the same even in the presence of background jobs.

4.6 Related Work

Spatial Clustering [72] creates a task workflow based on the spatial relationship of files in the input data set. It improves data reuse and diminishes file transfers by clustering together tasks with high input-set overlap. Two drawbacks to this approach are that (1) it cannot handle new jobs arriving asynchronously, and (2) it is application specific.

Storage Affinity [94] also addresses file reuse for data-intensive applications. The algorithm computes a data affinity value for each task, for each site, according to the input set of each task and the data currently stored at a site's networked storage. To address inefficient CPU assignments, they propose replicating tasks, also based on the storage affinity. The algorithm shows improved makespan and good data reuse, specially when compared to the XSufferage [13] scheduling heuristic.

Decoupling data scheduling from task scheduling was proposed by Ranganathan *et al.* [86]. The work evaluates four simple task scheduling mechanisms and three simple data scheduling mechanisms. Best results are obtained when a task is scheduled to a site that has a good part of its input data already in place, combined with proactive replication of a popular input data-set to a random/least-loaded site.

A pull-based scheduler is proposed by Viswanathan *et al.* [106]. It employs an Incremental Based Strategy, where a scheduler determines how to fraction a job among available workers, based on worker's computing speed and estimated buffer. This work completely ignores data transfer time, and requires knowledge of CPU speed and memory size in all workers.

Rosenberg *et al.* [92] study global scheduling strategies in the Grid-like environments theoretically. Their scheduling strategies focus mainly on *DAGs of tasks*, where tasks are inter-dependent and pre-ordered, and the dependency structure follows DAG (Directed Acyclic Graph). Although they discuss *pull* and *push* strategies, their studies do not assume (1) data-intensive applications (transfer time, storage capacity, data correlation, etc), (2) data-sharing, and (3) task-independence. Thus, the issues are not related to our work.

4.7 Summary

We have presented the motivation, background, design and evaluation of worker-centric scheduling strategies for data-intensive applications in Grids. We have argued that worker-centric scheduling strategies are a natural choice for data-intensive Grid applications due to the scale of data as well as the dynamic nature of resource availability in Grids. Our proposed strategies have been evaluated in large-scale simulations with a real workload trace from Coadd. We have found that metrics considering the number of file transfers generally give better performance over metrics considering the overlap between a task and a storage. We also found that worker-centric scheduling algorithms achieve better or comparable performance to task-centric scheduling. Our results have shown that we can achieve roughly up to 28% reduction in makespan with worker-centric scheduling compared to task-centric scheduling.

Chapter 5

MPIL: An On-Demand Key/Value Store

This chapter presents the motivation, design, analysis, and evaluation of MPIL (Multi-Path Insertion and Lookup). MPIL is an on-demand key/value store that addresses the challenges of machine scale and membership dynamism in a large-scale peer-to-peer setting. We detail how MPIL addresses them mainly in comparison to a well-known existing system, Pastry [93], in this chapter.

5.1 Motivation

Resource location and discovery in distributed systems such as the Grid, cooperative web caching, peer to peer email, etc., all require object insertion and querying mechanisms that are scalable and tolerant to node failures. Our work is motivated by two additional practical concerns that require far more from such object insertion and querying strategies (henceforth, together labeled as "lookup" strategies). These practical concerns are *overlay-independence* and *perturbation-resistance*.

Lookup strategies are usually coupled with the maintenance of an appropriately matched application-layer network ("overlay") among the participating hosts ("nodes" or "peers") on top of the Internet. Each node knows a few other nodes in the overlay according to specific overlay rules, and routes overlay messages such as insertion and querying of files. However, this makes it impossible to deploy a practical peer to peer application (e.g., cooperative web caching) on an *already-existing legacy overlay* (e.g., a Grid network) without first deploying the overlay maintenance protocols associated with the p2p application. These maintenance protocols might increase the overhead, or worse inhibit the performance of, other already-existing protocols in the legacy overlay that already maintain some kind of structure. This motivates the need to develop lookup strategies that are on-demand, i.e., lookup strategies that perform well over the current structure of the underlying overlay, independent of what the actual topology is.

A second and more important practical concern is the resistance to a type of dynamism that we call perturbation. If p2p overlays are to be deployed successfully for a variety of legal applications, the robustness of their behavior under the ordinary kinds of stress experienced by nodes will be a minimum requirement. Perturbation is one such kind of stress. A node is said to be *perturbed* if it is unresponsive for brief periods of time. Perturbation can be caused by many reasons and can occur at several granularities. Concurrent competing applications running on the host, packet buffer overflows, and congestion, can cause short-term perturbation, where the node is unresponsive for up to a few seconds. Longer-term perturbation with unresponsiveness granularities of several minutes or hours can be caused by user churn, i.e. rapid node departures and arrivals of users, a phenomenon present in Grid applications and file sharing overlays. We model perturbation by nodes whose availability *flaps* periodically, and study the effect of such periodic flapping on the lookup success rate. Success rate is the fraction of successful replies to lookups injected into the overlay.

Currently, overlays are either unstructured or structured. Unstructured overlays such as Gnutella [33] use flooding to query object replicas. While this strategy is perturbation-resistant and overlay-independent, it is neither efficient nor scalable. Structured overlays include Chord [100], Pastry [93], Tapestry [111], Kelips [36], Viceroy [67]. Also called DHTs (Distributed Hash Tables), these overlays map both objects and nodes to keys by using hash functions. Lookups are then routed within this overlay by using a routing algorithm that selects *one* next hop node at each step, based on key values of the destination and the current node. While structured overlay lookups are efficient and scalable, they are not overlay-independent because the routing algorithm is usually coupled with an appropriate overlay structure with maintenance strategies. For example, Pastry uses prefix routing based on key values, and nodes in the underlying overlay select neighbors based on the same metric. Recent studies reveal that many structured overlays may be churn-resistant, but we show that they may not be resistant to more general perturbations.

We present a new resource location and discovery algorithm called MPIL (Multi-Path Insertion/Lookup) that provides both overlay-independence and perturbation-resistance. MPIL achieves these goals by using a deterministic routing metric (like DHTs), but by exploiting *limited* redundancy (like unstructured p2p systems). The deterministic routing metric used is based on the hash value of keys (objects and nodes), just like Pastry or Chord, but unlike those systems, does not assume any characteristics about the underlying topology. This routing requires the use of limited redundant routing of lookups to insert and query multiple replicas of an object pointer. This limited redundant routing also provides perturbation-resistance. Put together, MPIL provides a cost-effective and convenient way of developing and deploying robust p2p applications that target any type of overlay. In a sense, the techniques of limited redundancy and overlay-independence achieves the best of both worlds from both structured and unstructured overlays.

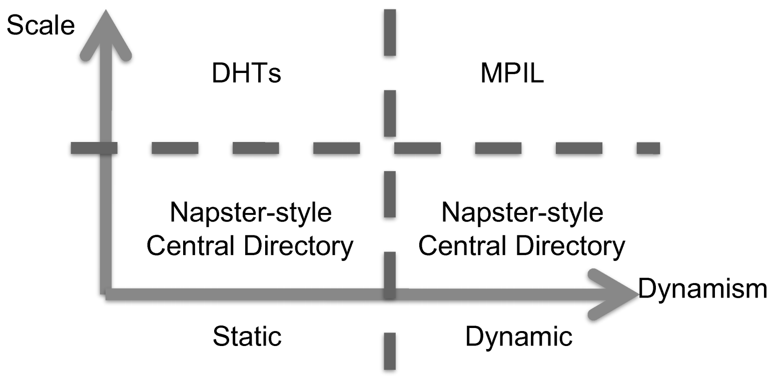
Scale
DHTs
MPIL
Napster-style Central Directory
Napster-style Central Directory
Dynamism
Static
Dynamic

Figure 5.1: Taxonomy of the Peer-to-Peer Lookup Solution Space

5.2 Taxonomy

Figure 5.1 shows the taxonomy of the peer-to-peer lookup solution space. In this plot, dynamism comes from perturbation, and scale comes from the number of peers. MPIL provides a solution for a large-scale dynamic environment (the upper right region). This is due to the fact that MPIL does not spend maintenance traffic. In comparison, DHTs typically need to manage a certain overlay structure, which requires considerable maintenance traffic (discussed in Section 4.5). Thus, DHTs are more suitable for a large-scale static environment, where less maintenance is required (the upper left region). In a small scale environment, a central directory can be used to provide the lookup functionality (the bottom two regions).

5.3 Effect of Perturbation on Pastry

To study the effect of perturbation on Pastry, we conduct a set of simulations with MSPastry. Our result indicates that although MSPastry already has various overlay maintenance techniques that deal with failures in overlays, they are not sufficiently perturbation-resistant.

