DATA LOCALITY TECHNIQUES IN AN ACTIVE CLUSTER FILE SYSTEM DESIGNED FOR SCIENTIFIC WORKFLOWS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Patrick Joseph Donnelly

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2016

© Copyright by

Patrick Joseph Donnelly

2016

All Rights Reserved

DATA LOCALITY TECHNIQUES IN AN ACTIVE CLUSTER FILE SYSTEM DESIGNED FOR SCIENTIFIC WORKFLOWS

Abstract

by

Patrick Joseph Donnelly

The continued exponential growth of storage capacity has catalyzed the broad acquisition of scientific data which must be processed. While today's large data analysis systems are highly effective at establishing data locality and eliminating inter-dependencies, they are not so easily incorporated into scientific workflows that are often complex and irregular graphs of sequential programs with multiple dependencies. To address the needs of scientific computing, I propose the design of an active storage cluster file system which allows for execution of regular unmodified applications with full data locality.

This dissertation analyzes the potential benefits of exploiting the structural information already available in scientific workflows – the explicit dependencies – to achieve a scalable and stable system. I begin with an outline of the design of the Confuga active storage cluster file system and its applicability to scientific computing. The remainder of the dissertation examines the techniques used to achieve a scalable and stable system. First, file system access by jobs is scoped to explicitly defined dependencies resolved at job dispatch. Second, workflow's structural information is harnessed to direct and control necessary file transfers to enforce cluster stability and maintain performance. Third, control of transfers is selectively relaxed to improve performance by limiting any negative effects of centralized transfer management.

This work benefits users by providing a complete batch execution platform joined with a cluster file system. The user does not need to redesign their workflow or provide additional consideration to the management of data dependencies. System stability and performance is managed by the cluster file system while providing jobs with complete data locality.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

This work would not have been possible without the kind direction of my advisor, Dr. Douglas Thain. His patient advice, thoughtful discussions, and helpful criticisms always had a way of making a difficult task seem manageable.

The Cooperative Computing Lab has been a home for several years now. Within it I found a fertile environment for challenging and rewarding projects. My fellow group members made this positive environment possible and have my gratitude. In particular, I wish to thank Hoang Bui, Peter Bui, Michael Albrecht, Li Yu, Dinesh Rajan, and Haiyan Meng for being enduring friends in addition to colleagues.

I am thankful to my fellow lab member Haiyan Meng for being a true friend when others had gone, for showing me the value of being in the office during normal working hours, and for teaching me how to manage time.

Outside the CCL, my friends Ryan Connaughton, Aaron Dingler, Steve Kurtz, and Nate Garrison were a persistent drag on productivity but their efforts kept me sane. Steve will be sad to hear the LEGO Super Star Destroyer is leaving orbit with no successor in sight.

I am grateful for the financial support provided by National Science Foundation grant OCI-1148330 and the Arthur J. Schmitt Presidential Fellowship in Science and Engineering. Without these programs, I would not have been able to focus on my research.

Finally, for teaching me to always challenge myself and encouraging me to pursue a doctorate, I thank my loving parents Dr. Joseph Donnelly and Elizabeth Donnelly.

CHAPTER 1

INTRODUCTION

The continued exponential growth of storage capacity [36] has catalyzed the broad acquisition of scientific data. Sensors, simulations, web traffic, and other sources of raw data are now stored and must be processed to draw new conclusions that were not before possible. Established approaches to scalable computing using combinations of resources from clusters, the Grid [30], and clouds [5] have been forced to respond to this *data deluge* [7]. This has introduced new challenges in distributed system design where the management of data is now at the forefront of concerns.

Early efforts by computer scientists sought to organize, manage, and transfer this data between geographically-distinct sites operated by different universities or institutions [10, 38, 46, 72]. The computation and storage of these resources were joined together in several efforts [16, 32, 49, 63] which formed the basis for the Grid [30]. Within this environment, scientists were taught to build workflows and applications that operated using explicit data dependencies and data transfers between sites performing computation [10, 46, 59].

However, the movement of data quickly became too expensive not only between sites but also within a local compute cluster of machines. This problem necessitated the creation of new software systems that solved data management within a local cluster by adopting a relatively new idea: *active storage*. Originally active storage was proposed as smart disks [66] whereby small computational tasks were relocated to the storage device for faster I/O and to harness unused processing capabilities. The concept of moving computation to data was an attractive idea for clusters where

moving data is expensive and spare computational resources are plentiful on machines storing data. The web industry began the effort of embracing active storage in clusters through the development of MapReduce [20].

MapReduce and other "Big Data" analysis systems of today [50, 95] require users to adopt structural constraints for their workflow which allows for data-locality, data-parallelism, and system stability. These constraints take the form of specific workflow templates [51, 53, 93] which permit the user to perform simple transformations on a monolithic dataset. This approach is highly effective when the objective is to compute relatively simple functions on colossal amounts of data. The small expense of writing or porting a small, widely known algorithm (such as k-means clustering) to these new platforms is well worth the payoff of running at colossal scale.

Unfortunately, these limited programming models are not so easily incorporated into scientific workflows developed for the Grid with data requirements that cannot be efficiently processed by these abstractions. For example, the CloudBLAST [51] bioinformatics framework is notable for using MapReduce to manage scheduling of sequential executions of BLAST by splitting an input sequence *query* against a (possibly large) genome sequence *database*. While this approach can be made to work, the database must still be distributed across all *map* tasks. This is known to not scale for larger databases [37] as MapReduce is not designed to assist with distributing common large data dependencies for parallel execution. Furthermore, the underlying distributed file system developed to support the programming model (e.g. HDFS [76]) is designed to only support parallel computation on chunks of a file. In summary, neither the abstraction nor the file system offers facilities for whole-file data parallelism required by scientific workflows.

There have been efforts to make suitable data-locality-aware abstractions [53, 93] available to scientific workflows [15] but adoption is limited as abstractions still impose structural constraints on the workflow. Instead, the scientific community

has adopted a flexible workflow model composed of standard sequential applications chained together by data dependencies and represented as a directed acyclic graph (DAG) of jobs. This work is based on the observation that abstractions are tools to express data dependencies to the compute engine but **scientific workflows already express sufficient structural information for scalable dependency management without the use of abstractions.** Because each job includes its list of dependencies, traditional workflow management systems like DAGMan [17], Makeflow [3], or Swift [52] are able to order and parallelize the execution of jobs and to transport dependencies with jobs. Unfortunately, the underlying compute platform does not use this information to control data access for scalability (e.g. Condor [49]) nor does it communicate dependencies to any coupled distributed storage system (e.g. SGE [32] on Panasas [54] or Condor with Hadoop [22, 92]).

This dissertation analyzes the potential benefits of exploiting the structural information already available in scientific workflows – the explicit dependencies – to achieve a scalable and stable system. To accomplish this, I have developed an active storage cluster file system named **Confuga** [23] which harnesses the workflow file dependency information to allow for the *efficient and controlled distribution of files* across active storage nodes and *scalable file system metadata access* by workflow jobs. Confuga combines the *workflow model* of scientific computing with the *storage architecture* of distributed cluster file systems. End users place their datasets in Confuga using standard file manipulation tools and then direct their workflow manager to submit jobs to Confuga. In this way, Confuga acts as a replacement for existing batch execution systems. The user does not need to redesign their workflow or provide additional consideration to the management of data dependencies used in their workflow.

Confuga is built around the idea of leveraging the **job namespace** to achieve a stable system. While typical distributed file systems [54, 73, 74, 88] must be designed

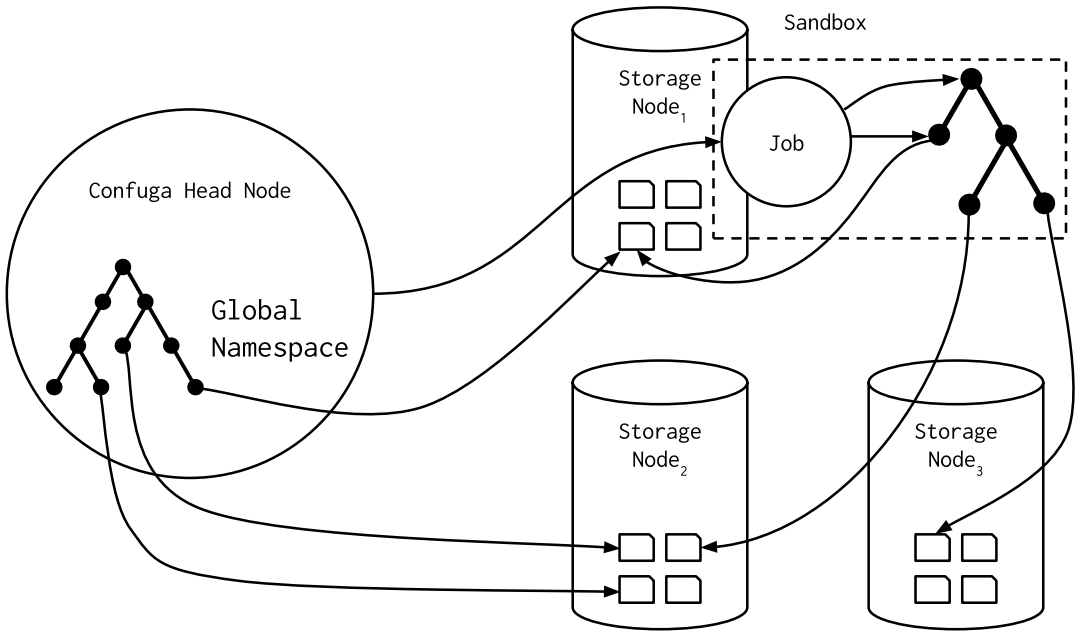


Figure 1.1. Namespace Scoping in Confuga

to support runtime access to any file at any time, Confuga is able to scope job visibility of the global namespace to the job's own defined subset. This is visualized in Figure 1.1.

Requiring the declaration of the job namespace allows Confuga to unobtrusively perform several optimizations which were not possible in prior work:

Scope file system access. In current systems supporting scientific computing, workflow jobs operate within a sandbox that is isolated from the larger workflow dataset. Confuga takes advantage of this restriction by limiting access to the global file system to the beginning and ending of a job. In fact, jobs do not interact with the global file system at all during execution and so cannot dynamically lookup files or read files not included in the dependency list.

Direct transfers within the cluster. Confuga uses the file dependencies for jobs to direct transfers between storage sites. This allows Confuga to optimize transfers for network and disk bandwidth.

Selectively relax control of transfers. While storage systems must normally react to completely unknown access patterns, Confuga is free to allow only some transfers to proceed in an uncontrolled manner. In this way, the scheduler may avoid work which would negatively impact the latency and performance of the cluster.

1.1 Overview

This dissertation will examine the design of an active storage batch system that manages data dependencies at scale with an emphasis on stability. In particular, this work evaluates mechanisms for scalable metadata access and data movement.

Chapter 2: Related Work Data management in distributed systems has been addressed at many levels of software with the goal of shielding the scientist from the mechanisms that make their workflow work. This chapter discusses data management from several viewpoints from the individual devices to distributed file systems to networked clusters and grids of computing resources.

Chapter 3: Architecture of the Confuga File System This chapter develops the design of Confuga, an active storage cluster file system for scientific workflows. Confuga is designed to exploit structural information available in scientific workflows to achieve metadata scalability and cluster stability. The challenge of data management is met by placing novel limitations on the consistency semantics of the distributed file system and by scoping the data access of workflow jobs to its defined dependencies. I will show that Confuga works for representative bioinformatics workflows and achieves its goal of active storage.

Chapter 4: Metadata Management Scalable metadata access in distributed workflows is a recurring problem in system design. This chapter discusses metadata and namespace management in scientific workflows. A mechanism for active storage namespace management is developed and applied to a workflow manager and distributed file system. The chapter concludes with an analysis and evaluation of how

Confuga incorporates these strategies and how metadata scalability is affected.

Chapter 5: Directed Transfers Historically, distributed file systems have supported POSIX-style dynamic and unpredictable file access by applications. Scientific workflows are written with all data dependencies known for each job. Confuga exploits this information by centrally planning and executing transfers for the workflow. These directed transfers are shown to improve transfer performance in the cluster and reduce data-intensive workflow execution time.

Chapter 6: Balancing Directed and Undirected Transfers While directed transfers can massively improve system stability and transfer performance, undirected transfers can provide a mechanism for the scheduler to relax control in order to improve performance. This chapter evaluates the comparative performance of these two transfer methodologies against several workflows. It concludes by showing that a balanced approach provides the best performance without destabilizing the cluster.

1.2 Relevant Publications

- *Attaching Cloud Storage to a Campus Grid Using Parrot, Chirp, and Hadoop* at the 2010 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) [22]. This paper explored implications of attaching Hadoop's HDFS file system to a campus grid. The techniques used in this paper were also applied to Confuga by enabling the Chirp distributed file system to export the Confuga head node.
- *Fine-Grained Access Control in the Chirp Distributed File System* at the 2012 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) [24]. This paper presents authentication tickets which provide an extended credential to authenticate with storage systems while enforcing fine-grained access control to data. Authentication tickets are used extensively in Confuga to provide delegated credentials with limited access to the head node and storage nodes.
- *Design of an Active Storage Cluster File System for DAG Workflows* at the 2013 International Workshop on Data-Intensive Scalable Computing Systems [25]. This workshop paper presents the proposed design of the Confuga active cluster file system.

- *Confuga: Scalable Data Intensive Computing for POSIX Workflows* at the 2015 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) [23]. This paper presents the design of Confuga as of its first prototype. The effectiveness of Confuga's active storage computing, its scalable metadata access mechanisms, and its directed transfer scheduler is evaluated.
- *Balancing Push and Pull in Confuga, an Active Storage Cluster File System for Scientific Workflows* to appear in the Journal of Concurrency and Computation: Practice and Experience. This article presents Confuga's directed and undirected transfer mechanisms used to control load in the cluster. The effectiveness of the two approaches is evaluated. A balanced approach to the two transfer methodologies is shown to most effectively limit load instability while maintaining high transfer performance.

1.3 Cluster Hardware

The experiments in this dissertation use an on-campus cluster composed of a single rack of 26 Dell PowerEdge R510 servers running RedHat Enterprise Linux 6.6, kernel 2.6.32. Each server has dual Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, for 8 cores total, 32GB DDR3 1333MHz memory, a 1Gb link to a Summit X460 switch delivering 220Gbps aggregate bandwidth. Our tests use one Seagate ST32000644NS 2TB disk on each server, with advertised 140MB/s sustained I/O bandwidth, 8.5ms seek time. Each Confuga storage node uses a single disk formatted with the Linux *ext4* file system. For evaluation, we use one node as the head node and the 25 other nodes as storage nodes.

CHAPTER 2

RELATED WORK

2.1 Distributed File Systems

Early distributed file systems shared an authoritative data store for local networks of machines. The most successful of were NFS [73] and AFS [39]. These file systems sought to fulfill a need for a common data store which shared deployments of software and collaboratively edited datasets in networked systems at large institutions.

However, in the realms of cluster and high-performance computing, new designs were needed to support increased parallelism and distributed load. Within that setting, PVFS [69], GPFS [74], Lustre [11], Ceph [88], and Panasas [54] have all been used successfully to support scientific workflows through a global namespace with POSIX consistency semantics.

As large datasets became common, it became necessary to expand the role of the distributed file system to support data-intensive computing. In particular, the file system needs to assist the job scheduler to locate jobs near data. In support of this approach, the Google File System (GFS) [33] was developed for data processing and web applications. GFS divides files into fixed-size 64MB chunks, each replicated across the cluster. It is optimized for workflows which are record oriented with each record less than the chunk size. Writes are performed by appending records to files. Consistency requirements in GFS are relaxed so that appends are defined but may result in inconsistent replicas for a chunk (i.e. two replicas are not bit-wise equal, but they do have defined structure following a successful write).

The most significant difference between Confuga and GFS is the usage of files. GFS focuses on record-oriented files while Confuga optimizes for workflows that use whole-files in a POSIX environment. For this reason, Confuga replicates whole files (not chunks) and each replica must naturally be consistent and bit-wise equal. Furthermore, Confuga's head node must be more involved in cluster transfers since whole files must be transferred between nodes to satisfy dependencies while maintaining a balanced load on the cluster.

Many distributed file systems implement some form of content-addressable storage (CAS) which allows for simple universal inode generation (the inode is the checksum), deduplication, and flat namespaces which index files. For example, Venti [62] and HydraFS [85] use block granularity for storing objects to achieve better deduplication, especially important in an archival system. The CAS technique is also common in distributed version control systems like Git [84] and Mercurial [55] to uniquely identify commits and objects where remote repositories are infrequently available. Confuga also uses CAS to allow storage nodes to assign universally unique identifiers for new files and to deduplicate common files.

2.2 Active Storage

Originally active storage began as smart disks [1, 45, 66]. Small computational tasks were relocated to the storage device for faster I/O and to harness unused processing capabilities.

Object storage devices (OSD) [34] were the eventual realization of active disks. The OSD abstraction provides an abstracted storage container which produces objects tagged with metadata. A set of operations for manipulating and searching/selecting the objects on the devices is part of the standard [87]. OSD has become an integral component of numerous file systems including Lustre [11] and Panasas [54] with the goal of increasing I/O throughput and reducing data movement [75]. Compu-

tation on Lustre storage servers has also been supported [26] to allow client programs to have direct access to data, and in [60] as a user-space solution.

The idea of active storage eventually transitioned away to *smart storage nodes*. Projects like Hadoop were developed for clusters built on commodity hardware that are dedicated to performing structured computation [20, 50, 95] on large datasets. The original use-case of Hadoop was the deployment of MapReduce [20] computations within a cluster. MapReduce provides a mechanism for executing a job on a large monolithic file, with the job divided into tasks that preferably execute on storage nodes hosting a split (or block) of the file.

Confuga is a natural evolution of this approach whereby users can execute whole applications with multiple dependencies and full data locality but do not need to modify their workflow to fit fixed computation frameworks like MapReduce. Clients need only specify the complete list of dependencies for each job and execute jobs conforming to workflow consistency semantics (i.e. no run-time data dependencies).

2.3 Data-Locality Aware Scheduling

The Cplant [12] project is notable for early work in managing distribution of certain shared dependencies (like an executable) across many nodes for parallel job launch. This is done through a spanning tree managed by the client's workflow manager. Confuga's use of push transfers is an evolution of this idea but differs in significant ways: the file system is able to place jobs on nodes which already have many or all of the job's dependencies which avoids redundant effort; replication is globally controlled across the cluster with awareness of all running workflows; replication can be to a subset of nodes/jobs requiring a dependency rather than all nodes; and, Confuga can push multiple dependencies in parallel without potential long tails caused by slow nodes.