Figure 5.2 summarizes our results. Each simulation consists of two stages. In the first stage, 1000 insertion requests are generated to the *static* overlay of MSPastry. These 1000 insertion requests have randomly-generated unique message IDs. In the second stage, 1000 lookup requests are generated by the same node which generates the insertion requests in the first stage. The lookup requests are generated every (online period + offline period) seconds one by one. These 1000 lookup requests are the lookup requests for 1000 IDs inserted in the first stage. The overlay in the second stage is not static; Each node gets perturbed with some probability. As mentioned earlier, our model of perturbation can be described as *flapping*. A perturbed node periodically flaps between

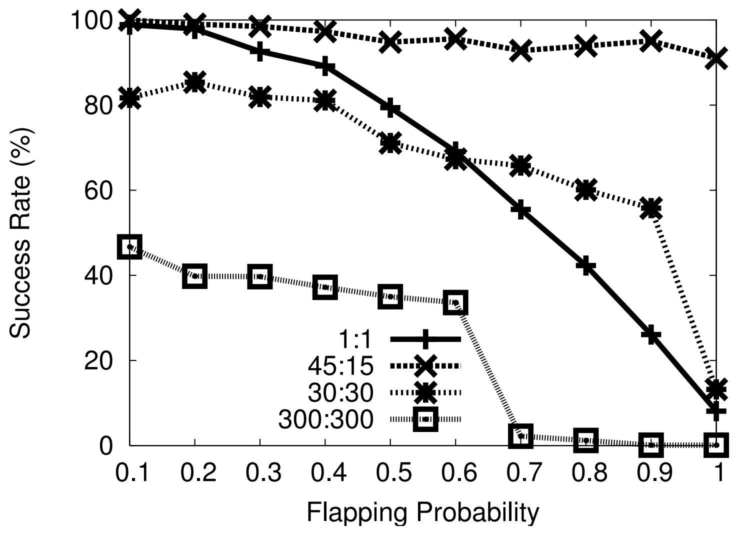

Figure 5.2: The effect of perturbation on MSPastry. x-axis (flapping probability) indicates the probability for a node to get perturbed. 1:1 indicates that the online period is 1 second and the offline period is 1 second. The same goes for 45:15 (idle:offline=45:15), 30:30 (idle:offline=30:30), and 300:300 (idle:offline=300:300).

being offline and being idle (online). At the beginning of each idle period, every node comes back online and stays online during the period. At the beginning of the offline period, however, each node decides whether to go offline or to stay online based on the flapping probability. Each node randomly picks its very first beginning of the flapping period (i.e. idle period + offline period). Lookups are performed after every node enters its flapping period.

As in Figure 5.2, when idle:offline period is 45:15 (seconds), MSPastry can route more than 90% of the messages successfully. This result shows that MSPastry is already robust to a certain level, which is due to the overlay maintenance techniques of MSPastry. However, the number of successful lookups decreases in other cases and with higher flap rates in general. When idle:offline period is 30:30 (seconds), the success rate is roughly about 85% even with the flapping probability of 0.1. When idle:offline period is 1:1 (seconds), the success rate drops almost linearly. With idle:offline period of 300:300 (seconds), the success rate is almost 0 with the flapping probability from 0.8 to 1. This result clearly shows that the overlay maintenance techniques of MSPastry are perturbation-resistant only to a limited degree. Further, both short-term perturbations (e.g., 1:1) and long-term perturbations (e..g, 300:300) drastically affect the lookup behavior.

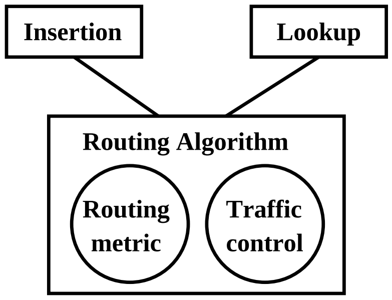

Figure 5.3: The architecture of MPIL. Insertion and lookup operations use the routing algorithm, and the routing algorithm uses a routing metric and a traffic control algorithm.

5.4 Multi-Path Insertion/Lookup Algorithm

Figure 5.3 shows the overall architecture of MPIL (Multi-Path Insertion/Lookup). The insertion and lookup operations use the routing algorithm, and the routing algorithm rely on two important bases, a routing metric and a traffic control algorithm. In fact, the insertion and lookup operations become straight-forward once we understand the routing algorithm.

The MPIL routing algorithm works as follows: when a node receives a lookup request for an object, it calculates the routing metric for each of its neighboring peers, and forwards the lookup to the "best" few peers. Below, we first describe how the routing metric is calculated for a given object, and then detail the routing algorithm itself.

5.4.1 Routing Metric

For a given object ID and a neighboring peer's ID, the routing metric is simply the number of matching digits appearing in same positions. Another way to view this metric is the number of 0's in XOR product of the two ID's; this is related to the concept of Hamming distance. Unlike the Kademlia overlay [71], which also uses an XOR, MPIL uses the XOR metric to select multiple next hops for the query – we detail this in Section 5.4.2.

Figure 5.4 shows an illustration of this routing metric. Consider the nodes with id's 1001 and 1011 from the 4-bit ID space (example on the left). The value of the MPIL routing metric is 3, since only the second-least significant bits do not match. Suppose a node currently holds a lookup request for an object with ID 1001, and the node has two neighbors 1011 and 0010. Since the values returned by MPIL are 3 and 1 respectively, the lookup is forwarded to node 1011.

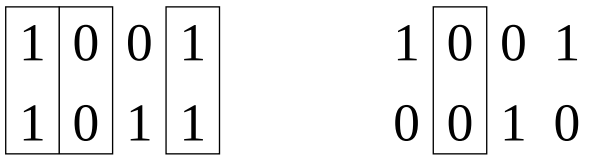

Figure 5.4: An example for the routing metric. On the Left, ID 1001 and 1011, on the right, ID 1001 and 0010. The routing metric gives the value of 3 and 1, respectively.

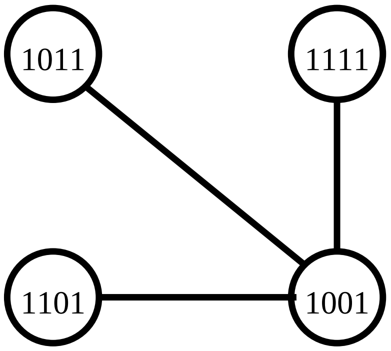

Figure 5.5: An example topology for continuous forwarding

5.4.2 Properties of MPIL's Routing Metric

The routing metric of MPIL has the following advantages over other routing metrics that exist for structured overlays.

Continuous Forwarding over Arbitrary Overlays MPIL is better than prefix or suffix routing at distinguishing neighbors of a node when trying to select the best next hop for a lookup message. For example, in the overlay of Figure 5.5, if node 1001 has a lookup message for object 0110, both prefix routing and suffix routing treat all neighbors as being equivalent - this may cause the lookup message to be dropped or evaluated again to break the tie. However, the MPIL routing metric returns 1111 as the best neighbor.

The reason can be explained probabilistically. In prefix routing, the probability that any given two IDs share no common prefix at all is 0.75 for base-4 representation, and 0.5 for binary representation. Considering that having a common prefix is a basic requirement, the probability needs to be far lower than 0.75 or 0.5 in order for the prefix routing to be used over arbitrary overlays. This problem becomes worse especially when the large fraction of the total nodes has only a small number of neighbors, e.g. power-law graphs.

On the other hand, for the MPIL routing metric, the probability of the above event is only $(\frac{3}{4})^{80} = (1.0113490...)^{-10}$ if we assume 160-bit ID space and base-4 representation. The MPIL routing metric thus distinguishes neighbors better; at the least, this ensures a lookup request undergoes *continuous forwarding* even over arbitrary overlays.

Redundancy For Robustness The MPIL routing metric provides an easy way to exploit redundancy for robustness since it provides an inherent way to create multiple paths to multiple peers. Since the MPIL routing metric counts the number of common digits in same positions, there can be multiple nodes that have the same number of common digits. In Figure 5.5, suppose the node 1001 forwards a message of ID 0001. 1111 and 0001 share 1 common digit, 1101 and 0001 share 2 common digits, and 1011 and 0001 share 2 common digits. Thus, 1101 and 1011 are both the candidates for the next hop. Unlike other routing algorithms that break this tie using other mechanisms, MPIL forwards messages to every candidate, thus creating multiple paths to multiple peers. We use the term *flows* or *paths*. If a node forwards a query to exactly one neighboring node, there is only one flow. For each additional neighbor that is chosen to forward the lookup, an additional flow is said to be created.

Such replication might cause some nodes to receive the same message. In this case, there are two options. A node can either silently discard the message and not forward it any more (thus stopping the flow), or forward the message again. We explore both options in our simulations.

The effectiveness of such redundancy is limited for prefix and suffix routing due to the lower distinguishability of their routing metrics.

5.4.3 MPIL Routing

MPIL routing works as follows; When a node receives a lookup message containing an object ID, the node calculates the value of each neighbor's routing metric w.r.t. this object, as described in Section 5.4.1. The node then forwards the message to the neighbor having the highest value. In the case that the node has several neighbors with the same highest value, the node has to choose multiple nodes from among all such highest-value neighbors.

To prevent message explosion, a message field called max_flows is used to limit the number of extra flows created. max_flows is an integer field in every message, and it is decreased each time a node creates an *additional* flow (recall that forwarding to exactly one node is not considered as an additional flow). When max_flows is decreased to 0, no additional flows can be created. This max_flows is conceptually similar to "quota" that is consumed by each node on a route whenever a node replicates a message. The original max_flows value of a message is specified by the originator of the message.