BAD-FS [8] built on Cplant's ideas of pre-staging dependencies by joining storage

servers which act as a cooperative cache [19] (a technique where clients use others' caches to distribute load) with compute servers which access data on these storage servers via interposition agents. The use of a local cache allows the workflow to avoid accessing expensive remote or off-site resources. Like Confuga, BAD-FS uses the declarative information from the workflow to inform a centralized scheduler how to manage data and job dispatch. However, Confuga utilizes this information to manage transfers between storage nodes and formalizes the workflow consistency semantics within a file system.

Data Diffusion [64, 65] presents a technique for improving data locality by allowing data to be cooperatively cached across all compute resources. The task dispatch framework used, Falkon, is aware of the cache state on resources and is capable of scheduling tasks near data. Shark [4] is another distributed file system which allows clients to cooperatively cache data to improve scalability.

Workflow managers that operate within grids adopt a limited role for data locality. The usual problem is harnessing execution nodes and delivering data to geographically distinct sites. Pegasus [21] and GridBLAST [47] addressed this by operating in tandem with Globus [10] to deploy off-site dependencies to the local staging site or shared file system. Ranganathan et al. [44] presented a framework for scheduling decisions across a grid composed of multiple virtual organizations and geographic locations. These scheduling decisions may include replicating needed files to local sites.

Pegasus also schedules jobs at compute nodes hosting data otherwise at random nodes. The Stork [46] data scheduler is designed to cooperate with the DAGMan [17] workflow manager to manage data placement. Stork acts as a transfer job manager between distinct sites on the grid and handles fault-tolerance and reliability of transfers. Job data may be stored on numerous data nodes [82] which is discovered and accessed via Parrot [79], a user-level virtual file I/O agent. While data placement

between sites on the grid has been well studied, Confuga manages transfers between active storage nodes within a cluster. Confuga also does not require the user to include data placement requests within the workflow description. Like Stork, Confuga uses the concept of transfer jobs to achieve reliable and fault-tolerant transfers between nodes.

Hadoop has sees a lot of attention to improve its greedy naïve scheduler for MapReduce. Delay Scheduling [94] is a work to improve the fairness of the Hadoop scheduler in clusters with many users, which is common problem for Yahoo! and Facebook in their Hadoop clusters. Delay Scheduling introduces a wait decision for scheduling MapReduce jobs to improve data locality while maintaining fairness. Fischer et al. [27] shows that the problem for scheduling in Hadoop is NP-Complete.

Quincy [42] is a scheduler for Dryad [41] which transforms the job scheduling decisions into a graph based min-max flow problem. It recomputes the solution for each scheduling event (new jobs, new resources available). This solution may result in killing tasks running on nodes, waiting for a node to become available for a task (an extension of Delay Scheduling), or scheduling a task away from input data. The algorithm uses heuristics to change the flow costs on graph edges to influence the decisions.

2.4 Transfer Management

Confuga focuses its attention on controlling transfer load once scheduling decisions have already been made but it uses concepts which appear in distributed and centralized schedulers. Partitioning storage nodes into *map* and *reduce* slots in Hadoop [91] is used to increase utilization of system resources (memory and network resources especially). This allows multiple map tasks to run concurrently and multiple long-running reduce tasks to accept input as map output is produced. Similarly, Hawk [58] uses cluster partitioning with dedicated short job servers. In Confuga, transfer slots

are used to express the demand placed on the networking and disk. In [23], optimal push transfer performance was observed when limiting storage nodes to one transfer slot under large transfer load.

Hawk also implements a hybrid scheduler design with a centralized scheduler for scheduling long-running *large* jobs and distributed schedulers for scheduling *short* jobs. The centralized scheduler improves performance by knowing the distribution of large jobs across the cluster and the associated wait time. The centralized scheduler is able to function under load because there are fewer large job than short jobs. Confuga also uses a hybrid design for transfers: *push* transfers are directed by the central scheduler while *pull* transfers are initiated in a distributed fashion by the individual storage nodes.

Central management of transfers has recently gained traction in data center management. The IOFlow [83] project seeks to implement a *software-defined storage architecture* which allows setting end-to-end policy for the storage systems. In particular, their work allowed establishing guarantees about a tenant machine's network link performance and the performance of routing to other destination machines.

With some success, scientific workflow abstractions have been used to manage data distribution to compute nodes. Instead of just placing tasks near its inputs and coordinating the data transformations, the abstraction also conducts transfers as needed. The All-Pairs [53] abstraction is notable for using this approach: split a large dataset and compare every pair of splits. All-Pairs will manage an efficient spanning tree distribution of the dataset to support parallel comparisons. This approach has been used in biometrics research [13] to perform comparison functions across every pair of subjects.

2.5 Batch System and Resource Acquisition

For workflows running on heterogeneous resources, the execution platform API is not the only concern. Efforts such as OGSA (Open Grid Services Architecture) from the OGF (Open Grid Forum) look to define capabilities services (such as execution and resource management) to guarantee interoperability across different types of resources and systems. Such capabilities can then be provided by particular applications (such as the Globus Toolkit [29]).

There has being extensive work on studying APIs that encapsulate different execution platforms. One of the most well-known is an API specification from the DRMAA [57] (Distributed Resource Management Application API) working group, sponsored by the Open Grid Forum. The goal of DRMAA is to define a set of mandatory API functions and job description attributes that are then implemented on top of particular execution platforms. In this sense, DRMAA is specified as "the greatest common denominator" across the different execution platforms. Coming from the other end, the SAGA [71] (Simple API for Grid applications) specification focuses on mandatory functionality on the application side. In this context, one main result of our work is that there is a limit to the issues of execution platforms that can be solved with an application API. That is, a lowest common denominator execution system API cannot be built without the cooperation of the execution systems.

Common APIs have the immediate disadvantage that some of the execution system features become unavailable to the workflow manager. Managers such as DAGMAN [17] for Condor, or YARN for Hadoop, tightly couple with the underlying execution platform, and can take advantage of particular optimizations and capabilities. Rather than specifying an execution system API to be used by the workflow manager, another approach is to let an execution system provide the common interface. For example, Condor-G [31] appears to the user as a local Condor pool, but it runs on top of other execution system resources.

2.6 Workflow Namespaces

Significant efforts have been made to explore namespaces as a solution to harnessing multiple machines as a single system. This first began with LOCUS [86] which was an early UNIX-compatible system that allowed distributing computation across a cluster of servers with fully transparent access to the file system. LOCUS had objectives of full transparency on placement of processes and data which would enable seamless load-balancing. It accomplished these objectives by presenting a common global namespace visible to all distributed processes for all of the distributed system resources. This made the *location* of an individual process on LOCUS immaterial to its access of resources. This single system image design has had a major influence on later work where a single file system interface or single job submission interface is presented for access to a cluster's resources.

The Plan9 operating system took this a step further by abstracting many resources normally part of the process control block and making them part of the namespace of the process [61]. This allowed Plan9 to abstract several details such as the CPU architecture a binary was compiled for. In this way, declaratively expressing the resources needed to the system provided opportunities for optimization. In the same way, Confuga uses the namespace mapping provided by jobs to optimize and manage transfers within the cluster.

Namespace management is closely related to the scalability of metadata operations in a distributed file systems. For example, distributed file systems like NFS [73] and AFS [39] conform to POSIX consistency semantics and provide a global namespace. AFS is notable for resolving certain performance issues by relaxing consistency semantics using `write-on-close`. Other POSIX extensions for high-performance computing have been proposed [90] which allow batching metadata operations and explicitly relaxing certain consistency semantics.

Today's highly parallel cluster file systems seek to manage metadata scalability

through aggressive optimizations without impacting POSIX consistency semantics. For example, the Ceph [88] cluster file system improves metadata access in the global namespace in several ways. One aspect of this is the decoupling of file system metadata from the file content or replicas. Additionally, Ceph dynamically divides the global namespace into sub-trees across metadata servers to improve spatial locality of metadata access. Processes which access files within the same sub-tree enjoy benefits of this locality and distributed lock-free metadata access. Finally, Ceph uses its CRUSH [89] algorithm to allow storage nodes and clients to autonomously lookup replicas without metadata server involvement.

Still, cluster file systems continue to suffer from a metadata bottleneck [2, 56]. Confuga avoids common metadata issues by designing namespace access for workflow consistency semantics where file access is known at dispatch and visibility of changes are only committed on task completion. By scoping access to the global namespace, Confuga is able to batch many operations (`open`, `close`, `stat`, and `readdir`) at task dispatch and completion and opportunistically prohibit dynamic file system access.

CHAPTER 3

ARCHITECTURE OF THE CONFUGA FILE SYSTEM

3.1 Introduction

This chapter presents the design of **Confuga** [23], an active storage cluster file system designed for executing unmodified POSIX DAG-structured workflows. Confuga combines the *workflow model* of scientific computing with the *storage architecture* of distributed cluster file systems. End users place their datasets in Confuga using standard file manipulation tools and then direct their workflow manager to submit jobs to Confuga. In this way, Confuga acts as a replacement for existing batch execution systems. The user does not need to redesign their workflow or provide additional consideration to the management of data dependencies used in their workflow.

Confuga sets the stage for evaluating data and metadata and management techniques in the context of an active storage system supporting scientific workflows. These techniques are explored in the following chapters.

Confuga is built with several properties that make it useful for this class of workflows:

Explicit namespacing for tasks and workflows. Executing jobs on a batch system is frequently accompanied by difficulties establishing an explicit namespace in which the job operates. In many distributed systems, tasks are executed within a global namespace. This can be useful for establishing a consistent namespace for all work but unfortunately may lead to challenges with strict application programs that expect a static namespace layout during execution. Additionally, workflows composed

of a number of sub-workflows regularly experience namespace difficulties, requiring special handling by high-level tools like Weaver [15]. Confuga solves namespace difficulties by requiring a mapping from the global namespace to the task namespace, for each job. Jobs are not permitted to access the global namespace except as defined by the mapping.

POSIX applications and workflows. Scientific workflows are typically written for execution within a POSIX-compliant sandbox or within a common global namespace. Because it is common for applications to be developed over several years on a single machine environment, it is difficult to port these applications to structured distributed systems that enforce a certain API for data access. Confuga allows applications to run within a sandbox with input file dependencies available in the sandbox as described by a namespace mapping. Applications can execute normally without modification.

Metadata scalability. Any file system that provides a global namespace must have a global service to provide metadata regarding the location and status of each file. This service can be implemented as either a centralized server or as a distributed agreement algorithm, but either way, the service does not scale. Confuga avoids this problem by exploiting the structural information available in the workload.

Load scalability. As workloads scale up, and the number of simultaneous users increase, it is all too easy for concurrent transfers and tasks to degrade each other's performance to the point where the entire system suffers non-linear slowdowns, or even outright task failures. For example, it is frequently observed that too many users running on Hadoop simultaneously will cause mutual failures [94]. Confuga avoids this problem by tracking the load placed on the system by each task or transfer, and performing appropriate load management, resulting in a stable system.

Drop-in replacement for existing batch and file systems. Confuga is used as a drop-in replacement for a batch system and a file system, combined into a single

entity that can be invoked by existing workflow managers. Confuga also leverages existing file system technologies which allowing regular interaction by normal system utilities.

3.1.1 Origin of the Name

Confuga is a portmanteau of the Latin word *con* and Italian's *fuga* (also Latin). A fugue (fuga) is a contrapuntal musical composition composed of multiple voices. Here the intended meaning is "with many voices" although it literally translates as "with chase".

That name no longer has any meaning for me.

– Darth Vader (Return of the Jedi)

3.2 Architecture

Confuga is a cluster file system used to coalesce multiple storage sites into a single global namespace while enabling robust support for job execution. Users interact with the cluster by uploading datasets for their workflow and then submitting jobs through a workflow manager.

A Confuga cluster is composed of a single head node and multiple storage nodes. The head node manages the global namespace, indexes file locations and metadata, and schedules jobs on storage nodes. The individual storage nodes run as dumb independent active storage servers. The architecture is visualized in Figure 3.1.

3.2.1 Storage Model

Just like GFS [33] and HDFS [76], the Confuga head node manages the cluster namespace and other file system metadata. Metadata and directory hierarchy operations like `stat`, `mkdir`, and `unlink` only require changes to the state on the head

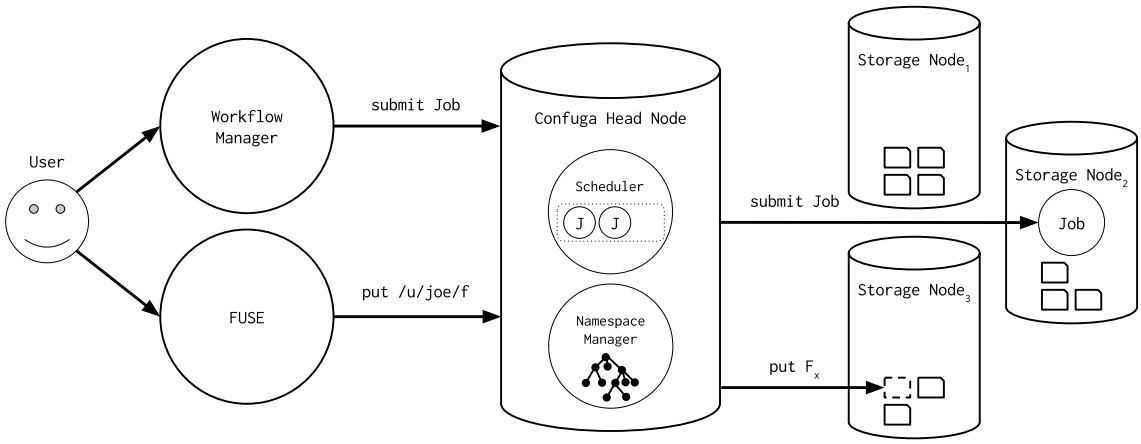


Figure 3.1. Confuga Architecture

node. Confuga is designed with the expectation that external metadata operations are infrequent and all other metadata transformations are executed by services (e.g. the job scheduler) executing on the head node.

An `open` of a file by external clients is exclusively mediated by the head node. Clients open a file for reading by requesting a file handle from the head node which may be used for subsequent `read` operations and finally closed. In the background, the head node will lookup an available replica, connect to the host storage node, and perform read operations on the replica. **In this way, the head node acts as an intermediary for external clients for all file operations.** Opening files for writing works similarly except the head node selects an available and random storage node for the new incomplete replica. When the client closes the file, the head node *seals* the replica making it immutable and *updates* the global namespace (the containing directory) with the new file and its associated replica. With these restrictions, Confuga has only a few caveats for client operations on files: file writes are globally visible only after closing the open file handle and files may only be written once.

We have chosen this model of Confuga mediating replica creation and reading to

ease development and to simplify the security domains between clients and storage nodes. In particular, it allows external clients to browse the file system and load data into or pull data out of the cluster by simply connecting to the head node.

Storage nodes are used as dumb storage, unaware of their role within the cluster file system. File replicas are stored in whole on one or more storage nodes, not as chunks as in GFS/HDFS. Each replica is managed and tracked by the head node. Replication is performed by transfers created by the head node. This allows for redundancy and increased data parallelism for jobs. Storage nodes and the jobs they run do not independently interact with the head node. Instead, the head node dictates which replicas on a storage node the job may access.

The head node tracks replicas within a flat namespace on storage nodes. Replicas are named according to their replica identifier (RepId) that is either a universally unique identifier (UUID) or the SHA1 hash of the replica content. Hashes are used for basic deduplication of files. In most circumstances, a SHA1 hash is used for the RepId except when a large output file is created by a job. Because jobs operate within a sandbox on a local POSIX file system, storage nodes cannot compute the hash of output files until after the job executable exits. To avoid delaying job completion in order to hash large output files (currently >16MB), the storage node will assign a UUID instead. The head node learns the RepId of new output files after reaping jobs. Overall, the use of RepIds allow storage nodes to safely assign a content identifier for new files created by jobs without head node involvement.

Externally, Confuga's file operations largely follow POSIX consistency semantics except new files are visible after close and files may only be written to once (just like HDFS). We have found these semantics are sufficient for clients to upload, manipulate, and download their datasets in Confuga. We emphasize that the intended goal of the Confuga file system is to support running jobs and not to be used as an external file store. Job namespaces and consistency semantics are discussed in

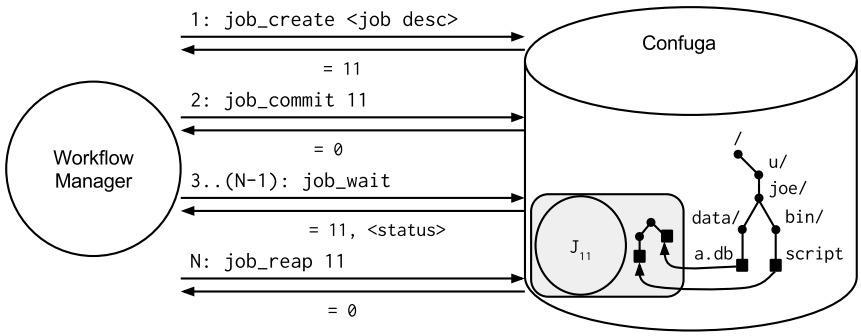


Figure 3.2. Confuga Job Execution Protocol

© 2015 IEEE.

Chapter 4.

3.2.2 Execution Model

After users place workflow datasets on Confuga, they may begin running workflows. As shown in Figure 3.1, Confuga presents itself as a single system image [86] which executes multiple jobs that read from and write to the cluster file system. That is, Confuga appears as a powerful monolithic system to the user and workflow manager. Each job executes an opaque executable within a private job namespace (i.e. a sandbox) constructed from a specification of the job's input and output files. During execution, jobs cannot see the global file system, only the sandbox. All input files are read atomically prior to a job starting. On job completion, the global namespace is atomically updated with the new output files by the head node.

Jobs are submitted to Confuga using a traditional *submit* and *wait* RPC interface with two-phase commit for reliability. The job protocol is shown in Figure 3.2. Each job specification is encoded in JSON [18], with typical attributes like the executable name, arguments, and environment. Confuga also requires the inclusion of the *job namespace* which lists the mapping of input files from the global namespace to the sandbox and of output files from the sandbox to the global namespace. This names-

```
{
   "executable":"./script",
   "arguments":[
      "./script",
      "-a"
   ],
   "files":[
      {
         "serv_path":"/u/joe/bin/sim",
         "task_path":"script",
         "type":"INPUT"
      },
      {
         "serv_path":"/u/joe/data/db",
         "task_path":"data/db",
         "type":"INPUT"
      },
      {
         "serv_path":"/u/joe/wf/out1",
         "task_path":"out",
         "type":"OUTPUT"
      }
   ]
}
```

Listing 3.1: Simple Confuga Job Description in JSON

pace mapping is static and cannot be changed during job execution. Additionally, file access is not strictly confined to files in the job sandbox to permit usage of executables (such as the shell), libraries, and other files available on the system. Namespace management is discussed in detail in Chapter 4. Listing 3.1 shows an example job specification.