To summarize, when a node receives a message, it does the following:

1. Creates a list of possible candidates for forwarding the message.
2. Compares the size of the list and ($max\_flows + given\_flows$), where $given\_flows$ is 0 if the node is the original sender, and 1 otherwise. max_flows is specified in the message.
3. Picks the minimum value of the two (say m).

```
M = Message ID
N = Node ID

if M has been forwarded already, discard it (optional).
for node in (neighbor_list of N - M.route):
    C = common digits between M and node
    if C is the largest until now:
        next_hop_list = [node]
    elif C is equal to the largest until now:
        next_hop_list.add(node)
Count common digits between M and N
if N has the largest value among all nodes in neighbor_list:
    N is the destination
    Perform message-specific actions (for insertion messages)
else:
    Apply the paths-limiting algorithm to next_hop_list
    Forward to nodes in next_hop_list
```

Figure 5.6: A Pseudo-code of the MPIL Routing Algorithm

4. Forwards the message to m nodes from the candidate list.
5. Replace the value of max_flows of each message that the node forwards to $(max\_flows - m + given\_flows)/m$.

In the last step, the decrease of max_flows by $m - given\_flows$ is because that is the number of *additional* peers that the node forwards the message to. The node divides the value by m for distribution of the original max_flows. If the final value is not an integer, a node can distribute the residue one by one in round-robin fashion to the m nodes.

The complete MPIL algorithm uses the routing metric in Section 5.4.1 and the algorithm for limiting multiple flows. Figure 5.6 shows the pseudo-code of the algorithm. Note that when choosing *next_hop_list* from *neighbor_list*, the number of common digits between *N* and *M* does not have any effect. In addition, there is a message field called *route*, which contains the list of nodes that the message has visited. The *route* field prevents the message from being forwarded to a node the message has visited already. Thus, Choosing *next_hop_list* is dependent only on peers in *neighbor_list*, excluding the nodes in *M.route* and *N*. Depending on the configuration, each node might discard a message that has been forwarded already. In this case, a sequence number or a random number should be attached to distinguish the message from old messages with the same message ID.

5.4.4 Insertion, Lookup, and Deletion

Both insertion and lookup use the routing algorithm, but there are differences in the details.

Insertion An object (or a pointer to its location) can be inserted using MPIL routing. An insertion message is propagated and replicated as usual in the MPIL routing algorithm, and an object is inserted at a node when none of its neighbor nodes have a higher MPIL routing metric value than the node. We call such nodes as *local maxima*. This results in multiple replicas of the object being created.

Replica placement is done by specifying the number of per-flow replicas (we call this number *num_replicas* for the discussion. In Figure 5.6, it says that if *N* is the destination, then it performs message-specific actions. In the case of an insertion message, *N* stores the object location specified in the message. However, this process continues *num_replicas*-times to store more replicas. This is possible because each node picks the next hop from (*its neighbor_list - M.route*) as in Figure 5.6, a list that does not include the node itself.

Since *max_flows* is the maximum number of possible paths, and each path creates *num_replicas* replicas, the maximum total number of replicas created by an MPIL object insertion request is bounded from above by *max_flows* $\times$ *num_replicas*.

Querying A lookup message that is a query for an object is propagated in essentially the same manner as insertion requests above. However, each recipient node checks to see it has the object; if it does, it stops forwarding the query and replies back directly to the querying node. The forwarding process stops when either the location is found or the message has passed through *num_replicas* local maxima. For arbitrary overlays, although MPIL can never guarantee a 100% lookup success rate, our simulations reveal that the success rate of MPIL are close to 100%.

Deletion Deletion can be done in many ways, but here we discuss just one of them. Whenever a replica is placed in a node, the node sends a periodic heartbeat to the owner of the original object. When the originator wants to delete a replica, it sends an explicit delete message to the node.

5.4.5 Comprehensive Example

Figure 5.7 shows an example of how MPIL inserts and queries an object. Suppose the node 0001 wants to insert an object with ID 1011. Originally, *max_flows* is 2 and *num_replicas* is 2. After node 0001, *max_flows* becomes 1. The node 0001 first selects 1001 among its neighbors because 1011 and 1001 share three common digits, while 1011 and 0000 share only one digit. Since 1001 shares the largest common digits among all of its neighbors, 1001 stores this object and decrements *num_replicas* by 1. But it still forwards the message to 1110 since *num_replicas* is 1. Since 1110 has two neighbors that share three common digits and *max_flows* is still 1, 1110 forwards this message to both neighbors. 0011 and

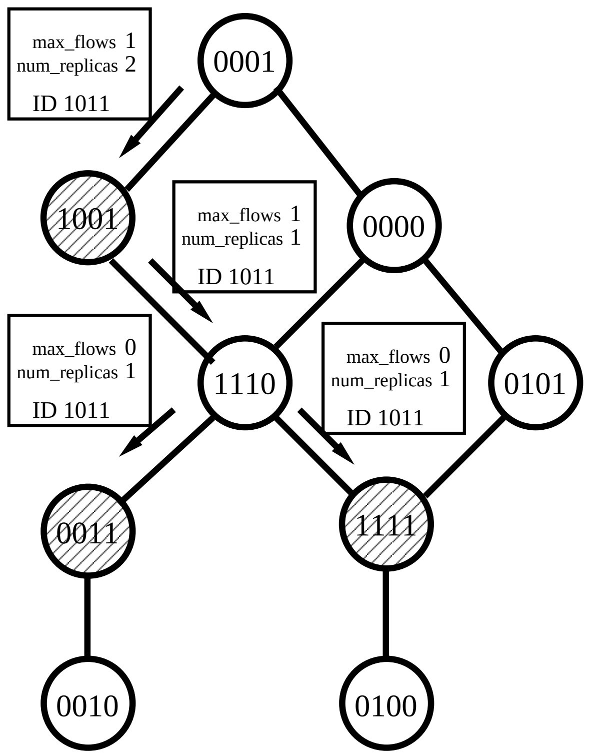

Figure 5.7: An example of MPIL. Gray nodes store the location of the object.

1111 receive this message and store the location because they share the largest common digits among all of their neighbors. They stop forwarding because *num_replicas* has reached 0. Lookup messages follow the exact same steps, but every node along the routes checks if it has the location. The notion of *flow* can be explained using Figure 5.7 again. There are two flows. One is from 0001 to 0011, and the other is from 0001 to 1111. We say that one additional flow is created by 1110.

5.5 Analysis

In this section, we present the analysis results of MPIL over various types of overlay topologies. First, we study the expected number of local maxima (See Section 5.4.4 for the definition of local maxima), which is an upper bound on the expected number of replicas, and expected number of hops to a local maximum from a node in general topologies. Recall that a local maxima node may store a replica (if the insert message reaches the node). We study two examples, random regular topologies and complete topologies.

There are several assumptions for the analysis. We assume that 1) there is an m-bit ID space with base-2^b representation, where $m = Mb$ for some constant M. Thus, each ID is a M-character-wide string with 2^b possible characters. 2) the total number of nodes is N, and the degrees is d. 3) There is a message with

ID a, and node IDs are $a_0, \ldots, a_N$. We say that a node ID, a_i, is *k-common* when a_i shares k common digits with the message ID, a.

5.5.1 General Overlay Topologies

For the expected number of local maxima, assume that the degree distribution function of a given type of overlays is known. Then, we can calculate the expected number of local maxima in an overlay topology by $N \times C$, where

$C = \sum_{d=1}^{N} \left\{ P(\# \text{ of neighbors} = d) \sum_{k=1}^{M} \left(A \times B^d \right) \right\}$

$A = \binom{M}{k} \left(\frac{1}{2^b} \right)^k \left(\frac{2^b-1}{2^b} \right)^{M-k}$

$B = \sum_{j=0}^{k-1} \binom{M}{j} \left(\frac{1}{2^b} \right)^j \left(\frac{2^b-1}{2^b} \right)^{M-j}$

C is the probability for a node to become a local maximum. A is the probability for a node to be *k-common*, and B is the probability for every other node to be *j-common*, for some $j < k$.

If we assume that the local maxima are distributed uniformly over the topology and we perform a random walk over the topology, then the expected number of hops to reach one of the local maxima from any node in the overlay is simply $\frac{1}{C}$.

Thus, if the degree distribution function is known, we can calculate the expected number of local maxima and the expected number of hops to one of the local maxima.

5.5.2 Random Regular and Complete Topologies

The degree distribution function of random regular overlay topologies, where each node has fixed d neighbors is given by,

$$P(\# \text{ of neighbors} = i) = \begin{cases} 1 & \text{if } i = d \\ 0 & \text{otherwise} \end{cases}$$

Then, we can calculate the average number of local maxima in a random regular topology, which is $N \times C$, where $C = \sum_{k=1}^{M} \left(A \times B^d \right)$. C is a constant for a given d. Figure 5.8 shows the average number of local maxima with different number of nodes and neighbors.

The expected number of hops to one of the local maxima by a random walk is also a constant, $\frac{1}{C}$

Similarly, we can calculate the expected number of replicas in a complete topology by using a similar equation, which is, $N \times \sum_{k=1}^{M} \left(A \times D^{N-1} \right)$, where

$D = \sum_{j=0}^{k} \binom{M}{j} \left(\frac{1}{2^b} \right)^j \left(\frac{2^b-1}{2^b} \right)^{M-j}$.

Compared to the equation for random topologies, this equation uses $N - 1$ instead of d, since it assumes a complete topology. Also, D is exactly the same as B except that the summation includes k, since it considers the number of replicas. Figure 5.9 shows the results for various number of nodes.

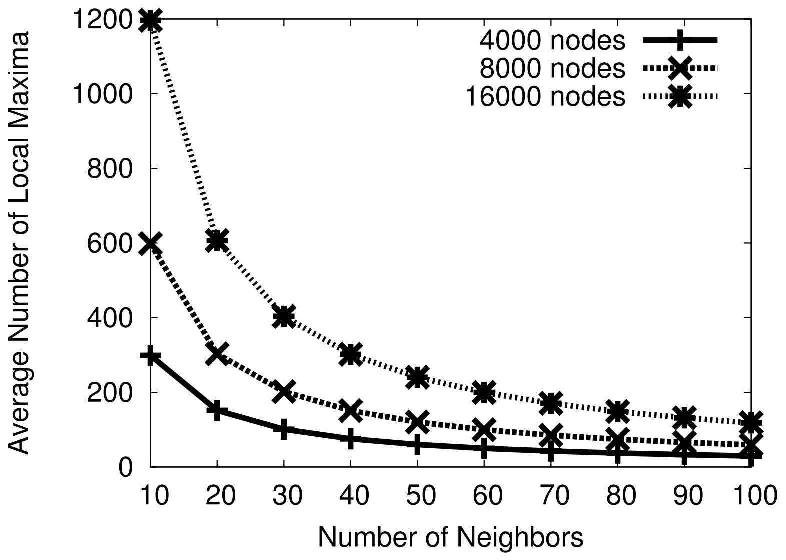

Figure 5.8: The expected number of local maxima for random regular topologies

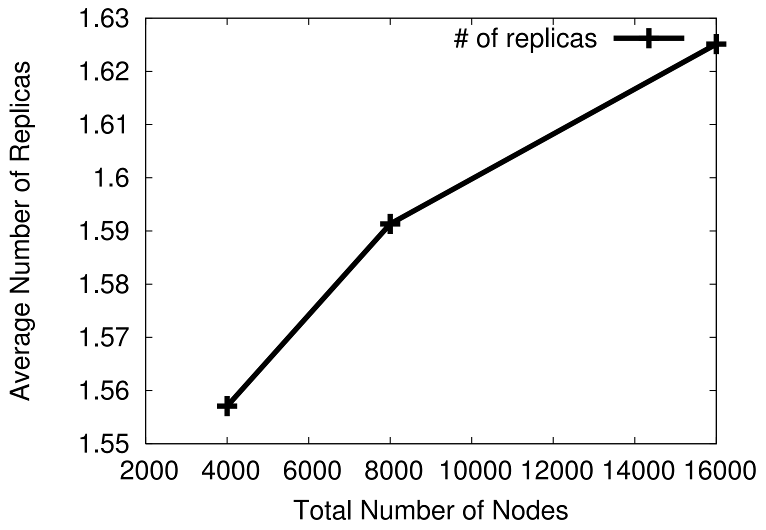

Figure 5.9: The expected number of replicas for complete topologies

5.6 Simulation Results

Two different classes of simulations are performed to examine MPIL and its robustness. The first class of simulations is mainly to evaluate the MPIL insertion/lookup performance over various static overlays. We wrote an application message simulator in Python for overlay-level routing. The second class of simulations evaluates the robustness of MPIL over structured overlays using MSPastry [14]. For ID-generation, we use random numbers picked from 160-bit ID space. All simulations are done on Pentium4 2.7GHz and 512M RAM.

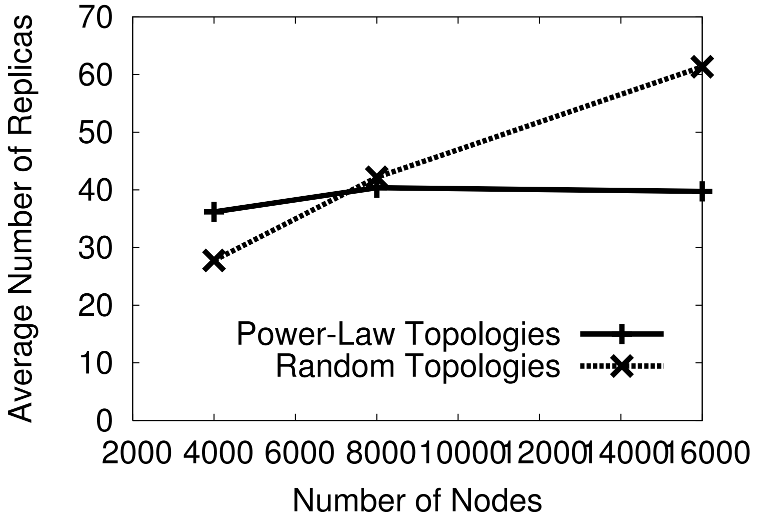

Figure 5.10: The behaviors of MPIL insertion - number of replicas

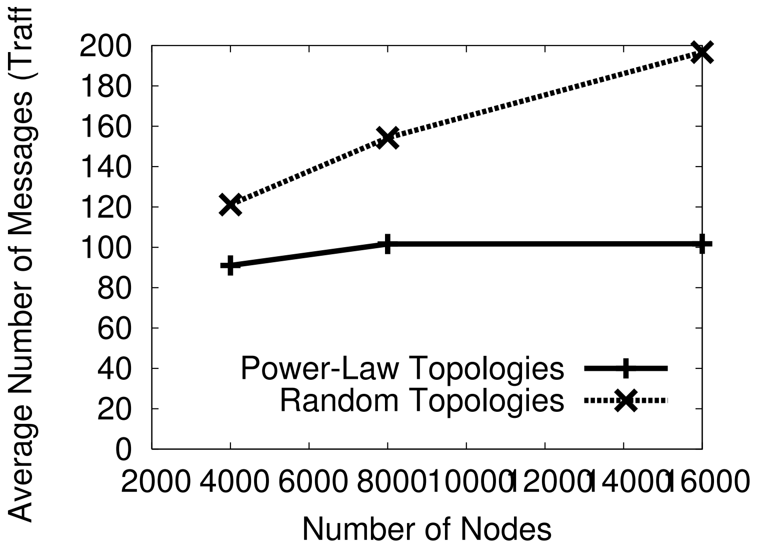

Figure 5.11: The behaviors of MPIL insertion - insertion traffic

5.6.1 MPIL Insertion/Lookup Performance Over Static Overlays

Overlay Topologies There are very few reliable benchmarks or workload generators for legacy distributed applications like Grid applications. However, we believe that power-law or random graph structures may be natural for legacy application overlays; we use these below.

10 different power-law graphs are generated by Inet [50], each with 4000 nodes, 8000 nodes, and 16000 nodes. We use 0% of degree 1 nodes. Similarly, 10 different random graphs are generated, each with 4000 nodes, 8000 nodes, and 16000 nodes. In these random graphs, each node has 100 neighbors, equally.

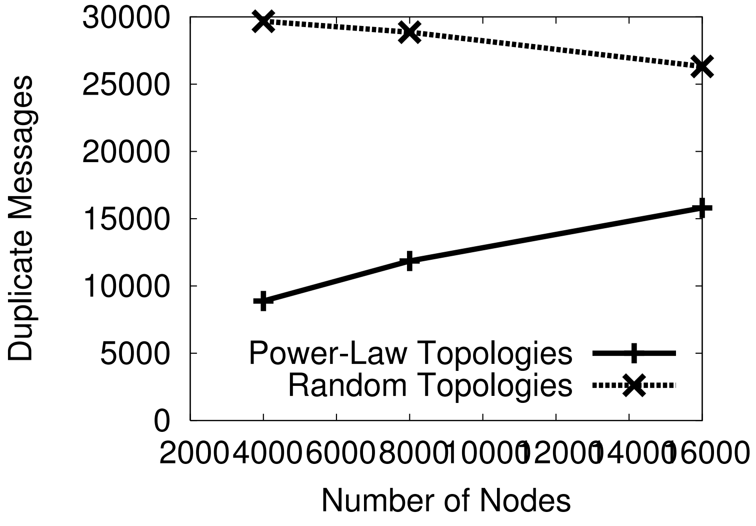

Figure 5.12: The behaviors of MPIL insertion - duplicate messages

Methodology For each overlay, random nodes are chosen to insert objects with different IDs 100 times. After that, those 100 objects are queried one by one again by randomly chosen nodes. Since there are 10 different overlays for each 4000 nodes, 8000 nodes, and 16000 nodes, the total number of insertion/lookup pairs is 1000 for every number of nodes. For all insertions and lookups, a node silently discards a message if the node receives the same message more than once.

Insertions Figures 5.10, 5.11 and 5.12 show the insertion performance of MPIL over the power-law and random overlays. The maximum number of flows is fixed at 30 and the per-flow replicas is fixed at 5 (See Section 5.4.2, 5.4.3, and 5.4.4 for the notion of flows, maximum flows specified by originators, and per-flow replicas). We measure two different categories - number of replicas and traffic per insertion - over 4000 nodes, 8000 nodes, and 16000 nodes.

Figure 5.10 shows the average number of replicas per insertion and Figure 5.11 shows the average number of total messages per insertion. Figure 5.12 shows the total number of duplicate insertion requests. Whenever a node receives the same insertion request from a different neighbor, it is considered as a duplicate request.