Normally, users do not concern themselves with writing these job specifications. Instead, Confuga expects to be invoked by a workflow manager, the user agent which submits and manages jobs on behalf of the user. Confuga does not order job execution by any specified dependency, this is the workflow manager's responsibility. Jobs are tied together through a directed acyclic graph (DAG) which orders jobs by file dependencies: one job's output file becomes the input of the next. Figure 3.3 shows a typical DAG-structured workflow run by Confuga.

Our collaborators use the Makeflow [3] workflow manager that builds on the venerable Make syntax for expressing job dependencies, which creates an implicit job execution order. Given a Makeflow specification file, Makeflow creates a DAG of the

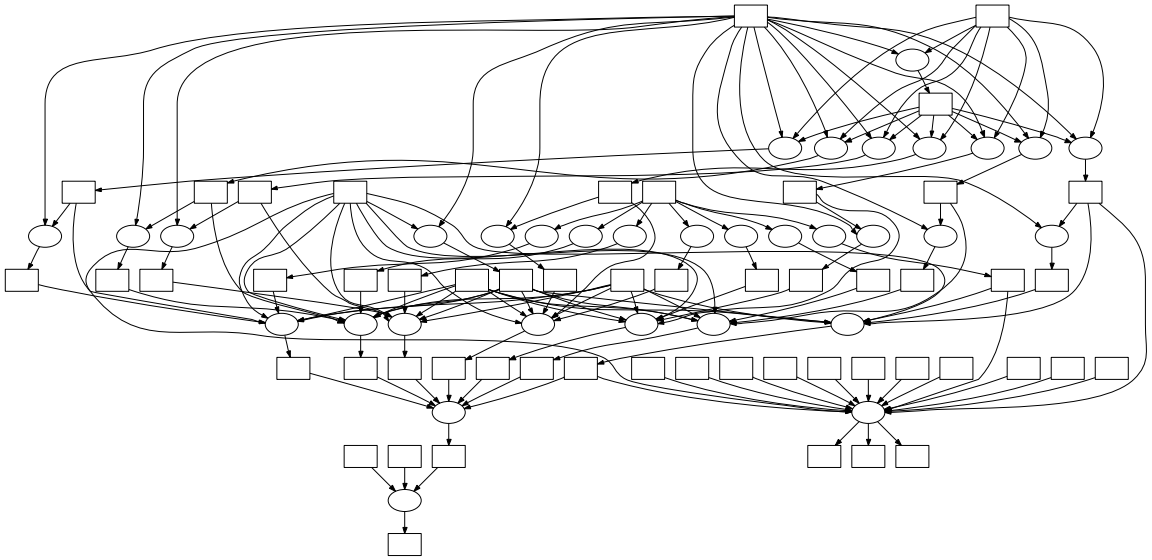


Figure 3.3. A Typical DAG-structured Workflow

This comes from a bioinformatics workflow using SHRiMP [70]. Here files are boxes, circles are tasks, and lines indicate dependencies.
© 2015 IEEE.

entire workflow, submits jobs as dependencies become available, and handles certain workflow fault tolerance policies. It is designed to easily switch between execution platforms and currently supports Condor [49, 82], SGE [32], Work Queue [14], and other systems. To programmatically create large workflows and the workflows used in our experimental results, we use the Weaver [15] workflow compiler.

Concurrent job execution in Confuga comes from dispatching jobs to multiple storage nodes. A scheduler on the Confuga head node handles the details of assigning jobs to storage nodes for execution, replicating necessary dependencies, and global namespace manipulation. The head node monitors the health of storage nodes via heartbeat messages from storage nodes sent to a catalog service. Using the catalog, the head node learns of unavailable storage nodes, newly available storage nodes, and other file system statistics. When the scheduler learns of a failure because of a lost storage node or a failed job, it will reschedule the job if the failure is transient (e.g.

a failed transfer) or pass the failure to the workflow manager if it cannot be handled.

In Confuga, data parallelism is achieved in two ways. Firstly the user constructs their workflow in a way that jobs use whole dependencies. (This model is often structurally incompatible with other big data abstractions like Map-Reduce where data parallelism is established by mapping jobs to chunks of a monolithic file while expecting each job to largely read only the mapped chunk.) Because Confuga must work to get all data dependencies at the site a job executes, the effort is only justified if the job actually needs the whole dependencies. Secondly, Confuga adjusts the replication of data to respond to needs of jobs. When a job dependency is missing from the storage node that the job is to be executed on, Confuga will plan the replication of the file to that storage node. This dynamically responds to demand for hot files and allows increased replication to benefit future jobs relying on that dependency.

Each job submitted to Confuga goes through several states. First, the scheduler performs namespace remapping which allows the job to be executed on storage nodes. This is a static translation that is independent of the storage node the job will execute on. Once the new namespace mapping is constructed, it is scheduled or assigned to an available storage node with preference towards a node with the most input file bytes (as some files are larger than others). Next the head node decides how to replicate missing input files since jobs must execute with all inputs files in their sandbox. Finally, the job is submitted to the storage node for execution. After several periodic waits, the scheduler will reap the finished job and set the job's outputs in the global namespace.

3.2.3 Implementation and Use

The Confuga cluster uses the Chirp [81] distributed file system for storage nodes. Chirp was originally designed for providing remote data access to jobs running on the

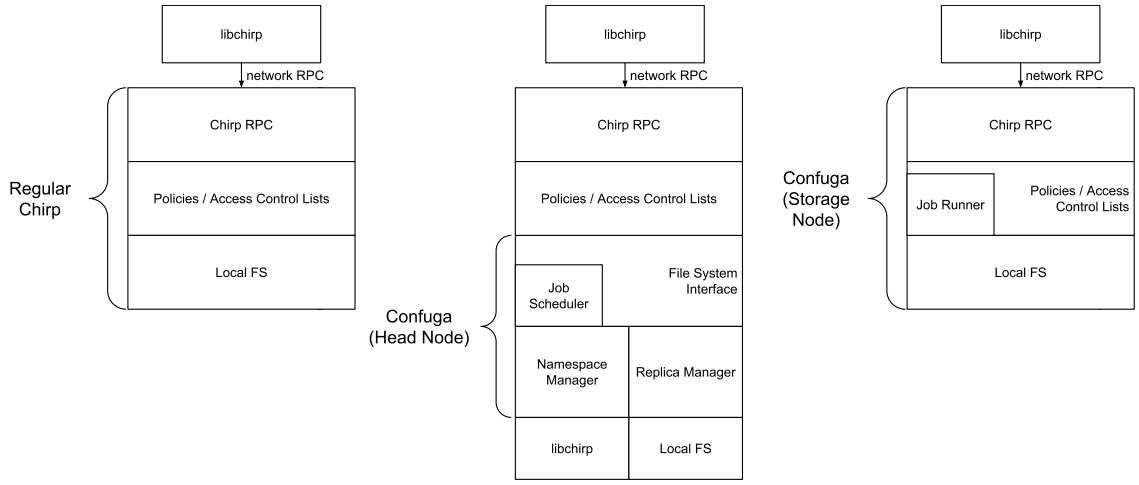


Figure 3.4. Software Stack Supporting Confuga

Grid and built for use with Parrot virtual file system adapter [79] and Condor [82]. Confuga utilizes Chirp for storage nodes due to ease of deployment by unprivileged users and several core features including authentication and access control. Still, while Chirp provides most features that Confuga needs to store and manipulate files, it was missing a robust job framework. Extensions to the protocol to support jobs is discussed in Section 3.3.

Confuga also uses Chirp to operate and *export* the head node for access by external clients[1]. A module was added to Chirp to interface with Confuga. Normal file I/O RPC like `stat` are redirected to the appropriate Confuga API call `confuga_stat`. Additionally, Confuga must perform several cluster upkeep operations and job scheduling asynchronously with Chirp RPC handling. The job scheduler added to Chirp to schedule and execute jobs is used to execute the Confuga daemon process. Figure 3.4 visualizes the software stack of the Confuga head node and storage nodes.

Because Confuga is exportable by a Chirp server, Clients may interact with

[1]The default exported file system is the local file system on the system running the Chirp server. Chirp also supports exporting other file systems like HDFS [22]. Confuga is itself another exported file system.

```
chirp_server                                                    \
    --jobs                                                      \
    --root='confuga://./confuga.root/?nodes=file:nodes.lst&auth=unix'
```

Listing 3.2: An Example Command to Start a Confuga Head Node

```
chirp://node1.nd.edu:9094/users/pdonnel3/.confuga
chirp://node2.nd.edu:9094/home/pdonnel3/.confuga
chirp://node3.nd.edu:9094/.confuga
```

Listing 3.3: nodes.lst

Confuga using a familiar POSIX-style I/O interface to interact with the file system through convenient command line tools, Parrot [79], FUSE [78], or the Confuga API. The user is free to organize files in a regular directory hierarchy with per-directory access controls that enable fine-grained sharing with colleagues.

Confuga is available in the Cooperative Computing Tools [2] software package. A Confuga cluster may be trivially started by configuring a Chirp server to act as the head node. Listing 3.2 shows a bare-bones command to start a head node. Listing 3.3 lists the cluster storage nodes in a file.

All options specific to Confuga are communicated via the `--root` switch passed to the Chirp server. The root URI includes the location of the head node state (`./confuga.root` in Listing 3.2) and miscellaneous options. Each option is joined by an ampersand (`&`). Listing 3.4 lists the current options Confuga accepts (from the `confuga(1)` manual page).

The head node manages its state in a directory on the local file system. This is specified in the root Confuga URI on the command line. The global cluster namespace is completely represented within this directory as a normal hierarchy of files and directories. Confuga stores in each file its associated RepId (corresponding to its content).

[2] Confuga was merged in commit `f485c6701b217180c2c4c98c0170e1469fd6191b`

```
auth <method>
      Enable  this method for Head Node to Storage Node authentication. The default
      is to enable all available authentication mechanisms.

concurrency <limit>
      Limits the number of concurrent jobs executed by the cluster. The default is
      0 for  limit less.

nodes <node-list>
      Sets  the  whitespace or comma delimited list of storage nodes to use for
      the cluster. May be specified directly as a list node:<node1,node2,...> or
      as a file file:<node file>.

pull-threshold <bytes>
      Sets the threshold for pull transfers. The default is 128MB.

replication <type>
      Sets the replication mode for satisfying job dependencies. type may be
      push-sync or  push-async-N. The default is push-async-1.

scheduler <type>
      Sets the scheduler used to assign jobs to storage nodes. The default
      is fifo-0.

tickets <tickets>
      Sets tickets to use for authenticating with storage nodes. Paths must
      be absolute.
```

Listing 3.4: Confuga Head Node Options

The head node also uses a SQLite database for managing the state of the cluster: location of file replicas, storage node heartbeats and the state of jobs. Jobs and replication are evaluated as state machines (a) to allow recovery in the event of faults by the head node or by the storage nodes and (b) to make scheduling and replication decisions using the complete picture of the cluster's current activities.

3.3 Augmentations to Chirp: Job Protocol

The Chirp protocol is closely modeled after regular UNIX system calls; its design was intended to closely correspond to the interface a regular application would use for speaking to a file system. Much of the design of Chirp's protocol was clearly meant to meld cleanly with the primary Chirp client application, Parrot. In general, Parrot directly passes intercepted system calls directly to the Chirp server with minimal or no modification. For this reason, Chirp implements a stateful protocol. Clients operate on opened files which eventually must be closed. The Chirp server maintains

```
1 ok, id     <- job_create(exe, args, env, bindings)
2 ok         <- job_commit(id)
3 ok, status <- job_status(id)
4 ok, status <- job_wait(id)
5 ok,        <- job_reap(id)
6 ok,        <- job_kill(id)
```

Listing 3.5: Chirp RPC for Manipulation of Active Storage Jobs

a connection with the client, remembering all open files in use by the client.

Confuga extends the Chirp protocol by adding various job control RPC, shown in Listing 3.5. The creation and deletion of a job may seem similar to the opening and closing of a file, but this similarity is only superficial. Jobs, once started, occupy resources on the Chirp server including CPU time, disk I/O bandwidth, and memory. They also are expected to operate through temporary network hiccups where the client's connection to the Chirp server is lost. For this reason, there must be a *persistent* job identifier associated with the job.

Creating this identifier requires a two-phase commit [35] protocol for the client. The essential reason is due to a possible failure between when the Chirp server creates the job and when the client securely records the job identifier. The Chirp server continues execution of the job even though the client has no knowledge of the success. To avoid this problem, creation of a job requires the client to acknowledge receipt of the identifier, through a `job_commit` RPC. Only after a job is committed will it be ready for execution.

The `job_create` RPC accepts a JSON [18] encoded job description. It provides an expressive mechanism of creating a process similar to UNIX `fork(3)` and `execve(3)`. A simple example executing shell code is shown in Listing 3.6. The definition of the job's input and output files contrasts with UNIX as files on the Chirp server are not dynamically accessible from the job's sandbox. Instead, files are bound from the server's namespace into a transient job sandbox. The way the file is bound is configurable but by default uses hard links to the actual server file. Jobs are also

```
{
  "executable": "/bin/sh",
  "arguments": [
    "sh",
    "-c",
    "echo Hello, world! | cat - input > output"
  ],
  "files": [
    {
      "task_path": "input",
      "serv_path": "/u/patrick/input.1",
      "type": "INPUT"
    },
    {
      "task_path": "output",
      "serv_path": "/u/patrick/output.1",
      "type": "OUTPUT"
    }
  ]
}
```

Listing 3.6: Chirp Job JSON Encoding

allowed access to files that strictly-speaking exist outside of the job namespace. In the above example, the job is using the system shell, `"/bin/sh"`. The development of this mechanism is explored in detail in Chapter 4.

There is also a set of RPC to wait for job completion. The `job_wait` RPC mimics the UNIX `wait(3)` system call except it also has a two-phase commit *reap* RPC, `job_reap`. Again, this is to ensure that the job submission site logs completion of a job successfully before the Chirp server reaps the job. Job status information is returned by the `job_wait` RPC in JSON, as shown in Listing 3.7.

Clients may also selectively wait for jobs to finish in Chirp using a workflow identifier. Currently, this is a simple integer that corresponds to all of the user's jobs. The limited expressiveness of this selection mechanism actually caused issues for us with our workflow manager: when the workflow manager was forcibly restarted, completed jobs for the previous instance would cause all calls to `wait` to immediately return. While this did not result in incorrect behavior, it is obviously undesirable to do a busy wait. This is resolved by a more expressive and selective wait RPC which matches a *tag* associated with the running (or restarted) workflow and its jobs. This

```
[
   {
      "id":3,
      "executable":"/bin/sh",
      "exit_code":0,
      "exit_status":"EXITED",
      "status":"FINISHED",
      "subject":"unix:batrick",
      "time_commit":1456326442,
      "time_create":1456326428,
      "time_finish":1456326442,
      "time_start":1456326442,
      "arguments":[
         "sh",
         "-c",
         "echo Hello, world! | cat - input > output"
      ],
      "files":[
         {
            "binding":"LINK",
            "serv_path":"/u/patrick/input.1",
            "size":null,
            "tag":null,
            "task_path":"input",
            "type":"INPUT"
         },
         {
            "binding":"LINK",
            "serv_path":"/u/patrick/output.1",
            "size":551,
            "tag":null,
            "task_path":"output",
            "type":"OUTPUT"
         }
      ]
   }
]
```


Listing 3.7: Chirp Job Status Following Completion

is to be implemented in future work [3].

Chirp uses a SQLite database is used for maintaining the state of jobs. This allows for concurrent access and modification of the table of jobs by multiple instances of the Chirp server, each serving a single client. The job framework is logically divided into two separate systems: a job management layer which registers intents (e.g. kill job 1) and a scheduler which actually executes jobs. The scheduler runs as a single instance alongside the Chirp server, using the SQL database to process job intents. This consists of scheduling and starting jobs, canceling jobs which have been killed, and waiting for jobs to finish.

[3]CCTools issue: `https://github.com/cooperative-computing-lab/cctools/issues/373`

3.4 Authentication

Access to storage within Confuga is protected through three authentication realms: client to head node, head node to storage nodes, and storage node to storage node.

Client authentication with the head node is achieved through several interoperable enterprise technologies including Kerberos and GLOBUS. Clients (via Makeflow and the Chirp toolset) use the Chirp protocol to interact with the head node. The first step on connection is to establish a subject credential with the client by iterating over a number of authentication mechanisms. The head node can be configured using `chirp_server` options to set the client authentication mechanisms.

The head node is configured at startup to use a specific authentication mechanism to access all storage nodes. The head node's credential enables complete access to all cluster state (such as replicas) located on storage nodes. For the Notre Dame campus cluster, we use a long duration ticket credential [24] which provides the strict subset of access the head node should have on storage nodes. Configuration is done through Confuga's auth URI option.

Storage nodes access other storage nodes using a separate ticket which is setup and periodically renewed by the head node. This ticket provides an even stricter subset of access, following the principle of least privilege, that only allows reads of replicas and the creation of new replicas in a separate directory. The latter restriction allows Confuga to check for successful replica creation (with consideration to myriad failures) before moving the new replica to the flat replica namespace on the storage node. Storage nodes must allow ticket authentication (by default this is so) to be part of a Confuga cluster.

Table 3.1 summarizes the authentication realms.

TABLE 3.1

CONFUGA AUTHENTICATION REALMS

Realm	Where	How
Client → HN	HN	`chirp_server --auth=<mechanism>`
HN → SN	HN	`chirp_server --root=confuga://...?auth=<mechanism>`
SN → SN	SN	`chirp_server --auth=ticket`

```
1 unix:pdonnel3 rwlda
2 hostname:*.nd.edu rl
3 address:127.0.0.1 rl
```

Listing 3.8: A Chirp ACL File

3.5 Access Control

Because Confuga is meant to support cooperative use by many users, it includes robust access controls set per-directory. This support is not actually implemented in Confuga's core but overlaid by the Chirp server running the head node. Chirp augments exported file systems in a number of ways to make it better suited for remote access in a grid or cluster. One of these augmentations is access control lists (ACLs). Chirp's ACLs dictate the access a credential is given in a directory and is maintained on a per-directory basis. An ACL file is maintained as a regular file on the underlying file system within its corresponding directory, hidden from view by Chirp clients. Listing 3.8 shows an example ACL file.

The initial prototype of Confuga treated an ACL file as normal replicated file like any other. This caused systemic slowdowns in all uses of Confuga as any client initiated file access would require remotely querying a storage node hosting a replica of the ACL. Clearly this was not a favorable situation from a design point. An ACL

```
meta:00000000000000000000000000000000000000000:0000000000000013
unix:batrick rwlda
```

Listing 3.9: Example Metadata File Storing a Chirp ACL File

```
CONFUGA_API int
confuga_metadata_lookup(confuga *C,
                        const char *path,
                        char **data,
                        size_t *size);
CONFUGA_API int
confuga_metadata_update(confuga *C,
                        const char *path,
                        const char *data,
                        size_t size);
```

Listing 3.10: Confuga Metadata File API

file is logically metadata and should be stored with other file system metadata on the head node, not as a regular file on a storage node; the client of the ACL file is the metadata server, not jobs running on the storage nodes. However, Confuga itself does not have a concept of the ACL file. The ACL is managed by the Chirp server operating the Confuga head node. It would be improper to leak the ACL abstraction into Confuga, so a *metadata file* type was designed for hosting metadata that need not be replicated or accessed by storage nodes.

The head node distinguishes metadata files from normal files by a content header. Normal replicated files store the RepID and file length. Metadata files have a different header shown in Listing 3.9. A nil RepID is indicated (the second field) as the metadata file has no replicas. Instead, the metadata is located following the header.

Normal POSIX distributed file systems constrain clients to conventional namespace metadata operations, such as `mkdir` or `chmod`. Confuga introduces a novel API for creating these metadata files which are not stored as regular files to improve access time and locality. Listing 3.10 shows the API that Chirp uses to store and lookup metadata files (like the ACL file).

The POSIX.1e draft standard defined an API for manipulating *extended attributes*

of files and directories. Each attribute is identified by a name belonging to some namespace used to isolate system attributes (like ACLs) from general user attributes. Attribute values are free-form blobs of (possibly binary) data. Extended attributes were proposed to associate metadata with files such as sensor configuration data, origin website, date of acquisition, etc. Confuga's metadata files are similar to extended attributes but allows the file's *content* to be metadata. It makes clear the intent that the file is logically metadata and should not be stored with other file content remotely.

3.6 Errors

Responding to errors is often the most challenging aspect in the design of a system. The developer must typically decide whether to defer error handling up the stack (ending at the user), retry the operation if the failure is potentially transient, or do something else entirely. Users of Confuga may encounter several classes of errors during the course of a workflow. Because Confuga acts as both the file store and the execution platform, I/O errors and opaque application errors may be returned by jobs.

3.6.1 Client File I/O Errors

For regular file I/O through the Chirp protocol, the errors which may be returned are the same as in Chirp which loosely follows the UNIX API. Because Confuga is built on the Chirp protocol, it must respond with errors to some operations which are not supported by Confuga but permitted by the protocol. For example, Confuga does not allow opening an existing file for writing (without truncation) or random writes to open files [4]. If a random write is attempted, a generic UNIX `EINVAL` error

[4] Hadoop's HDFS also has this restriction.

is returned indicating the offset is invalid.

File creation is performed by developing a new replica on a storage node with mediation by the head node. The actual namespace is not modified until the open replica is closed and sealed (i.e. becomes immutable). Because the new file is not added to the containing directory until `close`, the `close` operation may (rarely) fail due to out-of-space errors. This is not a failure mode unique to Confuga or even local file systems, ZFS [43] is a well-known example which may encounter out-of-space errors on close due to the use of copy-on-write.

Because files are created through a unknown number of writes to form a single replica, it is possible for a new unsealed replica to be placed on a storage node with insufficient space. The client may not learn this until late in file creation. Confuga tries to mitigate this possibility by preferring placement on storage nodes with more free space. The API presently has no mechanism to provide Confuga with hints on the size of the replica for proper placement. Additionally, because replicas are stored whole and not as blocks, there is the possibility that a replica cannot fit on any storage node. The prototype API is missing a way to query this.

3.6.2 Job Errors

There are a number of errors jobs may encounter from creation to completion. Among these include missing job dependencies, access control failure, transfer failure, storage node failure, connection failure, and missing output directories. Generally, Confuga tries to resolve most errors internally without involving the client (the workflow manager). The most common type of error returned to the client is due to missing dependencies or other problems with the job file requirements. Below are the stages of Confuga jobs and the possible errors which may occur.

Create: Creating a job involves two tasks: (1) generate a new job identifier and record the job specification to permanent storage; and (2) check the access control

list for each job input and output (recursively for directories). The most common error during job creation is missing dependencies (`ENOENT`) or access control failures. These errors are caught early by the Chirp server exporting the Confuga head node.

Execute: After the workflow manager commits the job, Confuga proceeds with executing the job on the storage cluster. This involves numerous stages where errors may occur.

1. The head node binds input files to replicas [5] which involves a `lookup` of each input file name in the global namespace. Missing files will result in an unrecoverable error. This binding of input files is performed once even when the job is rescheduled due to later errors.

2. Once input files are bound, Confuga will schedule the job to a storage node and replicate missing dependencies. This stage of the job's lifetime will commonly encounter errors such as failed transfers and temporarily lost storage nodes. Confuga conceals these errors by rescheduling a job when the operations cannot be tried. Despite the usefulness of handling these errors at the head node, the workflow manager may desire knowledge of the rescheduling or progress of replication of dependencies. This would allow the workflow manager to give the user better progress information to indicate stalls so the user may choose to take action. Future work on Confuga may expose this information as part of the job's status which is queried by the workflow manager.

3. After job dependencies are replicated, the head node dispatches the job to the storage node. The most common errors encountered here are network failures or temporarily lost storage nodes. These operations are retried for a time until the head node gives up and reschedules the job to a new storage node.

4. Finally, once the job completes execution on the storage node, the head node updates the global namespace with the new job outputs. Errors here are only a result of concurrent manipulation of the global namespace which prevents the head node from adding the outputs to the global namespace. For example, if another client removes a destination directory, inserting the job's output is no longer possible which results in an error (`ENOTDIR`).

Reap: After Confuga has completed the job, the workflow manager may retrieve the job status by waiting for completion via `job_wait` and `job_reap`. Beyond

[5] Atomically reading input files is explored in Chapter 4.

transient network and database failures, there are no other errors defined for these operations.

3.6.3 Other Errors

The most challenging errors to deal with in Confuga are those not directly related to the task being performed. For example, if an authentication ticket has expired or a storage node's state has been wiped then some operations will mysteriously fail for sometimes opaque reasons. For example, a transfer may fail because two storage nodes cannot authenticate. The resolution to this problem is not to schedule a new transfer on a different storage node. Instead, the head node must determine which of the potential problems caused the authentication failure. For example, the storage node to storage node authentication ticket may have expired or somehow become invalid or the access controls on some directory were changed. These types of potential (and fortunately rare) errors are numerous and require continued iterated development on the current prototype.

3.7 Evaluation of Active Storage Capability

So far, we have introduced the Confuga cluster file system. The next chapters will examine load control and scalability mechanisms employed by Confuga. Here, we will present a brief evaluation of its active storage capabilities for scientific workflows.

I have used two unmodified bioinformatics workflows, BLAST and BWA, for benchmarking Confuga. The BLAST workflow is composed of 24 jobs with a shared 8.5GB database. It is used for comparing a genomic sequence query against a reference database, yielding similar sequences meeting some threshold. The BWA workflow performs a genomic alignment between a reference genome and a set of query *reads*. The BWA workflow is composed of 1432 jobs, starting with a 274 way split of the 32GB query. The purpose of this alignment is to later compare how well the

reads align and where in the reference genome they align.

This evaluation will examine how Confuga responds to repeatedly executing the workflows on the same dataset. This is a particularly common scenario for workflows which utilize a database (as with BLAST). Our design of Confuga is explicitly intended to support and optimize for this class of workflows by supporting dynamic increases in file replication to increase data-parallelism and to maintain that replication so future workflows benefit. So, the first workflow pays the cost of increasing the number of replicas for its dataset. Subsequent workflows benefit from that prior work by having its jobs immediately scheduled on nodes hosting that dataset, thereby *returning to the data*.

Figure 3.5 shows two variations of workflows returning to the data. The hardware used in the cluster is as described in Section 1.3. Each graph indicates the activity on all storage nodes, both for transfers in/out and for jobs executing. For these tests, Confuga is configured to use use a simpler synchronous transfer mechanism for moving replicas between storage nodes. This means that the scheduler does a blocking transfer for each replica, preventing any other work during the transfer. Synchronous transfers are discussed further in Section 5.2.

In the first graph, we execute the same BLAST workflow twice consecutively. An 8GB dataset is uploaded immediately prior to running the first workflow, with replicas striped randomly across the cluster. The BLAST workflow executes with 24 long running (approx. 45 minutes) jobs that each share an 8GB dataset split into multiple input files. The goal is to have the second run of BLAST benefit from the prior replication work during the first run. From the graph, one can see that Confuga is initially busy directing copies of files from multiple storage nodes to SN_{12}, where the first job is dispatched. Storage nodes 1 and 12 are arbitrarily chosen as sources for replicas of the common input files for wider replication. This is shown in the numerous transfer tic marks in the first 35 minutes of the workflow. The final

stage of the BLAST workflow is 3 fan-in jobs which gather previous outputs, causing several transfers at `t=01:45`. Because the two BLAST workflows are identical and the outputs are small, these results were deduplicated so the second execution of BLAST runs the 3 analysis jobs without any fan-in transfers.

For the second graph, we run the same BLAST workflow twice consecutively, as before, but also run the BWA workflow concurrently with the first run of BLAST. The intent of this experiment is to determine how Confuga responds to two workflows with disjoint datasets and how data locality is affected. The BWA workflow begins with an initial fan-out job that splits a 32GB dataset into 274 pieces at SN_{11}. For the duration of the BWA workflow, these splits are transferred from SN_{11} to 6 other storage nodes. These transfers appear continuous and concurrent only because of the graph's minimum width of a transfer is 30 seconds (to ease visibility). The final fan-in job is also run on SN_{11} which gather outputs from the other nodes. We can see that the BWA workflow taxed the scheduler with the large number of jobs and synchronous transfers, preventing complete use of the cluster. Eventually, SN_8, SN_9, and SN_3 (briefly) were picked up by BWA late in the run. Because SN_8 was claimed by BWA, SN_6 was chosen for the next BLAST job (with some dependencies already there).

These experiments show that Confuga is able to execute workflows with full data locality and preserve prior work (the replication) to rapidly execute subsequent jobs operating on the same dataset. This validates the primary goal of Confuga to provide active storage capabilities to scientific workflows. In the case of BLAST in the first experiment, a second run of the same BLAST is 19% faster than the original run because its database is already distributed across the storage nodes. Finally, we have shown that data locality is a principal consideration for the scheduler which allows the cluster to execute workflows concurrently with disjoint datasets without significant mutual disruption.

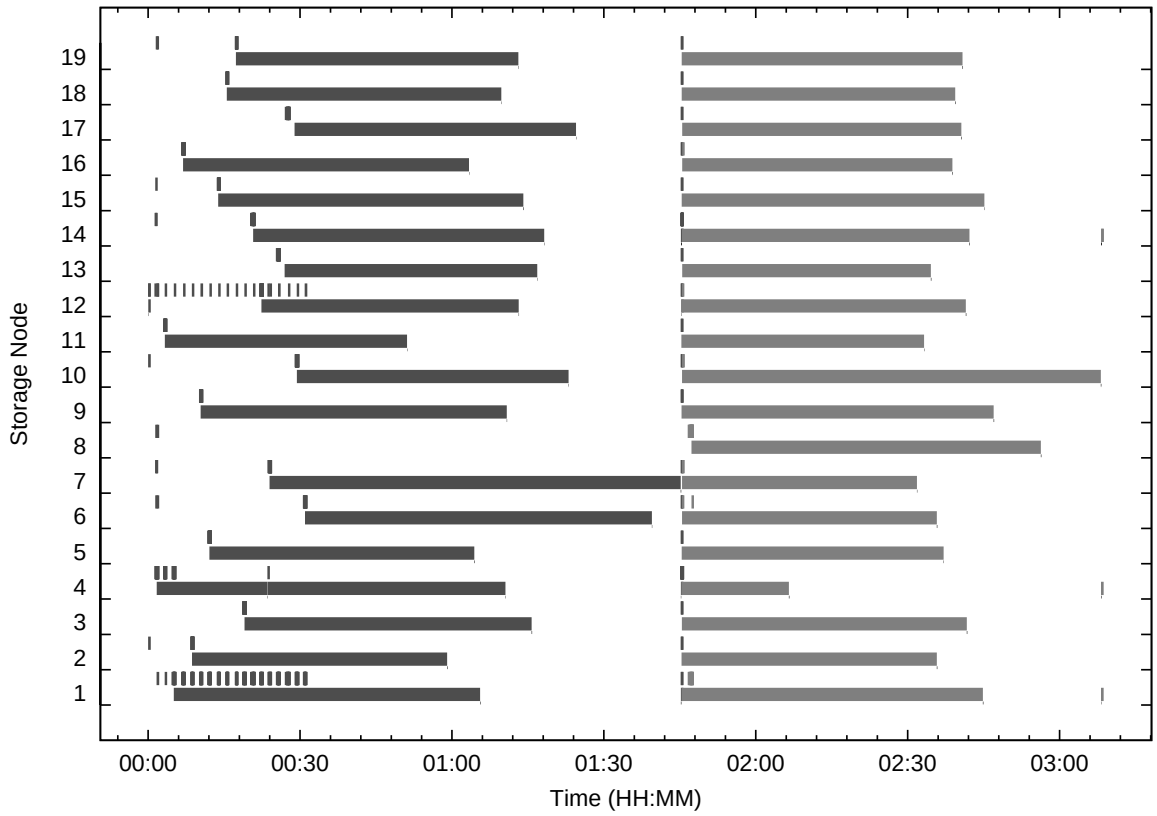
Figure 3.5. Hot Active Storage Experiment

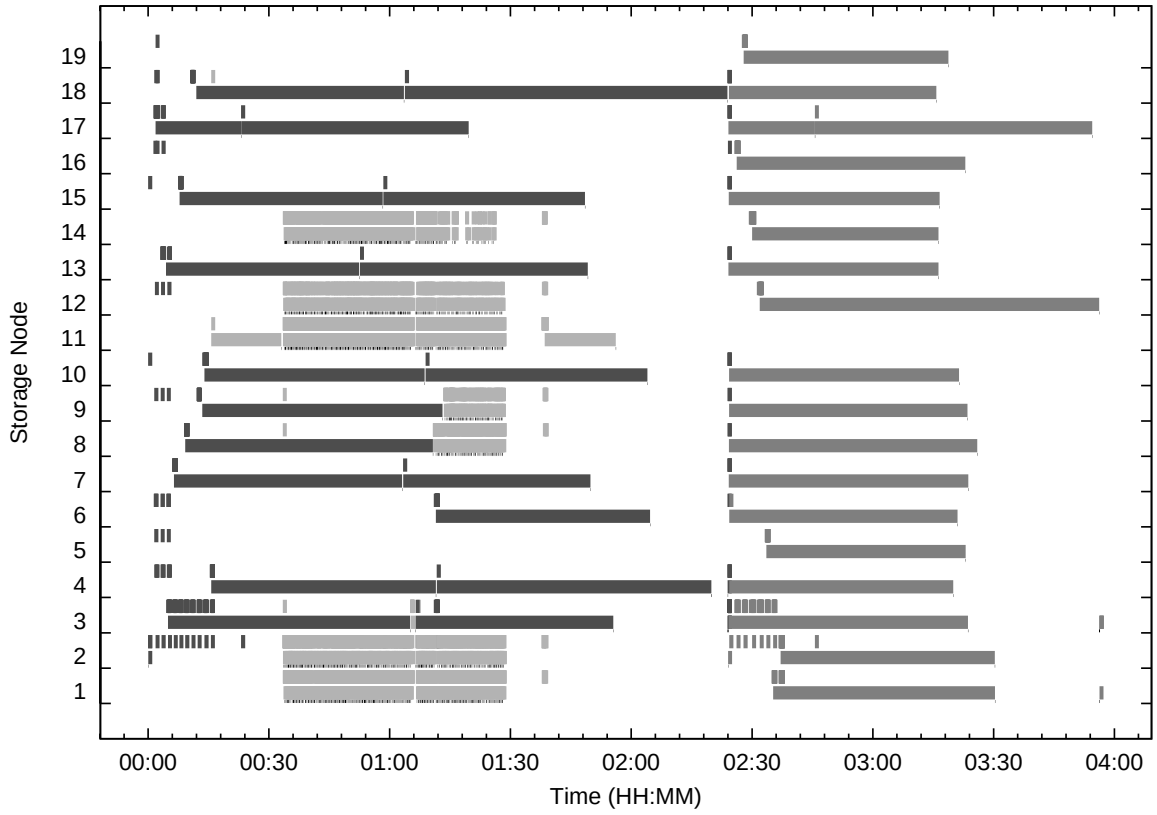
These graphs visualize the activity at each storage node during a run of workflows. Each storage node row has two bars for activity: the top bar (small tics) shows transfers in or out and the bottom bar (long tics) shows job execution. The width of the bar indicates the duration of the activity. Each transfer or job also has a minimum width of 30 seconds, to ease visibility. Additionally, dots below the job execution bar show when a job has finished to distinguish consecutive jobs.

The top graph has two sequential runs of the same BLAST workflow, each run colored differently. Transfers are also colored according to the workflow that initiated them. The BLAST workflow has a shared input dataset composed of multiple files, totaling 8GB. This graph demonstrates that Confuga benefits from previous work replicating files by starting the second workflow's jobs immediately on all storage nodes.

The bottom graph also has two sequential runs of the same BLAST workflow but additionally has a BWA workflow (light gray) running concurrently with the first BLAST workflow. The BWA workflow has an input data set of 32GB which is split 274 times. You can see the split done by SN_{11} at 00:15. This graph demonstrates that Confuga is able to run two workflows with disjoint data sets concurrently with data locality and without significantly displacing each other.

© 2015 IEEE.


Storage Node
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
00:00
00:30
01:00
01:30
02:00
02:30
03:00
Time (HH:MM)


Storage Node
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
00:00
00:30
01:00
01:30
02:00
02:30
03:00
03:30
04:00
Time (HH:MM)

3.8 Conclusion

This chapter presented the design of the Confuga active storage cluster file system. Confuga shares the perspective of other big data systems like Hadoop that effective data-locality is achieved by joining the batch system and the file system: the file system is able to respond to data demands through replication and place jobs according to data-locality. We have shown that Confuga is able to schedule jobs with full data locality and manage transfers with semantics compatible with scientific workflows.