The number of replicas is bounded by (the maximum number of flows) $\times$ (the number of per-flow replicas). Thus, in Figure 5.10, the maximum number of replicas is bounded from above by 150, regardless of the number of nodes. As discussed earlier, additional flows are not created at every node, but at a node that has multiple neighbors with the same number of common digits. Thus, the actual number of flows is usually less than the maximum specified by the originator. Figure 5.10 shows this behavior as well. Even with 30 maximum flows and 5 per-flow replicas, the actual number of replicas is much less than

		Per-flow Replicas				
# nodes	Max_flows	1	2	3	4	5
4000	5	52.9	94.4	97.7	98.7	99.1
	10	55.4	98.7	99.7	99.9	100
	15	56.0	99.0	99.7	99.9	100
8000	5	57.1	96.5	98.8	99.6	99.2
	10	60.5	99.2	100	100	100
	15	60.0	99.6	100	100	100
16000	5	58.3	98.1	99.7	99.9	99.9
	10	60.4	99.5	100	100	100
	15	60.9	99.8	100	100	100

Table 5.1: MPIL lookup success rate over power-law topologies

		Per-flow Replicas				
# nodes	Max_flows	1	2	3	4	5
4000	5	98.6	100	100	100	100
	10	98.8	100	100	100	100
	15	98.4	100	100	100	100
8000	5	97.0	99.9	100	100	100
	10	98.5	100	100	100	100
	15	98.7	100	100	100	100
16000	5	95.0	99.9	100	100	100
	10	98.4	100	100	100	100
	15	98.6	100	100	100	100

Table 5.2: MPIL lookup success rate over random topologies

150.

A couple of further observations can be made from Figures 5.10, 5.11, and 5.12. First, the number of replicas and traffic of insertions in the power-law overlays stay almost the same across different settings. The reason can be found in Figure 5.12. As the number of nodes increases, the number of duplicate messages increases in the power-law overlays. This means that more duplicate messages arrive at the same set of nodes and are silently discarded, which prohibits an insertion from storing more replicas. Second, the number of replicas and traffic increases in the random overlays in contrast to power-law overlays. The reason can be found again from the duplicate messages. In Figure 5.12, the number of duplicate messages decreases as the number of nodes increases in the random overlays. Thus, more messages follow different paths as the number of nodes increases, which leads to storing more replicas.

The difference in number of duplicate messages between random overlays and power-law overlays is because each node has 100 neighbors in the former, while many nodes have only a few neighbors in the latter.

Lookups Table 5.1 and 5.2 show the success rates of MPIL lookups in various settings. Note that the per-flow replicas (r) for lookups means that the lookup stops when a flow encounters a node with the largest common digits r-times.

# of Node	Actual # of Flows
Power-Law 4000	8.782
Power-Law 8000	9.151
Power-Law 16000	9.542
Random 4000	9.323
Random 8000	9.505
Random 16000	9.63

Table 5.3: Actual number of flows of lookups

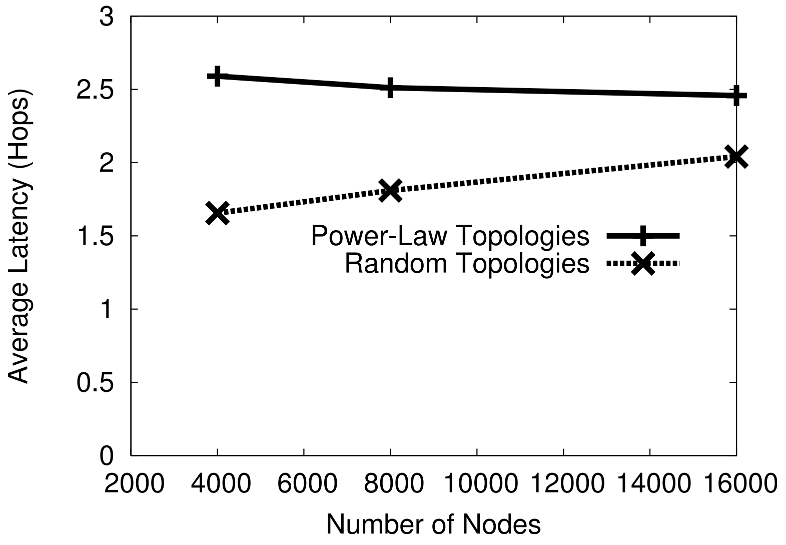

Figure 5.13: MPIL Lookup Latency (Hops)

Insertions are performed before lookups, and the number of maximum flows is fixed at 30 and the number of per-flow replicas is fixed at 5. Since we consider insertions to be rare events compared to the lookups, the traffic of insertions caused by the large number of maximum flows and per-flow replicas can be amortized over time.

From Table 5.1 and 5.2, three observations can be made. First, having more per-flow replicas gives higher success rates. This is obvious because the number of per-flow replicas for a lookup limits the path-length of the lookup. Second, having more flows gives higher success rates. This is also obvious because the number of maximum flows limits the number of search paths. Third, having larger numbers of nodes gives higher success rates with the same number of max_flows and per-flow replicas, although the difference is very small and may be negligible. The reason of the difference can be found in Table 5.3. Table 5.3 shows an example of the average number of flows that are *actually* created by lookups with 10 max_flows and 3 per-flow replicas over various numbers of nodes. As the number of nodes grows, the actual number of flows grows also, even though the maximum flows and per-flow replicas are the same. Since there are more flows for bigger overlays, the success rates increase, accordingly.

Figures 5.13 and 5.14 show the latency and required traffic of lookups. In

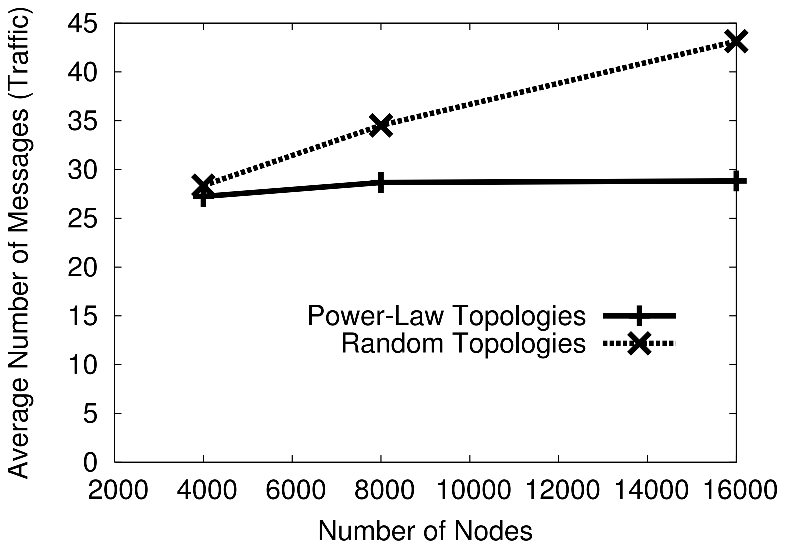

Figure 5.14: MPIL Lookup Traffic

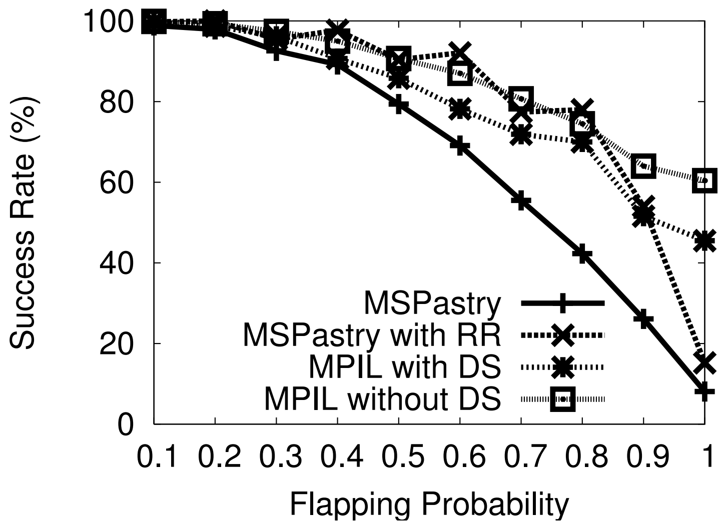

Figure 5.15: Success rate of MSPastry simulations - idle:offline=1:1

this simulation, max_flows is fixed at 10 and per-flow replicas is fixed at 5, since that setting gives 100% success rates for all 4000, 8000, and 16000 nodes in both the power-law overlays and random overlays. Also, Figure 5.13 only shows the number of hops of the first successful reply of a lookup among all successful replies. Multiple successful replies are possible because there are multiple replicas stored in the system. However, Figure 5.14 shows the total traffic per a lookup request, as well as the traffic for the first successful reply.

As in Figure 5.13, the latency stays almost same even though the number of nodes increases. Table 5.1 shows a similar result because more than 50% of lookups are satisfied even with 1 per-flow replicas and almost all lookups are satisfied with 2 per-flow replicas. Although limiting per-flow replicas does not

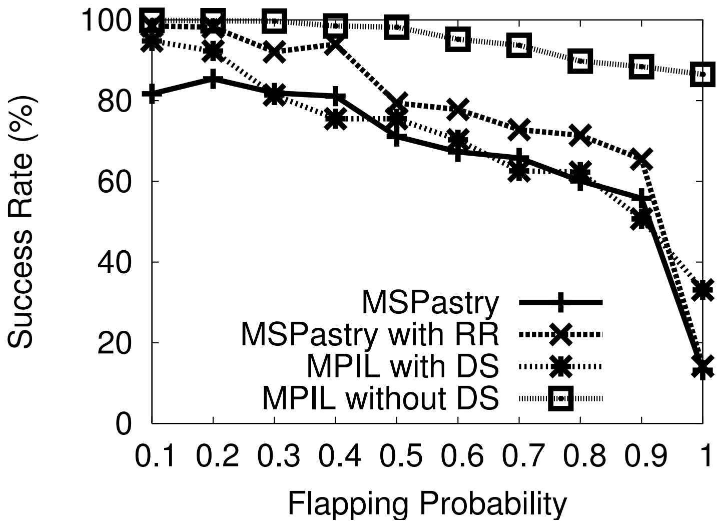

Figure 5.16: Success rate of MSPastry simulations - idle:offline=30:30

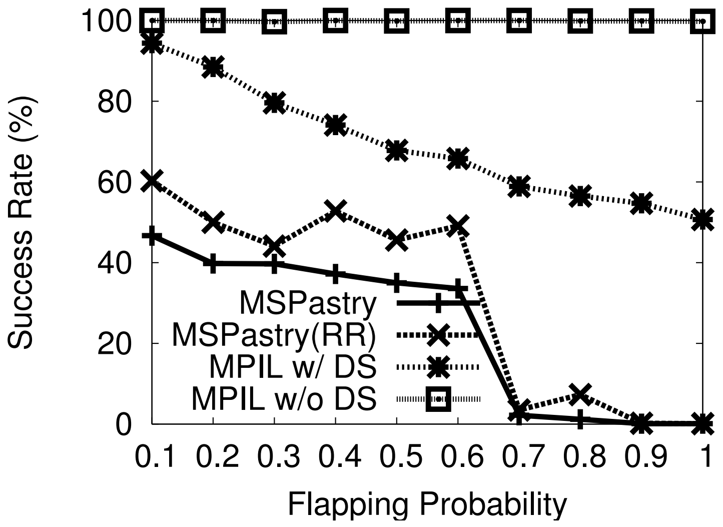

Figure 5.17: Success rate of MSPastry simulations - idle:offline=300:300

accurately limit the number of maximum hops a lookup request is propagated, it definitely has a correlation. Therefore, both Figure 5.13 and Table 5.1 tell us that MPIL lookups cause small and steady number of hops across different numbers of nodes in the power-law overlays.

Lookup traffic also stays almost same in Figure 5.14. The same reason from the case of insertions can be applied to here. Since the number of duplicate messages increases as the number of nodes increases, more flows and traffic are suppressed.

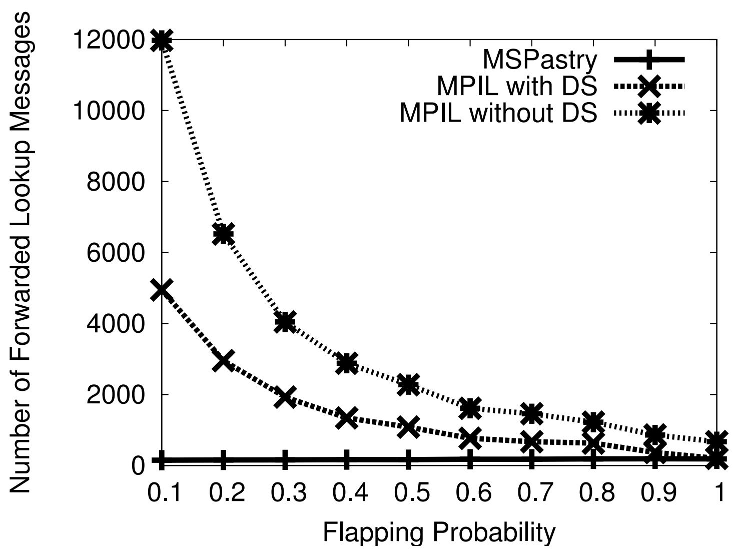

Figure 5.18: Overall traffic of MSPastry simulations - lookup traffic

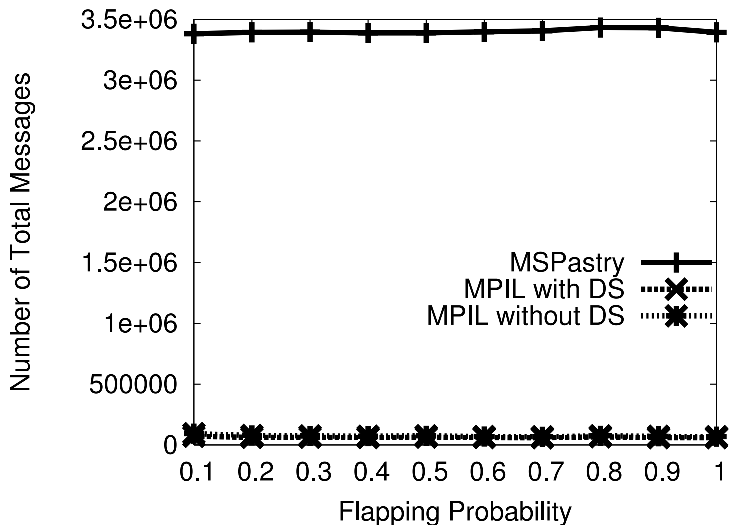

Figure 5.19: Overall traffic of MSPastry simulations - total traffic including maintenance messages

5.6.2 MPIL Over MSPastry

In this section, we study the robustness of MPIL under perturbation. We run MPIL over the overlay of MSPastry by implementing the MPIL algorithm in MSPastry. We compare the robustness of MPIL to that of MSPastry with its overlay maintenance techniques.

Methodology To evaluate the robustness of MPIL, we conduct a set of simulations using MSPastry. These simulations are done in the same condition as in Section 5.3. 1000 insertions are generated first, and 1000 lookups for the same

IDs of insertions are generated next. For MSPastry simulations, we use MSPastry with all the overlay maintenance techniques described in [14]. For MPIL simulations, we modify MSPastry; we replace the original routing algorithm of MSPastry with MPIL. However, we do not use any of the overlay maintenance techniques. In other words, we use the structured overlay of MSPastry, but none of the overlay maintenance techniques. A total of 1000 nodes are used in all simulations using a topology generated by GT-ITM [10] as an underlying (Internet) topology.

MSPastry Configuration The default parameters of MSPastry simulations across all simulations are; $b = 4$, $l = 8$, *leafset probing period* = 30 seconds, *routing table maintenance period* = 12000 seconds, *routing table probing period* = 90 seconds, *probe timeout* = 3, and *probe retries* = 2.

MPIL Configuration All MPIL simulations are done with 10 maximum flows and 3 per-flow replicas for both insertions and lookups. However, the number of replicas actually inserted by the insertions in the network is typically 6-7.

Success Rate Figures 5.15, 5.16, and 5.17 show the success rates of the original MSPastry and MPIL under various perturbation probabilities. In all three figures, “MSPastry” shows the simulation results with the original MSPastry with no modification. “MSPastry with RR” shows the results with MSPastry with *Replication on Route* (RR). Using RR, every node on the route of an insertion message stores a replica whether it’s the target node or not. In these simulations, the typical number of hops of an insertion message is 2-3 for MSPastry. Thus, 2-3 is the typical degree of replication for MSPastry with RR. “MPIL with DS” shows the simulation results of MPIL with Duplicate Suppression (DS), while “MPIL without DS” shows the results of MPIL without DS. If MPIL uses DS, each node silently discards any message that the node forwarded before. Otherwise, each node forwards a message repeatedly, even if it received the same message before.

Intuitively, DS is good for static overlays because it reduces traffic. However, each node is likely to have a different set of neighbors in dynamic overlays. Thus, if a node keeps forwarding a message in dynamic overlays, the message is likely to take a different route each time, and the chance of arriving at one of the replicas can be increased. Figures 5.15, 5.16, and 5.17 actually confirms this intuition. MPIL without DS always gives higher success rates than MPIL with the duplicate suppression. However, MPIL typically gives higher success rates over the original MSPastry, regardless of DS.

Traffic MPIL gives better success rates under perturbation as in Figures 5.15, 5.16, and 5.17. However, since MPIL uses multicasts, the traffic generated by

MPIL can be far more than that of MSPastry. Figure 5.18 compares the lookup traffic of the original MSPastry and MPIL, when idle:offline is 30:30. As shown, MPIL creates a lot more lookup traffic than the original MSPastry, especially under low perturbation probabilities.

However, the original MSPastry uses various overlay maintenance techniques that create consistent background traffic, while MPIL does not use any of the techniques. Figure 5.19 shows the total number of messages sent including all the maintenance and control messages. As shown, MPIL creates far less messages than the original MSPastry if all the messages are counted. However, if the lookup frequency were higher, MSPastry's overhead might be justified.

5.7 Related Work

Perturbation has been studied in other contexts besides overlay networks. Birman *et al.* [8] study the effect of perturbation in the context of multicast protocol. In their simulation, virtually synchronized multicast groups are used to study the effect of perturbation. They measure throughput of a live node in the presence of perturbation - some fraction of the multicast group members sleep for some fraction of each second. Their result shows that even with a single perturbed group member, the throughput drops significantly, decreasing rapidly as the number of perturbed nodes increases.

Many studies have shown that the arrival rate and the departure rate of nodes in peer-to-peer systems are very high, which proves the instability of peer-to-peer systems. For example, Bhagwan *et al.* [7] show churn data for the Overnet p2p system. Saroui *et al.* [95] study node availability of Napster and Gnutella. Their result can be summarized as the best 20% of Napster peers has an uptime of 83% and more, and the best 20% of Gnutella peers has an uptime of 45% or more.

Robustness issues of DHTs have been studied recently [89, 14, 61]. Li *et al.* [61] study the effect of churn to some popular DHTs including Chord, Tapestry, Kelips, and Kademlia. Castro *et al.* [14] study the performance and dependability of Pastry with MSPastry implementation. Rhea *et al.* [89] identify three factors that affect the behavior of overlays under churn - reactive vs. periodic recovery, message timeout calculation, and proximity neighbor selection - and discuss various techniques that can be used for the three factors.