The next chapters will explore scalable metadata access, load stability, and transfer management in Confuga.

CHAPTER 4

METADATA MANAGEMENT

In the previous chapter, we developed the design of the Confuga cluster file system for scientific workflows. From the beginning, Confuga was made to take advantage of the structural information and semantics of DAG-structured workflows to achieve metadata scalability. In this chapter, we will first discuss namespace management in workflow managers. As we will see, traditional approaches to linking the workflow manager with the workflow data-set either are unscalable or require unreasonable privileges and agreement on the workflow namespace. I will discuss how the Makeflow workflow manager was adapted to work with an active storage batch system where these conventional methods of accessing the workflow data are not available. I will conclude with a discussion of how Confuga utilizes the namespace information communicated by the workflow manager to achieve scalable metadata management.

4.1 Namespace in Workflow Managers

The scalability of metadata is linked to the management of namespaces. In systems with traditional distributed POSIX file systems, how metadata scales is closely related to the namespaces in which processes operate. This is well explored in file systems like Ceph [88] which use dynamic sub-tree partitioning to improve the spatial locality of metadata. This is done by separating the global logical namespace into multiple smaller physical namespaces maintained by metadata servers. Processes which access files within the same physical namespace enjoy benefits of spatial locality and lock-free metadata access.

We will see in this section how workflow managers function and how the workflow namespace is defined. Then, we will take a look at how our workflow manager, Makeflow, was enhanced to support active storage computation on Confuga.

4.1.1 Makeflow

The Makeflow [3] workflow manager builds on the venerable Make syntax for expressing job dependencies, which creates an implicit job execution order. Users of Makeflow seek to to unify the specification of jobs to be executed on some distributed batch execution platform, such as Condor [49], SGE [32], or Work Queue [14]. This abstraction allows for trivial parallelism of normal UNIX programs by establishing data dependencies. The Makeflow runtime also handles various details including the handling of workflow fault-tolerance, execution statistics, task ordering and dependencies.

Figure 4.1 shows an example Makeflow file for a bioinformatics workflow[1]. Like Make, the Makeflow specification abstracts jobs into *recipes*. Each recipe includes a *rule* shell command that takes input file *dependencies* and produces some *target* output files. In other words, the recipe defines how to create the targets. Makeflow organizes these recipes into a dynamic acyclic graph (DAG) in order to produce all the desired output files. The engine will then submit the command (rule), list of input files (dependencies), and the expected output files (targets) to the underlying distributed batch system specified by the user in the Makeflow command invocation.

Prior to this work, Makeflow operated under the assumption that all of the workflow files are located on the local file system and that the batch system would return output files for inspection by Makeflow to check for existence and non-empty outputs. So, if the workflow indicates a dependency `a`, Makeflow will execute the `stat` system

[1]Normally, users do not write Makeflows manually. Instead, Weaver [15] is a common approach to *compiling* workflows to the Makeflow language for execution.

```
input.0 input.1: input job.sched gen_submit_file_split_inputs.pl
    ./gen_submit_file_split_inputs.pl 3909

COMMON_INPUTS = blastx legacy_blast.pl distributed.script job.params blast_database/

output.0 error.0 total.0: input.0 $(COMMON_INPUTS)
    ./distributed.script 0

output.1 error.1 total.1: input.1 $(COMMON_INPUTS)
    ./distributed.script 1
```

Listing 4.1: Sample of a Bioinformatics Workflow

call to test for its existence. When Makeflow submits a new job to the batch engine, it uses this same file name `a` for the job. The batch engine is responsible for sending the file to the job, and as we will see, there are a number of ways to do this.

4.1.2 File Access

Due to the distributed nature of the complete workflow system, scientific workflow managers and the jobs executed must have some way to agree on the state and location of files accessed by both. The simplest mechanism to achieve this is to have the workflow manager work on a local file system tree. This serves as the authoritative copy of the workflow data. It is also easily accessed by the user after workflow completion. Jobs will receive a copy of a defined sub-tree to work on during execution. I refer to this as a **local namespace** Both Makeflow and Pegasus [21] work in this way.

On the other hand, with easily deployed distributed file systems like AFS [39], Ceph [88], Panasas [54], and PVFS [69], it can be simpler for the workflow manager and the jobs to use a common namespace for file access, with the underlying file system mounted in the same way. In this case, dependencies are used only for ordering the execution of jobs. Jobs access files normally without assistance from the workflow manager or the distributed batch execution system. Some job dependencies may also be implicit (i.e. unspecified) because the workflow manager need not consider the dependencies for job ordering. In this situation, the workflow manager may need to

resolve name conflicts within the global namespace as some jobs may require static file names as input. Joining the jobs' namespaces in the global namespace would cause failures and so requires special handling[2]. I refer to this scenario as a **global namespace**.

Up to this point, Makeflow was written to (mostly) seamlessly operate within either a global or local namespace. For a local namespace, the workflow manager manipulates workflow files locally and assumes the batch system will distribute files as necessary. For a global namespace, the workflow manager is executed in the appropriate location (`chdir`) in the global namespace so names resolve. Or, the workflow files are written with absolute path names to the common global namespace (e.g. `/afs/nd.edu/user31/pdonnel3/workflow`). The batch system (e.g. SGE [32]) does not move the files to the workers.

4.1.3 Bound Namespaces

In a conventional multi-process setting, all processes operate within a single namespace. This allows trivial communication and instant visibility of changes to the file system. In a distributed batch execution system, the namespaces are often physically and logically separate. The main challenge in these environments is how to access these namespaces and bind required files to the execution environment. The normal approach to solving this problem is one of avoidance: use a global namespace that is accessible by jobs and the workflow manager.

However, requiring the use of a global namespace to execute workflows with data located on a distributed file system is not always feasible. Users may not have privileges to mount a global namespace on their workstation, let alone all of the machines used to execute jobs. Furthermore, the use of a global namespace enables the user to skip the listing of certain dependencies which do not impact job ordering.

[2]Of course, ideally programs would not be written to require static file names.

This impacts the potential for data locality as the batch system cannot make locality determinations if it lacks the full picture.

This was resolved by the development of a new namespace type, a **bound namespace**. This namespace type made the joining of the *workflow namespace* and the *job namespace* explicit through a *binding* of file names between the namespaces.

So now the dependency list is used to construct a mapping between the workflow and job namespaces, in addition to job ordering failure detection. For many workflows, this mapping is *trivial* as jobs are frequently written as though executed with access to the workflow namespace. For example, Listing 4.2 shows a rule from a bioinformatics workflow written with the job executing relative to the root of the workflow namespace.

```
1 workflow/00000001.out: workflow/bio-kernel.py
      workflow/Mosquito.FilteredPacBioSubreads.58Cells.130905.fastq
      workflow/genome.00000001 workflow/_Stash/0/0/0/0000001
2     workflow/_Stash/0/0/0/0000001 workflow/bio-kernel.py workflow/00000001.out
          workflow/Mosquito.FilteredPacBioSubreads.58Cells.130905.fastq
          workflow/genome.00000001
```

Listing 4.2: Single Rule from the IALR Bioinformatics Workflow

For example, the dependency `workflow/_Stash/0/0/0/0000001` should be mapped into the job sandbox as `workflow/_Stash/0/0/0/0000001`. Unfortunately, support for this trivial mapping is not widespread among batch systems. For example, Condor is well-known for mapping job input files into a flat sandbox in the job. For example, `workflow/_Stash/0/0/0/0000001` would be mapped to `0000001` in the job sandbox. Users planning to execute workflow on Condor must be write jobs to operate in a flat sandbox and implicitly rely on Condor's namespace mapping.

The three namespace types are shown in Figure 4.1.

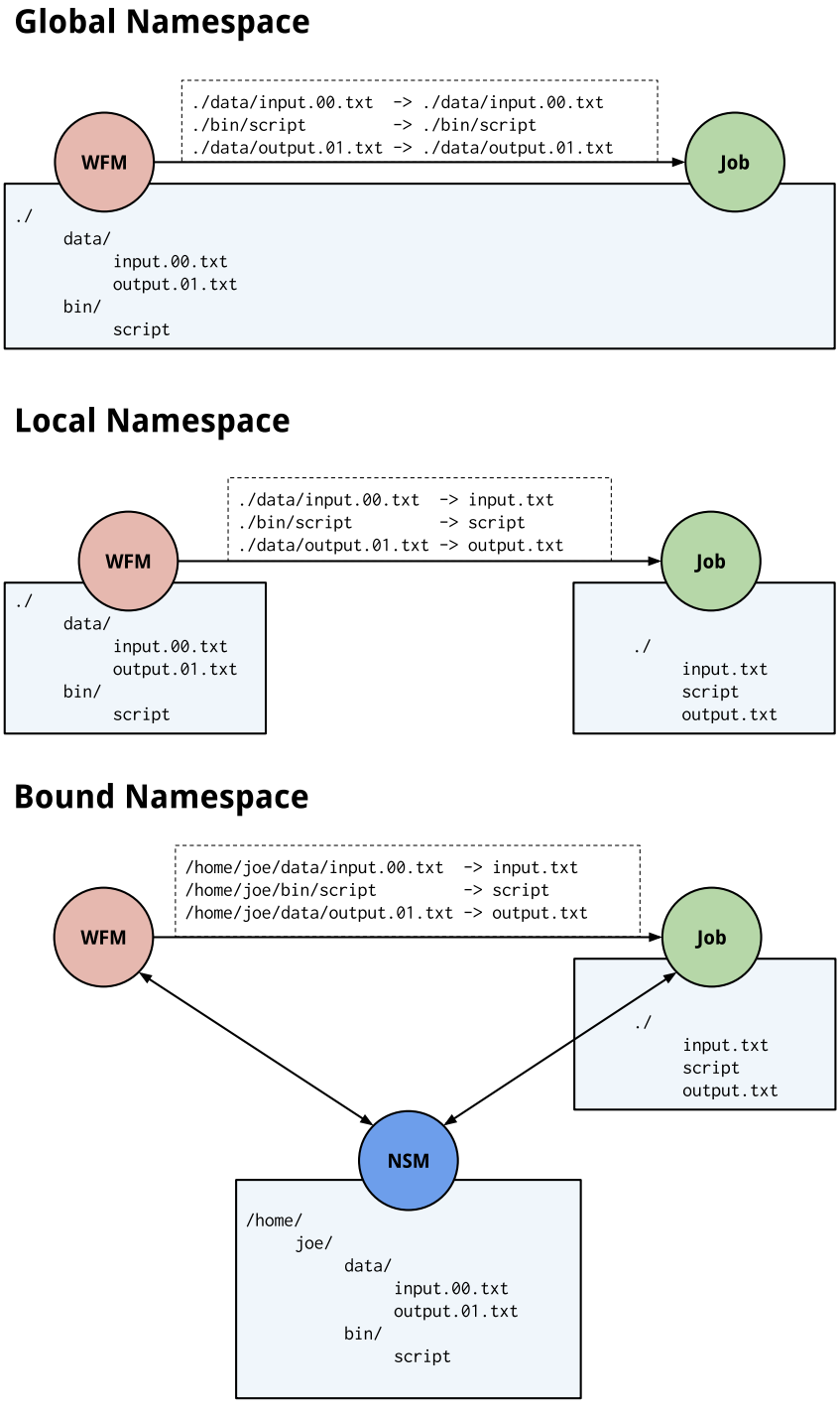


Figure 4.1. Distributed Execution System Namespaces

As we will soon see, explicitly defining the **workflow namespace** and **job namespace** allows the workflow manager and batch system to trivially manipulate and transform the workflow namespace as needed without disrupting job file access. Additionally, formally defining the workflow namespace allows for relocatable workflows. This examined next in the extensions to Makeflow.

4.1.4 Extending Makeflow for Active Storage

Workflow managers require access to files in the workflow namespace for a number of reasons: checking input file dependencies exist before dispatching a job, confirming job outputs are created on completion, garbage collecting unneeded intermediate files, wrapping job executables with diagnostic or monitoring code, etc. In the context of Confuga, Makeflow must be able to manipulate workflow files which are on the storage cluster and not locally accessible.

In early versions of Makeflow, all file access assumed a local or global namespace where Makeflow must simply be executed relative to the workflow namespace (whether in `/panasas/home/j/workflow` or `/workflow`). File operations such as `stat` or `unlink` were executed as normal system calls handled by the kernel. For the batch systems that Makeflow supported at the time, this setup was sufficient. However, when the workflow namespace is not locally mounted but only available through remote RPC on a cluster, Makeflow must be extended to operate on an explicitly defined bound namespace.

Makeflow uses a *batch library* which is used to multiplex between different batch execution platforms like Condor [49, 82], SGE [32], and Work Queue [14][3]. It is responsible for scheduling and managing the job executions and files on the underlying resources. This common abstraction is widely adopted [14, 32, 82] to mask the

[3]For example, the Makeflow command-line switch `-Tcondor` configures the batch execution library to use the Condor job driver.

underlying complexities of the batch systems. Prior to this work, Makeflow used this batch library to only submit jobs with a list of input and output files. Makeflow and the batch library would separately access these files for different purposes: Makeflow checks for file existence for pending jobs and the batch library *may* check these files to perform necessary file transfers to the remote site.

In order to support active storage where the workflow namespace is not locally mounted, it is necessary to move namespace management access at the batch library layer[4]. This allows Makeflow to operate on the namespace without knowing where workflow files are located. Figure 4.2 visualizes the transformation in workflow file access.

The changes to Makeflow and the batch execution library were not particularly onerous. Only 6 file system operations were identified in Makeflow that required equivalent functions in the batch execution library. So for example, instead of using `stat("bin/blast")` to determine the existence of the input file for a job, Makeflow uses `batch_fs_stat(Queue, "bin/blast")`. These operations are shown in Table 4.1. The table shows at which stage in the workflow the operation is executed.

Finally, it was necessary to allow the user to specify a location for the workflow namespace. For conventional batch systems, this is the working directory of Makeflow. The Makeflow specification will refer to paths relative or absolute to the working directory of Makeflow. For active storage, the workflow namespace is configured to be the workflow path on the remote storage for the batch execution system. So with a workflow namespace of `"/home/john/workflow1/"`, `"bin/blast"` resolves to `"/home/john/workflow1/bin/blast"`. This resolution happens in the batch library layer; Makeflow continues to operate with paths as written in the Makeflow specification file. This makes each Makeflow specification **relocatable** in the sense

[4]This functionality was added to Makeflow in commit `7aa72f27209fcac122fab6521f3dc64db871c91f`.

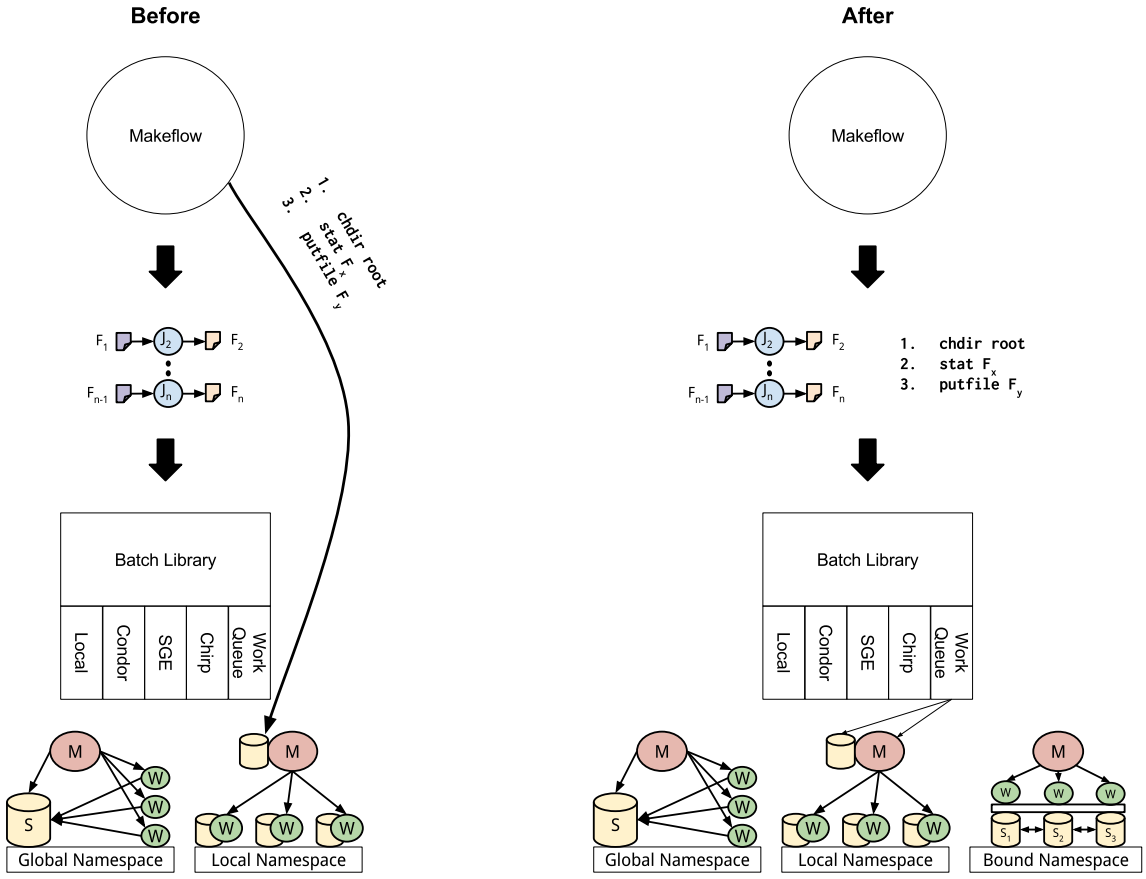


Figure 4.2. Makeflow and Batch Job Library with and without I/O Extension

TABLE 4.1

BATCH JOB FILE OPERATIONS

Operation	**Stage**
chdir	Workflow Boot
getcwd	Workflow Boot
mkdir	Job Setup
putfile	Job Setup
stat	Job Setup, Job Finish
unlink	Job Finish

that the same workflow may be executed in different locations so long as the initial dependencies are available in the new workflow namespace. This is a useful property for workflow sharing and reproducibility.

4.2 Developing a Job Framework for Active Storage

In addition to abstracting the workflow namespace access in Confuga, it was necessary to incorporate the use of bound namespaces into a job framework usable for active storage. The framework was designed with an eye towards Confuga but was implemented first for the Chirp [81] distributed file system[5], a file system service similar to NFS [73] which is designed to be deployed on a grid to make available data for remote access. Chirp is attractive for deployment on grids and clusters for a number of reasons including trivial user deployment without administrator privileges, strong and usable authentication and access control mechanisms, and multiple mechanisms for applications access.