Efficient search algorithms for unstructured overlays have been studied recently [73, 65]. Lv *et al.* [65] explore the use of random walks and replication to improve the inefficiency of unstructured p2p systems caused by flooding. More recently, it has come to our attention that Morselli *et al.* [73] propose a search algorithm that combines random walks and a DHT routing algorithm based on Chord routing algorithm to improve the search efficiency of unstructured p2p systems. This study shares some similarities with our study in that 1) it uses name-space virtualization for unstructured overlays, 2) it proposes a replication strategy, and 3) it separately considers overlays and routing algorithms. However, their focus is producing unstructured p2p systems, while our focus is producing robust p2p systems.

Castro *et al.* [15] use flooding and random walks over Pastry's structured overlay to support complex queries. Our approach is different, because we develop a robust resource discovery algorithm that runs over any type of overlay.

5.8 Summary

We have presented the motivation, design, analysis, and evaluation of MPIL (Multi-Path Insertion and Lookup). The central challenges that MPIL addresses are the scale of peers and a type of dynamism called perturbation. Our evaluation has shown that MPIL performs well over various types of overlay structures and perturbations, while providing better lookup success rates than MSPastry under both short-term and long-term perturbation. MPIL provides a lookup success rate roughly 6 times better than that of MSPastry even in the worst case of idle:offline = 1:1.

Chapter 6

Conclusions and Future Directions

This dissertation has discussed why and how on-demand operations are effective in overcoming the challenges of scale and dynamism in a variety of distributed systems. We have presented four on-demand operations that address the challenges of scale and dynamism.

First, we have presented the design and evaluation of Moara, a group-based aggregation system. Moara achieves scalability with increasing numbers of machines, injected queries, and groups, by: (1) intelligently resolving composite query expressions, (2) constructing single-attribute aggregation trees that perform in-network aggregation, and (3) dynamically maintaining group trees based on query rates and group churn rates, thus reducing bandwidth consumption. Our experimental evaluations using simulations and deployments atop Emulab and PlanetLab demonstrate the effectiveness of Moara in answering queries accurately within hundreds of milliseconds across hundreds of nodes, and with low per-node bandwidth consumption.

Second, we have argued for the need, requirements, design, and evaluation of ISS (Intermediate Storage System) that treats intermediate storage as a first-class citizen for dataflow programs. Our experimental study of existing and candidate solutions shows the absence of a satisfactory solution. Our experimental results show that we can almost eliminate the interference with a combination of three techniques, i.e., an asynchronous rack-level selective replication mechanism. Our failure injection experiments show approximately 59% improvement in completion time using ISS.

Third, we have argued that worker-centric scheduling is more desirable than task-centric scheduling to exploit locality of interest present in data-intensive applications. We base our argument on two problems of task-centric scheduling, namely, unbalanced task assignments and premature scheduling decisions. We proposed various metrics, both deterministic and randomized, that can be used with worker-centric scheduling and found that metrics considering the number of file transfers generally give better performance over metrics considering the overlap between a task and a storage. We also found that worker-centric scheduling algorithms achieve better or comparable performance to task-centric scheduling, with the randomized approaches performing best.

Fourth, we have presented a new approach to resource location and discovery that is both perturbation-resistant and overlay-independent. Our algorithm, called MPIL, is independent of the underlying overlay structure. MPIL works well over both unstructured overlays that are random or power-law, and over structured overlays – like MSPastry. Under both short-term and long-term perturbation (which may arise from multiple concurrent client applications or churn respectively), MPIL has a better success rate than MSPastry routing, and at the cost of only slightly increased communication for each object lookup request. MPIL successfully provides any distributed application the ability to insert and query objects reliably and robustly over any arbitrary overlay, without the need to change the existing overlay maintenance mechanisms.

Future Directions There are many interesting directions that arise from this dissertation. First, the workload characteristics study of large-scale monitoring is a largely unexplored area. Workload characteristics, e.g., types of queries that users and administrators are interested in, the frequency of queries, etc., can be used to optimize the performance of a monitoring system. In fact, the optimization techniques of Moara were inspired by the observation that group-based monitoring is a common case for many users and administrators. However, this might be just the tip of the iceberg. There are likely many more characteristics and common cases where one can optimize the performance. To this end, it is necessary to know the workload characteristics of large-scale monitoring.

Second, optimization techniques for a general distributed querying system is another area that is largely unexplored. The techniques of Moara are specific to aggregation queries. Other types of queries, e.g., join, are known to have very different characteristics. Although a general distributed querying system that supports a wide range of queries has been proposed in the literature [43], optimization techniques for different types of queries beyond aggregation have not been studied well.

Third, there is a possibility that asynchronous rack-level replication can benefit more cloud applications than just dataflow programming frameworks. The experimental results of ISS show that asynchronous rack-level replication is indeed non-interfering and has low-overhead. Based on this observation, one can imagine a more universal use of asynchronous rack-level replication as a means to achieve better data availability in general. More broadly, it might be possible to design a family of locality-aware replication techniques.

Fourth, a fine-grained monitoring capability integrated with ISS can further improve ISS in terms of replication interference minimization. For example, if network and disk activities are monitored closely, ISS can make a fine-grained decision on when, how fast, and where to replicate intermediate data without interfering with foreground network and disk activities. This fine-grained decision can be based on control theory, where the master node of ISS receives feedback from monitoring components regarding the current job progress, network and disk activities, and failed components. The master can then decide the rate of replication periodically based on feedback control techniques.

Fifth, ISS can utilize a background transport protocol such as TCP-Nice and TCP-LP to further reduce network interference in addition to its asynchronous rack-level selective replication mechanism. However, it is not clear whether this will indeed benefit ISS or not, since a background transport protocol might only be able to achieve a low replication rate. This is due to the fact that a background transport protocol aggressively reduces its window size in anticipation of future congestion in the network. Thus, one might find that intermediate data is often not completely replicated, which leads to a longer window of vulnerability. Experimental validation is necessary to verify this hypothesis.

Finally, it is necessary to study computational limitations of MapReduce and how to improve it. Although it is well-known that MapReduce is not Turing complete, the exact scope of computational expressibility of MapReduce is not well-studied. This is an important concern as more and more researchers and scientists from various fields are interested in using MapReduce for their jobs. MapReduce has demonstrated its efficiency and usefulness in certain domains such as ad-hoc data analysis. There will be tremendous value if one can extend this benefit to more domains.

References

[1] William E. Allcock, Joseph Bester, John Bresnahan, Ann L. Chervenak, Ian T. Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. *The Computing Research Repository (CoRR)*, cs.DC/0103022, 2001.

[2] Amazon Web Services. `http://www.amazon.com/gp/browse.html?node=3435361`.

[3] Microsoft Azure Services Platform. `http://www.microsoft.com/azure`.

[4] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. *SIGOPS Operating Systems Review (OSR)*, 25(5), 1991.

[5] Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[6] Mayank Bawa, Aristides Gionis, Hector Garcia-Molina, and Rajeev Motwani. The Price of Validity in Dynamic Networks. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2004.

[7] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding Availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[8] Ken Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 2002.

[9] Eric Brewer. Towards Robust Distributed Systems (Invited Talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[10] Kenneth Calvert, Matthew B. Doar, Ascom Nexion, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 1997.

[11] Roy Campbell, Indranil Gupta, Michael Heath, Steven Y. Ko, Michael Kozuch, Marcel Kunze, Thomas Kwan, Kevin Lai, Hing Yan Lee, Martha Lyons, Dejan Milojicic, David O'Hallaron, and Yeng Chai Soh. Open Cirrus^{TM} Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[12] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the International Conference for High Performance Computing and Communications (SC)*, 2000.

[13] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceeding sof the 9th Heterogeneous Computing Workshop*, 2000.

[14] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and Dependability of Structured Peer-to-Peer Overlays. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2004.

[15] Miguel Castro, Manuel Costa, and Antony Rowstron. Should We Build Gnutella on a Structured Overlay? In *Proceedings of the Third Workshop on Hot Topics in Networks (HotNets)*, 2004.

[16] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth Multicast in a Cooperative Environment. In *Proceedings of the Nineteenth ACM Symposium on Operating systems principles (SOSP)*, 2003.

[17] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A Large-scale and Decentralised Application-level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[18] Walfredo Cirne, Francisco Brasileiro, Jacques Sauve, Nazareno Andrade, Daniel Paranhos, Elizeu Santos-Neto, and Raissa Medeiros. Grid Computing for Bag of Tasks Applications. In *Proceedings of the Third IFIP Conference on E-Commerce, E-Business, E-Government (I3E)*, 2003.

[19] Cisco. Cisco Data Center Infrastructure 2.5 Design Guide. `http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Inf%ra2_5/DCI_SRND.pdf`.

[20] CRA. Grand Research Challenges in Distributed Systems. `http://www.cra.org/reports/gc.systems.pdf`.

[21] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of the 9th International Euro-Par Conference*, 2003.

[22] Jeffrey Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. In *Keynote I: PACT*, 2006.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[24] Matthew B. Doar. A Better Model for Generating Test Networks. In *Proceedings of the Global Telecommunications Conference (Globecom)*, 1996.

[25] Dryad Project Page at MSR. `http://research.microsoft.com/en-us/projects/Dryad/`.

[26] Emulab. `http://www.emulab.net`.

[27] Franck Cappello et al. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID)*, 2005.

[28] Ian T. Foster et al. The Grid2003 Production Grid: Principles and Practice. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2004.

[29] Facebook. Hive. `http://hadoop.apache.org/hive/`.

[30] Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. Processing Set Expressions over Continuous Update Streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2003.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating systems principles (SOSP)*, 2003.

[32] The Globus Alliance. `http://www.globus.org/`.

[33] The Gnutella protocol specification v 0.4, Document revision 1.2. www.clip2.com, 2003.

[34] Google App Engine. `http://code.google.com/appengine/`.

[35] Google and IBM Join in "Cloud Computing" Research. `http://www.nytimes.com/2007/10/08/technology/08cloud.html`.

[36] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[37] Indranil Gupta, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Efficient Epidemic-Style Protocols for Reliable and Scalable Multicast. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[38] Indranil Gupta, Robbert van Renesse, and Ken Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2001.

[39] HDFS (Hadoop Distributed File System). `http://hadoop.apache.org/core/docs/r0.20.0/hdfs_user_guide.html`.

[40] Keren Horowitz and Dahlia Malkhi. Estimating Network Size from Local Information. *ACM Information Processing Letters*, 88(5):237–243, 2003.

[41] John H. Howard. An Overview of the Andrew File System. In *Proceedings of the Annual USENIX Winter Technical Conference*, 1988.

[42] HP Labs Dynamic Cloud Services. `http://www.hpl.hp.com/research/cloud.html`.

[43] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2005.

[44] Ryan Huebsch, Minos Garofalakis, Joseph M. Hellerstein, and Ion Stoica. Sharing Aggregate Computation for Distributed Queries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2007.

[45] Adriana Iamnitchi, Shyamala Doraimani, and Gabriele Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *Proceedings of the Fifteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2006.

[46] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proceedings of the 2007 EuroSys Conference (EuroSys)*, 2007.

[47] Navendu Jain, Michael Dahlin, Yin Zhang, Dmitry Kit, Prince Mahajan, and Praveen Yalagandula. STAR: Self Tuning Aggregation for Scalable Monitoring. In *Proceedings of the 33rd International Conference on Very Large Databases (VLDB)*, 2007.

[48] Navendu Jain, Dmitry Kit, Prince Mahajanand, Praveen Yalagandula, Michael Dahlin, and Yin Zhang. PRISM: Precision-Integrated Scalable Monitoring (extended). In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[49] Mark Jelasity and Alberto Montresor. Epidemic-Style Proactive Aggregation in Large Overlay Networks. In *Proceedings of the 24th International Conference of Distributed Computing Systems (ICDCS)*, 2004.

[50] Cheng Jin, Qian Chen, and Sugih Jamin. Inet: Internet Topology Generator. `http://topology.eecs.umich.edu/inet`, 2002.

[51] David Kempe, Alin Dobra, and Johannes Gehrke. Computing Aggregate Information Using Gossip. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003.

[52] Anne-Marie Kermarrec, Laurent Massouli, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14:248–258, 2001.

[53] Steven Y. Ko and Indranil Gupta. Perturbation-Resistant and Overlay-Independent Resource Discovery. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[54] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On Availability of Intermediate Data in Cloud Computations. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[55] Steven Y. Ko, Ramses Morales, and Indranil Gupta. New Worker-Centric Scheduling Strategies for Data-Intensive Grid Applications. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2007.

[56] Steven Y. Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. Moara: Flexible and Scalable Group-Based Querying System. In *Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2008.

[57] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Ken Birman, and Al Demers. Decentralized Schemes for Size Estimation in Large and Dynamic Groups. In *Proceedings of The IEEE International Symposium on Network Computing and Applications (NCA)*, 2005.

[58] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-Fly Sharing for Streamed Aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2006.

[59] Aleksandar Kuzmanovic and Edward W. Knightly. TCP-LP: Low-Priority Service via End-Point Congestion Control. *IEEE/ACM Transactions on Networking*, 14(4):739–752, 2006.

[60] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid*, 2003.

[61] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the Performance of Distributed Hash Tables under Churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[62] Jin Liang, Steve Y. Ko, Indranil Gupta, and Klara Nahrstedt. MON: On-demand Overlays for Distributed System Management. In *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS)*, 2005.

[63] Linux Monitoring Tools. `http://www.linuxlinks.com/Software/Monitoring/Network/`.

[64] Michael Litzkow, Miron Livny, and Matt Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, 1988.

[65] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the International Conference on Supercomputing (SC)*, 2002.

[66] Sam Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acqusitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, March 2005.

[67] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable Dynamic Emulation of Butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[68] Sorting 1PB with MapReduce. `http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html`.

[69] Keith Marzullo, Robert Cooper, Mark D. Wood, and Kenneth P. Birman. Tools for Distributed Application Management. *IEEE Computer*, 24(8):42–51, 1991.

[70] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation and Experience. *Parallel Computing*, 30(7), July 2004.

[71] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[72] Luiz Meyer, James Annis, Marta Mattoso, Mike Wilde, and Ian Foster. Planning Spatial Workflows to Optimize Grid Performance. *Technical Report, GriPhyN 2005-10*, 2005.

[73] Ruggero Morselli, Bobby Bhattacharjee, Michael A. Marsh, and Aravind Srinivasan. Efficient Lookup on Unstructured Topologies. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[74] Dushyanth Narayanan, Austin Donnelly, Richard Mortier, and Antony Rowstron. Delay Aware Querying with Seaweed. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, 2006.

[75] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[76] NSF. The NSF GENI Initiative. `http://www.nsf.gov/cise/geni/`.

[77] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2008.

[78] Owen O'Malley. Introduction to Hadoop. `http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/atta%chments/YahooHadoopIntro-apachecon-us-2008.pdf`.

[79] Venkata N. Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. NetProfiler: Profiling Wide-Area Networks Using Peer Cooperation. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[80] KyoungSoo Park and Vivek S. Pai. CoMon: a Mostly-scalable Monitoring System for PlanetLab. *SIGOPS Operating Systems Review (OSR)*, 40(1):65–74, 2006.

[81] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First Workshop on Hot Topics in Networks (HotNets)*, 2002.

[82] Michael Pinedo. *Scheduling: Theory, Algorithms and Systems.* Prentice Hall, New Jersey, USA, second edition, August 2001.

[83] PlanetLab. `http://www.planet-lab.org/`.

[84] PlanetLab Contributed Software. https://wiki.planet-lab.org/twiki/bin/view/Planetlab/ContributedSoftwar%e.

[85] Powered by Hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[86] Kavitha Ranganathan and Ian T. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proceedings of the Eleventh IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2002.

[87] Japan AIST Press Release. Providing Computing Power on demand Using the Grid technologies. http://www.aist.go.jp/aist_e/latest_research/2005/20051202/20051202.htm%l.

[88] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164 – 206, May 2003.

[89] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2004.

[90] Right Scale. http://www.rightscale.com.

[91] Arnold L. Rosenberg. On Scheduling Mesh-Structured Computations for Internet-Based Computing. *IEEE Transactions on Computers*, 53(9), September 2004.

[92] Arnold L. Rosenberg and Matthew Yurkewych. Guidelines for Scheduling Some Common Computation-Dags for Internet-Based Computing. *IEEE Transactions on Computers (TC)*, Vol. 54, No. 4, April 2005.

[93] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[94] Elizeu Santos-Neto, Walfredo Cirne, Francisco Vilar Brasileiro, and Aliandro Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. In *Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2004.

[95] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, 2002.

[96] Vijay Sekhri. Lessons Learned on Summer 04 Grid SDSS Coadd. https://www.darkenergysurvey.org/the-project/simulations/sdss-grid-coad%d/summer-04-grid-coadd, 2004.

[97] Simple Network Management Protocol. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/snmp.htm.

[98] Alfred Spector. Plenary Talk. ACM SOSP, 2003.

[99] Stanford SLAC Project (Network Monitoring tools). http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[100] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2001.

[101] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *SIGOPS Operating Systems Review*, 15(3):51–64, July 1981.

[102] Yufei Tao, George Kollios, Jeffrey Considine, Feifei Li, and Dimitris Papadias. Spatio-Temporal Aggegration Using Sketches. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, 2004.

[103] Igor Tatarinov and Alon Y. Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2004.

[104] Top 500 list. `http://www.top500.org`, 2009.

[105] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[106] Sivakumar Viswanathan, Bharadwaj Veeravalli, Dantong Yu, and Thomas G. Robertazzi. Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID)*, 2004.

[107] Werner Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[108] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., June 2009.

[109] Praveen Yalagandula and Mike Dahlin. A Scalable Distributed Information Management System. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2004.

[110] Yong Yao and Johannes Gehrke. Query Processing for Sensor Networks. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[111] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1), January 2004.

Author's Biography

Steven Y. Ko was born in Lincoln, Nebraska, but spent most of his life in Seoul, Korea. His research interest lies in the area of large-scale distributed systems, with an emphasis on Cloud Computing and data centers. His thesis was advised by Prof. Indranil Gupta, and his work benefits from ongoing collaborations with HP Labs. He has published a number of papers in various conferences and journals such as Middleware, ACM TAAS, IEEE DSN, Usenix HotOS, and IEEE SRDS. During his Ph.D career, he also worked at HP Labs and Lawrence Livermore National Lab. He received his B.S. from Yonsei University and M.S. from Seoul National University both in Korea. He has a lovely wife, Kyungmin Kwon, and a cute little daughter, Natalie Ko.