[5]Confuga is overlaid on Chirp and supports the same job protocol.

To be clear, the active storage we are developing for is different from the original description. The first papers on active storage began as smart disks [66] and grew within the HPC community to smart object stores which can harness unused CPU to perform simple functions on data [60] with the goal of increasing I/O throughput and reducing data movement [75]. More recently, projects like Hadoop were developed for clusters built on commodity hardware that are dedicated to performing structured computation on large datasets. This framework is designed to support the latter use-case: ship executables to storage sites for execution on large data.

Chapter 3 introduced the Chirp job protocol used by Confuga in Section 3.3. Here we will review the namespace binding aspects of the protocol.

4.2.1 Creating a Job

Jobs are created in Chirp using two-phase commit with the `job_create` and `job_commit` RPC. Listing 3.6 on page 30 showed an example job specification. Listing 4.3 shows a reproduction of its file array.

The `files` array provides the namespace binding for the job. The `task_path` corresponds to the location in the sandbox of the job. The `serv_path` is the location of the file on the server. The `type` indicates if the file is an input or an output. Each file is bound into the sandbox using one of a number of binding methods. By default, files are hard linked (`link(3)` in UNIX) into the sandbox of the job. File copies are also supported when the job needs to modify inputs but the cost is proportional to the file size.

The careful reader will notice that the executable `/bin/sh` is not included in the namespace binding for the job. Normally Makeflow dictates that the user includes *all* dependencies for each job but some may still go unspecified. System utilities and libraries are generally available so the user can "get away" with leaving them out.

```
"files": [
  {
    "task_path": "input",
    "serv_path": "/u/patrick/input.1",
    "type": "INPUT"
  },
  {
    "task_path": "output",
    "serv_path": "/u/patrick/output.1",
    "type": "OUTPUT"
  }
]
```

Listing 4.3: Chirp Job File Binding

```
"files":[
   {
      "binding":"LINK",
      "serv_path":"/u/patrick/input.1",
      "size":null,
      "tag":null,
      "task_path":"input",
      "type":"INPUT"
   },
   {
      "binding":"LINK",
      "serv_path":"/u/patrick/output.1",
      "size":551,
      "tag":null,
      "task_path":"output",
      "type":"OUTPUT"
   }
]
```

Listing 4.4: Job Status Following Completion via Wait

4.2.2 Waiting for a Job

Clients wait for jobs in Chirp using the `job_wait` and `job_reap` RPC. Waiting for a job returns the complete job status information detailing all job metadata. Listing 3.7 on page 31 shows example job status return from `job_wait`. Listing 4.4 shows a reproduction of the file array component of the job status.

After waiting for a job which has finished, the output files have been mapped into the server namespace. In the status returned by `job_wait`, each output file includes its size and the finalized name in `serv_path`. This finalized name is discussed next in Section 4.2.3.

TABLE 4.2

JOB OUTPUT FILE NAME INTERPOLATION

Sigil	Replacement
`%g`	A 20 byte UUID.
`%h`	The SHA1 hash of the file content.
`%j`	The job's ID.
`%s`	Like %h if file size < threshold, otherwise %g.

4.2.3 Output File Name Binding

The name for an output file of a job is not always static. It can be necessary to compute the name dynamically after the job has completed. For example, if the file is to placed in a Content Address Store (CAS), it is desirable to name it according to its hash. The Chirp job protocol allows this dynamic naming of an output file via string interpolation of the `serv_path`. The sigils supported are shown in Table 4.2.

The variety of sigils supported were a result of an evolution in requirements for Confuga. The jobs sent to storage nodes originally would store files in a CAS directory using a `serv_path` like `".../file/%h"`. After more extensive tests were performed, it became obvious that always hashing the file content resulted in significant slow downs when a job produces a large output file. The reason for this is that the storage node must hash the entire output file once the job completes. If the file is large, this means the entire file must be read from disk.

This cost was determined to be unacceptable. The benefits of deduplication and deterministic naming are not justified. Instead, the use of Universally Unique Identifiers (UUIDs) were incorporated to give new output files names. These identifiers can

be generated using the `%g` sigil. It was observed however that both techniques could be used depending on the file size. Confuga uses the `%h` sigil which allows hashing files with size less than a threshold, currently 16MB. This technique of deduplicating small files is similar to the interned "short string" table in the Lua programming language version 5.2 [40]. Lua conditionally interns strings with size below a threshold. The rationale is the same in Lua: long strings are unlikely to be duplicates and the cost of producing a useful hash is large[6].

4.2.4 Challenges

Moving the workflow namespace outside of the submission machine into an external store exposed problems for some workflows. For example, many archived Makeflow experiments experienced problems due to "local recipes". These are recipes that are to be executed by Makeflow at the submission site. The general intent is to perform minor or bottleneck work at the submission site rather than dispatching a job to the execution engine. For example, it is common for an early job to split a large input file and subsequent jobs to depend on single splits. It can be beneficial to just perform the split in Makeflow as nothing is gained by moving the task to a remote execution node.

Local recipes were problematic for two reasons:

- Local recipes only make sense when the submission site running Makeflow has direct access to the workflow namespace on a locally mounted file system. When executing jobs for active storage, this is not the case! These jobs must be executed as normal jobs in order to access files in the workflow namespace. To fix this, Makeflow was modified to conditionally submit local jobs to the remote batch queue when Makeflow does not have direct access to the workflow namespace. Because of this change, the `LOCAL` qualifier on recipes became a suggestion rather than a requirement.
- Local recipes allowed unnamed dependencies in the workflow namespace. Since

[6]Lua was also confronting the problem of hash collisions in malicious strings causing denial-of-service attacks on table access.

a local job would be run with access to the workflow namespace, any dependencies which are available but not named in the dependency list would cause no problems. Had the job been remote with for example Condor, the unnamed dependency would not have been sent along with the job and resulted in a runtime error by the job executable. Local jobs however have access to the full workflow namespace and can thus access any file that exists, even if it was not listed in its dependencies.

When using a bound namespace and active storage, these (formerly) local jobs no longer have full visibility of the workflow namespace! As stated in the previous point, local jobs are executed as regular jobs. So all dependencies must be bound into its job namespace or they will not be available. Many workflows had to be corrected to include these missing dependencies in the local recipes.

4.3 Confuga Metadata Scalability

Up to now, we have outlined the changes to necessary to make the Makeflow scientific workflow manager work within an active storage environment. Here we will examine how Confuga leverages the information that Makeflow gives the batch execution system to scale metadata access for the file system.

Scalable metadata access is a persistent problem in distributed file systems which provide a global namespace for file metadata and location. POSIX consistency semantics in a distributed setting have significantly contributed to this problem. AFS [39] is well known for relaxing certain requirements to meet scale and performance needs. In particular, AFS introduced *write-on-close* so other clients will not see changes to a file until the writer closes the file. Doing otherwise would require propagating the update to the file's blocks and size to the file server before any write may complete.

Confuga minimizes load on the head node by exploiting the structural information available in the workflow. Specifically, Confuga relies on the complete description of the input and output files available from the DAG workflow manager. Using this information, Confuga is able to perform all metadata operations, including access control checks and determining replica locations, prior to job dispatch to a storage

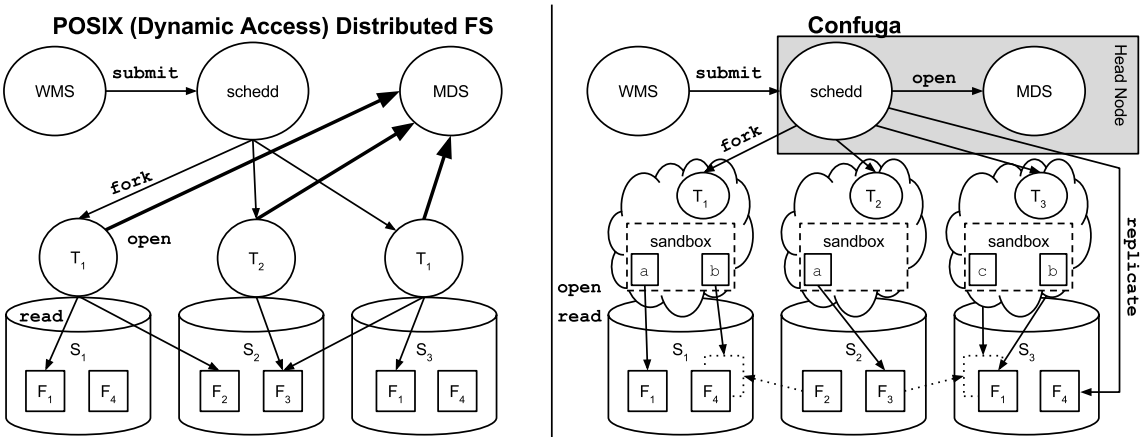


Figure 4.3. Distributed File System Metadata Access Patterns

Large cardinality in bold arrows.
© 2015 IEEE.

node. The job description sent to the storage node contains a complete task namespace mapping to immutable replicas local to the storage node. The storage node requires no further information to execute the job. Figure 4.3 visualizes the different models for handling metadata operations for Confuga and traditional POSIX distributed file systems.

4.3.1 Namespace Remapping

The full description of the job namespace forms the foundation for the design and optimizations of Confuga. In Confuga, this description is provided in the form of a **namespace mapping** of the job's namespace or sandbox to the workflow namespace (a bound namespace). In Confuga, this workflow namespace would be a sub-tree of the global cluster file system namespace.

Prior to scheduling a job, Confuga performs access control checks for each input and output. These access controls are maintained per-directory. Once access control checks are complete, Confuga will **bind** each input file by looking up its replica identifier. Input files that are directories expand recursively to an equivalent input file

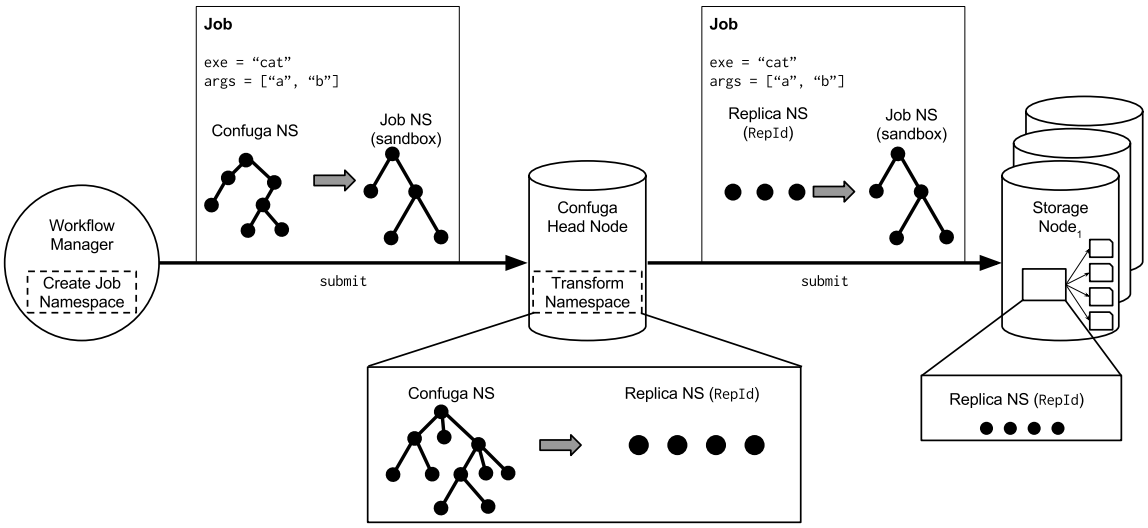


Figure 4.4. Confuga Job Namespace Remapping

list. Replicas in Confuga are whole-files and immutable, so this operation effectively causes each job to atomically read its inputs from the global namespace prior to execution. Binding each input file to a replica is done through namespace remapping, as shown in Figure 4.4. Each input file in the sandbox of the job is remapped to its corresponding replica in the flat replica namespace. Similarly when a job completes, Confuga learns the replica identifiers for each of the job's output files and atomically updates the global Confuga file system namespace. Directories as output files is not permitted.

What this amounts to is a layering of consistency semantics. Individual jobs execute within a namespace (sandbox) with normal POSIX consistency semantics. All operations within its namespace are native accesses; the cluster file system does not introduce any remote file operations or new mount points. At the level of the workflow, consistency is maintained at job boundaries, start and end. I refer to these semantics as `read-on-exec` and `write-on-exit`. Restricting consistency semantics in this way has a number of advantages:

1. Because each job includes its full list of data dependencies, Confuga is able to make smarter decisions about where, and most importantly when, to schedule a job to achieve complete data-locality and to control load on the network.

2. Since all input files can only be pulled from the workflow namespace before the job begins execution, Confuga can eliminate dynamic/remote file operations by a job. This is an essential aspect of Confuga's execution model as it allows for controlling load on the network: a job cannot dynamically open files on the cluster which would introduce network overhead out of the head node's control.

3. We are able to reduce metadata load on the head node by isolating storage nodes from the global cluster namespace. Any metadata required for a job and its files can be sent as part of job dispatch and retrieved when reaping the job. In fact, storage nodes in Confuga are treated as dumb active storage file systems, completely unaware of the global context.

4.3.2 Caveats

Restricting a system to a certain execution model usually has drawbacks; Confuga is no different in this regard. While the model is flexible enough to run any type of workflow expressed as a DAG of jobs, it will not perform optimally when the input files are largely unused. Either because there are input files which are never opened by the job or because a large input file is not fully utilized. Confuga also requires that all dependencies are fully replicated on the storage node chosen for a job's execution. This means that all of a job's files must fit on disk. This requirement encourages structuring a large data set as multiple files, which is already a de facto requirement for DAG workflows. Our experience suggests this is not a significant burden on users.

It is worth noting that using workflow information in this way to optimize metadata operations is not uncommon in cluster file systems. For example, because Hadoop's MapReduce implementation knows which blocks a Map task will work on, it can minimize future work by isolating the Map task from the file system by looking up file metadata and replicas for the Mapper and forwarding data blocks directly to the Map function. Naturally, not following the MapReduce model results in performance penalties, e.g. by opening numerous other data files in HDFS. Ad-

ditionally, Hadoop also relies on HDFS's immutable file blocks to reduce (eliminate) consistency concerns.

Confuga also relies on each job operating atomically, without consistency updates from the head node. This is a common constraint in DAG-structured workflow managers that synchronize and order tasks through output files. When jobs would run on a master/worker framework or on a cycle scavenging campus grid, it was not useful to allow network communication between jobs because (a) jobs would need to be able to "find" other jobs, (b) communication must get through firewalls between subnets, and (c) communication would introduce dependencies between jobs unexpressed in the workflow description. Confuga takes advantage of the DAG workflow model by eliminating consistency checks and dynamic file access for jobs.

4.4 Performance

Evaluating the effectiveness of Confuga's metadata management for scalability requires tracking the number of metadata operations made by jobs during the course of the workflow. This allows us to quantify the number of metadata operations made by each job. As a way of comparison, we also can track the metadata operations made by the head node in the course of executing jobs.

I have used the `strace` utility to monitor the I/O system calls made by jobs running in the cluster. These I/O system calls will execute within the sandbox of the job on the storage node. They do not result in communication with the head node. The collective system calls can be used to quantify the amount of RPC traffic Confuga avoided by batching operations prior to submitting jobs.

The Confuga head node metadata operations are shown in Table 4.3. The scheduler binds all input files for each job prior to scheduling. This involves a `lookup` of the replica identifier for each file dependency. When a directory is named, then it is recursively broken down into an equivalent mapping of regular files, each of which is

TABLE 4.3

HEAD NODE METADATA OPERATIONS

Operation	Stage	Purpose
`lookup`	Job Input Binding	Lookup Replica ID for given file name.
`opendir`	Job Input Binding	Open a directory for reading.
`readdir`	Job Input Binding	Read single entry from directory.
`update`	Job Output Binding	Update the Replica ID for the given file name.

also requires a replica lookup. Likewise, when a job ends, each output file's replica identifier is written to the global namespace, the `update` operation.

We have used two unmodified bioinformatics workflows, BLAST and BWA, for evaluating the effectiveness of Confuga's metadata scaling. The BLAST workflow is composed of 24 jobs with a shared 8.5GB database. It is used for comparing a genomic sequence query against a reference database, yielding similar sequences meeting some threshold. The BWA workflow performs a genomic alignment between a reference genome and a set of queries. The BWA workflow is composed of 1432 jobs, starting with a 274 way split of the 32GB query genome. The purpose of this alignment is to later compare how well the genomes align and where in the reference genome they align.

The two bioinformatics workflows, BLAST and BWA, are shown in Tables 4.4 and 4.5. The tables show operations in three contexts: **"Scheduler → MDS"** operations performed by the scheduler on the global namespace (all local on the head node); **"Scheduler → Storage Node"** operations by the head node on storage nodes; **"Storage Node → File System"** operations by storage node jobs on its sandbox ("Sandbox") and local system utilities ("All"). (So, "All" includes opera-

tions on the "Sandbox" and other file system operations done on the entire storage node namespace including local system files, libraries, and executables.)

Because each BLAST job (24 jobs) depends on a database of files contained in a directory, the scheduler must resolve this directory to a number of equivalent input files which are reconstructed as the directory for each job by the storage node. For this reason, the scheduler performs several `opendir` and `readdir` calls for the database directory. For each job, there were an average of 74 metadata operations the head node needed to perform on the global namespace. On the other hand, each job performed on average 293 metadata operations. This confirms that the head node is able to avoid numerous metadata operations by pre-batching operations prior to job dispatch.

The BWA workflow (1432 jobs) is notable for being very streamlined with the I/O in its sandbox. There are approximately 10 sandbox `open` system calls for each job but only 1 `stat` for the entire workflow. For BWA, the ratio of `lookup+update` to `open+stat` is not as favorable as BLAST because each job would dynamically open up only some of its input files. This result highlights that metadata operations in Confuga are the result of the workflow description and not the job. [For this BWA workflow, the disparity may indicate a problem with the workflow description where jobs include more dependencies than necessary. Or, each job dtynamically selects files to open at run-time.]

It's worth pointing out that there is significant traffic outside the sandbox of each job. Ideally workflows use only files within its sandbox. The reasons for this are intuitive: implicit dependencies can cause unexpected failures as workflow execution depends on inputs defined by the execution site. These implicit dependencies may vary significantly between machines and impact the performance, correctness, or stability of the workflow. There has been effort to contain workflows and package all dependencies [15] but change is slow. For Confuga's part, it does not prevent appli-

cations from accessing utilities outside the sandbox. [In fact, completely restricting the namespace to the sandbox is challenging and may require root. For example, applications regularly use `/dev/null` and `/proc`. Properly restricting these system files requires privileges that Confuga does normally require.]

TABLE 4.4

METADATA AND HEAD NODE OPERATIONS FOR BLAST WORKFLOW (24 JOBS)

Scheduler → MDS		**Scheduler → Storage Node**				**Storage Node → File System**			
Metadata	**Count**	**Job**	**Count**	**Transfers**	**Count**	**All**	**Count**	**Sandbox**	**Count**
lookup	881	job_wait	440679	thirdput	618	stat	14085	stat	4687
readdir	779	job_create	30	rename	597	open	3648	open	1407
update	104	job_commit	26	access	597	access	436	readlink	361
opendir	19	job_reap	24	-	-	readlink	400	getcwd	247
-	-	job_kill	0	-	-	getcwd	247	getdents	14
-	-	-	-	-	-	getdents	14	-	-
-	-	-	-	-	-	statfs	7	-	-

TABLE 4.5

METADATA AND HEAD NODE OPERATIONS FOR BWA WORKFLOW (1432 JOBS)

Scheduler → MDS		**Scheduler → Storage Node**				**Storage Node → File System**			
Metadata	**Count**	**Job**	**Count**	**Transfers**	**Count**	**All**	**Count**	**Sandbox**	**Count**
lookup	13170	job_wait	90672	access	2350	open	17928	open	10731
update	3578	job_create	1433	thirdput	1452	access	1432	stat	1
readdir	0	job_reap	1432	rename	1448	stat	21	-	-
opendir	0	job_commit	1432	-	-	readlink	2	-	-
-	-	job_kill	0	-	-	-	-	-	-

4.5 Conclusion

This chapter presented the namespace management used in scientific workflow managers and how to adapt workflows for active storage on Confuga. Confuga is also shown to manage metadata operations in a scalable manner by exploiting workflow semantics of *read-on-exec* and *write-on-exit*. All metadata operations are batched and executed prior to and after a job's execution.

While these semantics are very effective when the workflow structure permits these restrictions, they are not applicable for workflows which operate on shared files. For example, HPC workflows often rely on a shared store such as GPFS [74] for consistent editing of files. However, many environments have embraced the model we have discussed where jobs are atomic and operate without a shared file namespace or communication mechanism (other than output files). For these cases, Confuga is an appropriate match which can safely eliminate most metadata operations which impede scalability.

It is suggested that workflows should be written with strict adherence to dependencies. While it can be help the user in the immediate situation to be forgiving when a workflow is incomplete, future adaptations or applications of the workflow may fail on other systems. This is particularly troublesome when looked at from a perspective of preservation. When archived workflows were applied to our active storage batch system, missing dependencies caused the workflow to unexpectedly fail. It is an ongoing subject of research in the Cooperative Computing Lab is to enforce the scope and explicitness of dependencies to address this issue.

CHAPTER 5

DIRECTED TRANSFERS

5.1 Introduction

So far, I have presented the general design and metadata scalability of Confuga. This chapter will present the use of directed transfers in Confuga to control transfer load on the cluster.

Typical cluster file systems present a POSIX interface that permit dynamic and uncontrolled access to files stored on cluster nodes. An application program *opens* a file and *reads* or *writes* an unknown quantity of data. Distributed file systems have little information in how applications will behave and must robust performance to all access patterns. For example, popular cluster file systems like GPFS [74], Ceph [88], Lustre [60], and Panasas [54] all operate in this way.

Confuga uses directed transfers to address **load instability** in the cluster. These are transfers which are planned and managed by a central authority (e.g. the head node). This makes data distribution in Confuga an oddity relative to other popular distributed file systems. Usually, the file system does not have any information about metadata or data requests. In fact, the usual way to study file systems is to observe how they react to unknown workloads [6, 67, 68].

The goal of directed transfers is to eliminate load instability and improve transfer performance. As workloads scale up, and the number of simultaneous users increase, it is all too easy for concurrent transfers and tasks to mutually degrade performance to the point where the entire system suffers non-linear slowdowns, or even outright

task failures. For example, it is frequently observed that too many users running on Hadoop simultaneously will cause mutual failures [94]. Confuga avoids this problem by tracking the load placed on the system by each job and transfer and by performing appropriate load management which results in a stable system.

Confuga receives the information necessary to control transfer load from the workflow manager. The job description lists the files needed and how each file is mapped from the workflow namespace to the job namespace (discussed in Chapter 4. Confuga then uses this information to plan all transfers to the storage node the job is scheduled on and to safely eliminate dynamic transfers during the job's execution.

This chapter explores how Confuga uses directed transfers to replicate job dependencies to storage nodes. These are implemented as **push transfers** which direct a storage node to replicate a file to another. We will also examine mechanisms for managing these transfers in the scheduler and how they may be used to control load and achieve performance.

5.2 Synchronous Push Transfers

The early prototype of Confuga exclusively used a synchronous transfer mechanism for moving replicas around the cluster. The head node makes a transfer request to the *source* storage node directing it to copy the replica to a *target* storage node. This is a *stop-the-world* operation that causes the scheduler to block until the transfer completes. So other work by the scheduler (and other head node operations) is halted.

Synchronous transfers are performed using the Chirp `thirdput` operation [80]. The purpose of thirdput is to avoid moving data through the caller as an intermediary. Instead, data may be moved directly from the source to the target. Figure 5.1 visualizes the transfer.

The early versions of Confuga used synchronous transfers for rapid prototyping

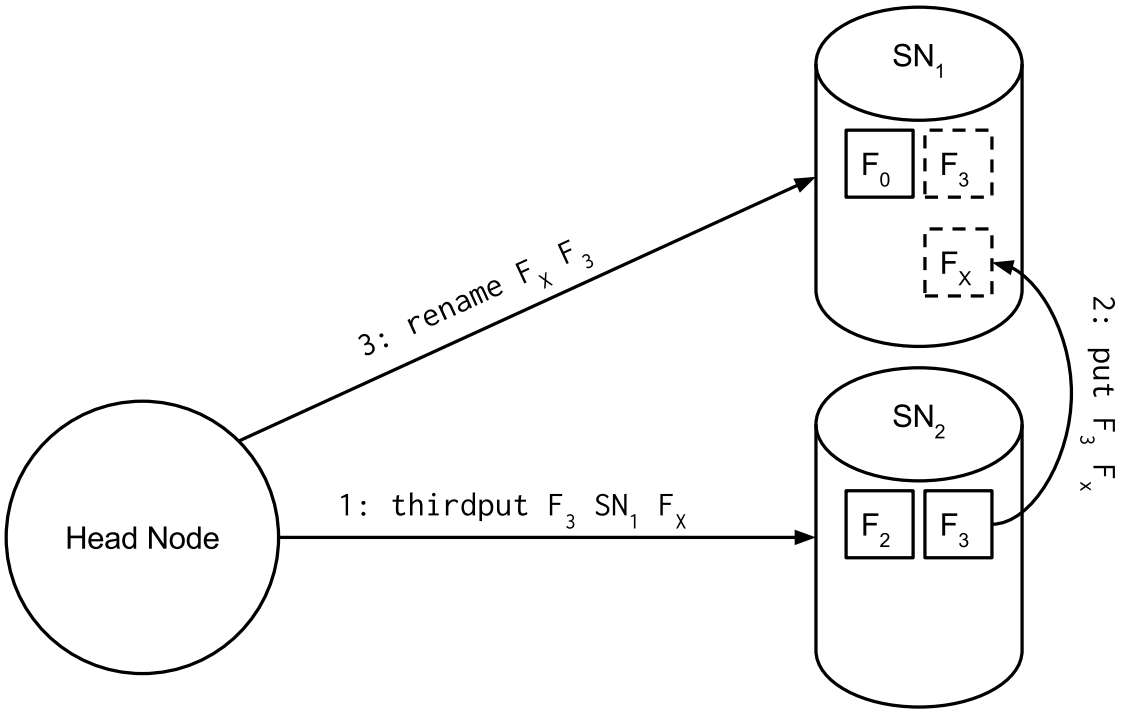


Figure 5.1. Synchronous Transfers in Confuga

These are blocking replication operations performed by the scheduler. The `thirdput` *RPC is used to have a storage node transfer its replica to another storage node.*
© 2015 IEEE.

and then later as a basis of comparison to asynchronous transfers. Because synchronous transfers execute serially and so only one transfer is active at a time, there can be no transfer parallelism. So, the head node essentially limits the cluster to a single transfer at a given time. Naturally, this severely limits scalability and the usefulness of this transfer type. On the other hand, there is no interference between transfers. So there is no other competition for storage node network or disk resources.

In practical situations, synchronous transfers should be avoided. It is possible that for smaller files a `thirdput` operation may actually save time versus scheduling an asynchronous transfer. Even so, scheduling multiple file transfers in a single batched operation would likely be preferable. Using synchronous transfers in this way has not

been examined in this work.

In evaluating this transfer mechanism, I refer to this replication strategy as **sync**.

5.3 Asynchronous Push Transfers

An asynchronous push transfer is a head-node initiated replica transfer from a source storage node to a target storage node. Unlike synchronous transfers, the scheduler may perform other functions during the course of the transfer. This includes scheduling or reaping other asynchronous transfers or jobs.

Confuga implements asynchronous push transfers using **transfer jobs**. These are special jobs executed by storage nodes which execute file system operations within the storage node file system instead of a user application. The head node benefits from this logical extension of jobs through asynchronous execution, code reusability, and reliable creation, tracking, and reaping of transfers within a distributed context. A failed transfer job is handled by the scheduler similarly to regular jobs; the head node will reschedule the transfer if it is a transient failure otherwise pass the fault up to the job which scheduled the transfer.

Transfer jobs execute the `putfile` RPC [81] which copies a given source replica to a temporary file on the target storage node. The head node monitors the job's progress through periodic `job_wait` RPCs on the source storage node and `stat` RPCs of the temporary file on the target storage node. When the transfer completes, the head node will atomically move the temporary file to the replica namespace on the target storage node. This prevents an incomplete replica from being used by other jobs or left behind by a failed transfer. The entire transfer job process is visualized in Figure 5.2.

Authentication for transfer jobs is managed through tickets which are periodically renewed by the head node. The head node sets the ticket up periodically as described in Section 3.4. The use of tickets enforces secure and restricted storage node to storage

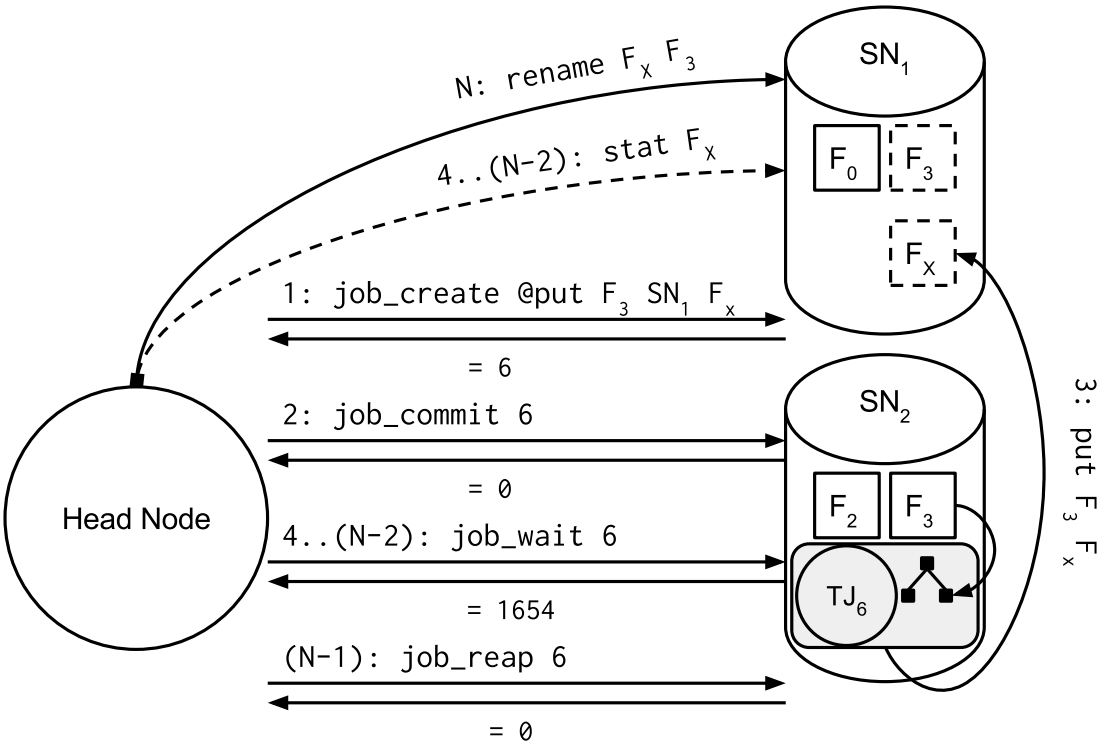


Figure 5.2. Asynchronous Transfers in Confuga

These are a series of short blocking operations that create a **transfer job** *on the storage node. The transfer job executes asynchronously with the head node. Operations are numbered in order.*
© 2015 IEEE.

node authentication and access control. Unlike other distributed cluster file systems, Confuga does not assume exclusive access to storage nodes[1]. For that reason, steps are taken to limit the transfers jobs' access to data and protect cluster data from other users.

Listing 5.1 shows an example push transfer job specification used by Confuga. The special executable `@put` is interpreted by the storage node as a Chirp file system operation. Internally, the storage node job scheduler creates a sub-process that asyn-

[1]More specifically, the Chirp servers operating the storage nodes may be used by other systems and clients.

```
{
    "executable":"@put",
    "tag":"confuga:D79A5D963ABEEB71B01FE2B759C38E3CBDE54F7B",
    "arguments":[
        "@put",
        "localhost:10003",
        "file",
        "/confuga/open/23EC16B19AA5AAF0CC2E92E9BB849F7D"
    ],
    "environment":{
        "CHIRP_CLIENT_TICKETS":"./confuga.ticket"
    },
    "files":[
        {
            "task_path":"file",
            "serv_path":"/confuga/file/74D8F3DB0BDAF92F32B074B6740482A599F3A3E4",
            "type":"INPUT",
            "binding":"LINK"
        },
        {
            "task_path":"./confuga.ticket",
            "serv_path":"/confuga/ticket",
            "type":"INPUT",
            "binding":"LINK"
        },
        {
            "task_path":".chirp.debug",
            "serv_path":"/confuga/job/debug.%j",
            "type":"OUTPUT",
            "binding":"LINK"
        }
    ]
}
```

Listing 5.1: Example Push Transfer

chronously executes (like any other job) the corresponding Chirp client API call. In this case, the job will execute the `putfile` RPC.

Just like other jobs, the transfer job executes within a sandbox. The replica and authentication ticket are mapped into its sandbox as `./file` and `./confuga.ticket`, respectively. The environment variable `CHIRP_CLIENT_TICKETS` is used to indicate which ticket to use for the Chirp operation. Finally, it's worth noting that Confuga saves the transfer job debug information which includes the Chirp debug output. This has been useful on numerous occasions for debugging failed or slow transfers.

5.3.1 Load Control using Transfer Slots

The Confuga head node organizes push transfer load management using **transfer slots**. Each storage node has a configurable number of transfer slots which are

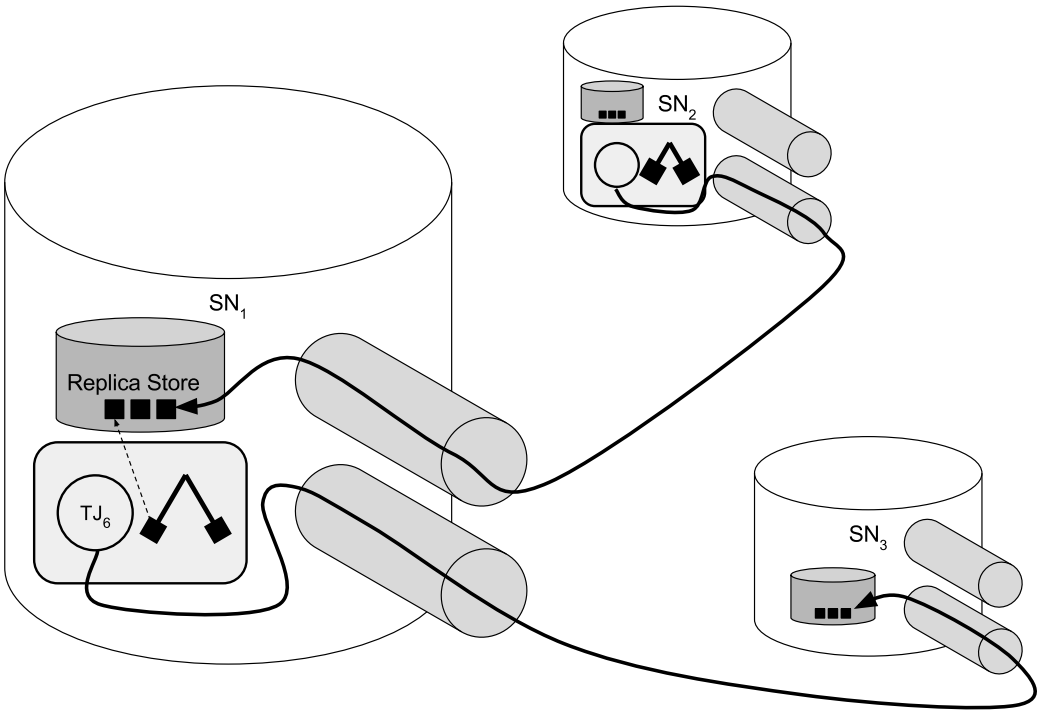


Figure 5.3. Transfer Slots

Transfer slots are used to limit concurrent asynchronous push transfers.

occupied by transfer jobs. So, an active push transfer uses a transfer slot at both the sender and the receiver. This mechanism effectively enforces load control by limiting the number of parallel push transfers a storage node may participate in. If the head node needs to use an occupied transfer slot, it must defer the transfer until a transfer slot becomes available.

Transfer slots are an abstraction only known by the head node. The storage nodes do not enforce these limits[2]. Instead, the scheduler keeps track of running transfers and enforces the transfer slot limits on itself.

The idea of transfer slots is loosely based on *Map* and *Reduce* slots in Hadoop's version of MapReduce [20]. Hadoop uses these slots to approximately divide up

[2]Storage nodes only have weak limits on the number of running jobs which are configured at start on the command-line.

available cores, memory, and network resources among the two task types. Confuga also uses this idea of slots to abstract the demand placed on a storage node's network and disk resources.

Transfer slots are not occupied by regular jobs (i.e. not transfer jobs). So a storage node may participate in asynchronous push transfers while a regular job is executing there. This policy was chosen so that replication may continue even while the cluster is under load. Additionally, immediately placing a job on a storage node hosting a replica used by other future jobs would prevent its replication until the job finishes. Thus, having regular jobs occupy one or all transfer slots would severely hurt data parallelism. On the other hand, resource contention may be introduced by transfer jobs that will make requests to the storage device. Those requests could interfere with the same requests by a regular job. This work does not study contention from this scenario directly.

5.4 Job File Replication

Before a job can be dispatched to a storage node for execution, Confuga must fully replicate all missing dependencies on the storage node. This is a result of several non-orthogonal decisions:

- `read-on-exec` **Consistency Semantics**: Each dependency is read entirely prior to execution. External modifications during the job's execution are not visible.
- **No Dynamic Access**: Jobs may only access the files given in their dependency list.
- **Uncoupled POSIX Sandbox**: Jobs execute within a POSIX sandbox that does not require special file system mounts or library use. Instead, regular files are linked into a sandbox for the job to access.
- **Whole File Access**: Jobs operate under the assumption that all parts of a whole file dependency are available with constant access cost.

Confuga performs all push transfers for a job while it is *scheduled*. A job in this

state has been assigned to a storage node but its dependencies are not yet all been replicated. The scheduler forms decisions about file transfers considering only jobs in the scheduled state.

In this dissertation, when discussing the scheduler I am referring to the aspect which schedules transfers to replicate job dependencies. For scheduling jobs, Confuga uses a simple First-In-First-Out (FIFO) scheduler for all configurations. While there has been significant effort in the community for constructing schedulers emphasizing fairness [94] (for users), smarter placement, and rescheduling [42], this work and dissertation is focused on how to control load on the cluster, particularly the network load **after** scheduling (placing) jobs.

All transfers we analyze in this chapter are *directed*; the scheduler controls the movement of all files. This allows Confuga to completely control load on the storage nodes and cluster network. Storage nodes do not independently initiate any transfers. Analyzing the potential benefits of *undirected* transfers by storage nodes independently pulling job dependencies is subject of Chapter 6.

5.4.1 Constraining Concurrent Job Scheduling

In early designs of Confuga, jobs were optimistically scheduled on all available storage nodes in one phase of the scheduler. The scheduler would then move on to replicating necessary dependencies for all of these scheduled jobs. For some workflows, this has not been the best default approach as workflows rarely fully utilize the cluster (due to replication and job execution time). Instead, it can be useful to conservatively limit the number of *scheduled* jobs that the Confuga scheduler considers at a time. This allows some jobs to execute sooner and enables future jobs to possibly reuse the same nodes that become available. Additionally, each scheduled job uses more of the cluster network for replicating its dependencies.

We refer to this scheduling strategy as `fifo-m`, where `m` is the maximum number of

jobs in the scheduled state at any time. The early optimistic scheduler corresponded to `fifo-inf`, which practically limits the scheduler to having up to one scheduled job for each storage node (so `fifo-inf` is equal to `fifo-j` where `j` is the number of storage nodes).

Synchronous transfers (`sync`) always uses the `fifo-1` configuration of the scheduler. The reason for this is that transfers in this configuration are serially performed on one scheduled job until all of its dependencies are replicated. So, no transfer parallelism is gained by having multiple jobs in the *scheduled* state.

5.4.2 Constraining Concurrent Transfer Jobs

Once a job is in the scheduled state, the scheduler attempts to replicate any missing dependencies to the storage node it is assigned to. When using asynchronous transfers, a greedy approach would immediately dispatch transfer jobs for all missing dependencies. When there are several large dependencies, this would result in the target storage becoming overloaded or in the cluster network becoming saturated. In addition, some source storage nodes hosting many of the missing dependencies may also become overloaded by transfers out.

As discussed in Section 5.3.1, Confuga resolves this problem through the use of transfer slots. The head node can be configured to assign `n` transfer slots to each storage node. This prevents a node from becoming the target or source of more than `n` concurrent transfer jobs. So the scheduler must wait to replicate any missing dependencies for a scheduled job until the storage node has free transfer slots available. Likewise, a source for a popular replica cannot be overloaded by more than `n` transfers. We refer to this scheduling policy as `async-n`.

Figure 5.4 shows the Confuga scheduler with the `fifo-m` and `async-n` configurations. These configurations are set in the Confuga URI shown in Section 3.2.3.

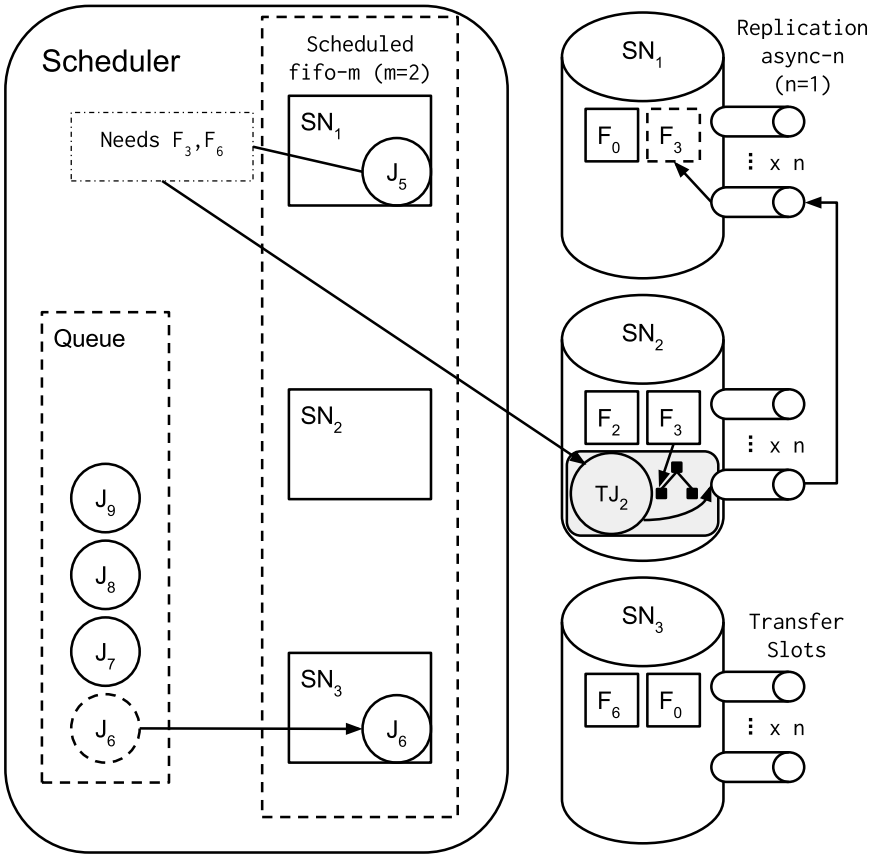


Figure 5.4. Confuga Job Scheduler

This diagram shows the two scheduler parameters we are manipulating in this chapter: `fifo-m` *and* `async-n`*. This figure shows* `m = 2` *and* `n = 1`*.* `fifo-m` *limits the number of jobs,* `m`*, which may be in the "scheduled" state, where the job has been assigned a storage node for execution and may begin replicating missing dependencies. So,* J_7 *may not be scheduled until either* J_5 *or* J_6 *leaves the scheduled state and is dispatched.* `async-n` *limits the number of transfers to and from a storage node. Two missing dependencies of* J_5 *need to be replicated to* SN_1*:* F_3 *and* F_6*.* F_3 *is currently being replicated to* SN_1*, via "Transfer Job"* TJ_2 *(a Transfer Job is simply a job which transfers a file).* F_6 *will wait to be replicated until both* SN_1 *and* SN_3 *have a free transfer slot* ($n = 1$)*.*

© 2015 IEEE.

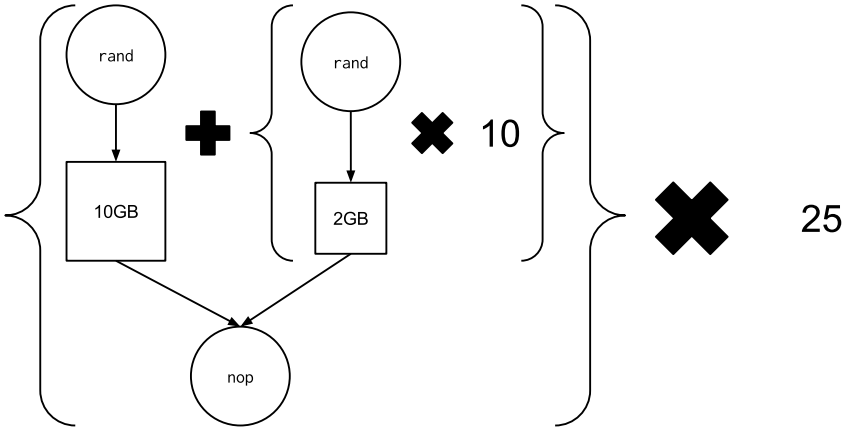


Figure 5.5. Directed Transfer Stress Test Workflow

This workflow is designed to stress the Confuga scheduler with several large dependencies that must be distributed throughout the cluster.
© 2015 IEEE.

5.5 Evaluation

This section examines the consequences of allowing asynchronous and concurrent replication within the cluster. The goal of concurrent replication is to fully utilize the available network and disk resources. As usual, concurrency is not a completely positive change. The cluster may not be able to fully use the resources it allocates or the accounting overhead of concurrency may slow down the head node.

I have evaluated Confuga's scheduling parameters described in the previous section using the workflow visualized in Figure 5.5. The goal of this workflow is to stress the Confuga scheduler with short jobs and several large data dependencies. The workflow uses 25 instances of a simple producer and consumer pipeline. (One producer/consumer per storage node.) The consumer executes no operation (`NOP`) as we are only interested in the transfers needed to move dependencies. Each consumer receives 30GB of data from 11 producers. The hardware configuration of the cluster is as described in Section 1.3.

Figure 5.6 shows the *makespan* of the workflow for varying configurations of the

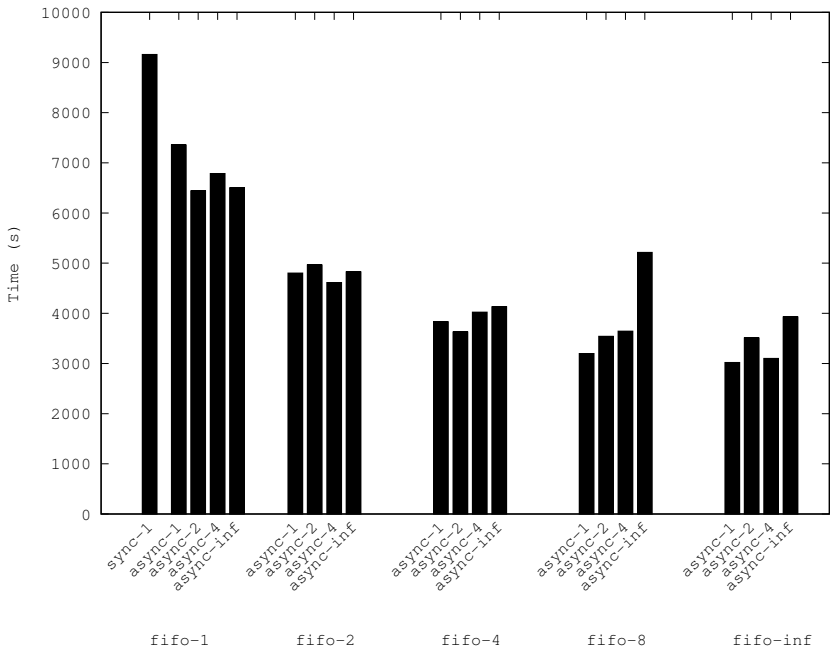


Figure 5.6. Time to complete

© 2015 IEEE.

scheduler. The makespan includes the time to execute the producers, replicate dependencies for the consumers, and execute the consumers.

This graph indicates that the most important factor in decreasing the makespan is increasing the number of jobs in the scheduled state, i.e. `fifo-0` to `fifo-inf`. Arbitrarily limiting the focus of the scheduler to a smaller set of scheduled jobs negatively impacted performance. Instead, it is better to allow the scheduler to perform more transfers. Interestingly, from a time standpoint, this figure indicates transfer slots have a minor impact on the performance of the workflows. However, we will see that the use of transfer slots can have a significant impact on the bandwidth of the cluster.

For the same workflow, Figures 5.7 and 5.8 show the bandwidth of the cluster and individual transfers, respectively. The cluster bandwidth is the average aggregate bandwidth of transfers. This view of the performance of transfers gives us key insights into the ability of Confuga to replicate needed dependencies.

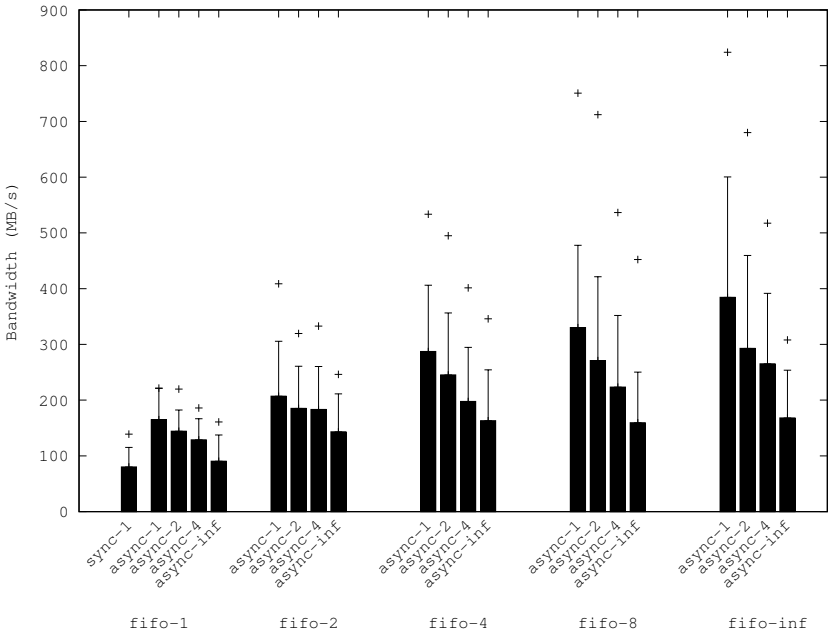


Figure 5.7. Cluster Transfer Bandwidth

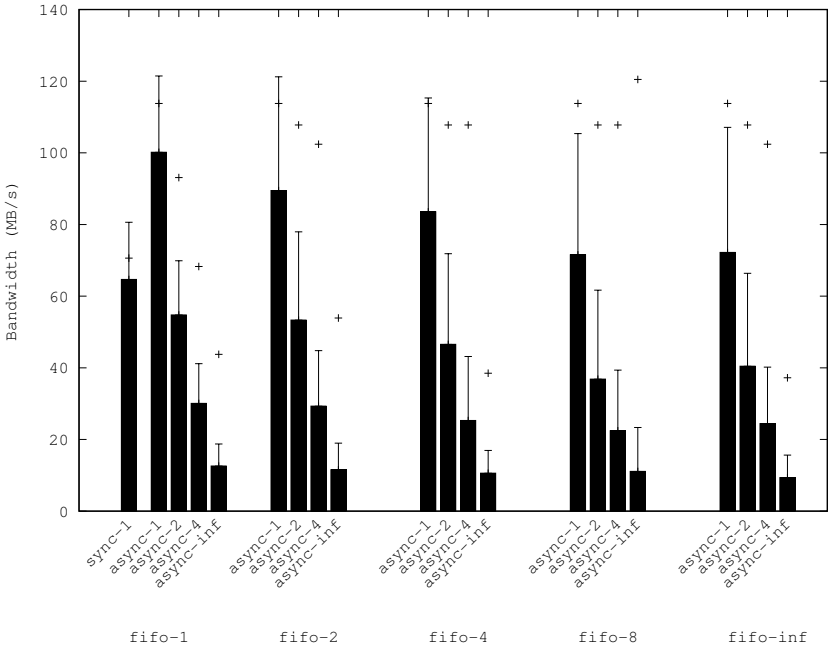


Figure 5.8. Individual Transfer Bandwidth

Bars, whiskers, and stars are respectively average, standard deviation, and maximum.

© 2015 IEEE.

Because the workflow is limited by the ability to transfer files between nodes, an increase in the cluster bandwidth leads to a decrease in execution time. Increasing the number of scheduled jobs via `fifo-m` has the most significant impact on the cluster bandwidth, across all configurations of `async-n`, except `async-inf`. This indicates that the parallelism gained by increasing the number of transfer slots (`n = 1 to inf`) has a strongly negative impact on transfer performance even when there is only one scheduled job (`m=1`).

Individual transfer bandwidth is negatively impacted by an increase of concurrency of transfers on a storage node (`async-n`) or on the cluster (`fifo-m`). Despite this, the utilization of the cluster network increases. This allows for the system as a whole to accomplish more, even though individual transfers are slower.

Restricting the cluster to a single transfer slot per storage node, `async-1`, achieves the best performance for all configurations of `fifo-m`. Allowing a single transfer to saturate the link for a storage node maximizes the bandwidth of individual transfers even as the cluster performs transfers for more scheduled jobs. This result indicates that while transfer slots are a useful abstraction for managing transfers in the cluster, increasing transfer parallelism at the storage node is not beneficial.

Overall, this experiment shows that a directed and controlled approach to managing transfers on the cluster is essential for achieving performance. For example, enforcing a limit on transfers to one per storage node offered a 228% increase in average cluster bandwidth and a 23% reduction in workflow execution time (`fifo-inf/async-1 vs. fifo-inf/async-inf`).

5.5.1 Reflection on Results

In our own clusters, we recommend using `fifo-inf` as a default. The reader may wonder why this configuration was introduced when the early prototype already had that behavior. Limiting the number of scheduled jobs happened to have an extremely

positive impact on certain workflows with jobs that had large dependencies but short execution time. Scheduling jobs on all of the storage nodes committed the head node to distributing the dependencies to all storage nodes. When the cluster is limited to a single scheduled job (`fifo-1`), the cluster makes small commitments to replication. The time to completely copy dependencies on one storage node to another is obviously faster than to distribute the same dependencies to the entire cluster. Most importantly, when jobs finish more quickly than the time to replicate the dependencies, a future job can be scheduled to replace the finished job. In effect, the jobs are satisfied by a small working set of storage nodes.

I expect that future work will look at other mechanisms to achieve the advantages of fewer scheduled jobs without limiting the cluster transfer bandwidth. Better informed *job* schedulers like Quincy [42] will reschedule jobs instead of continuing replication unnecessarily when jobs are satisfied by a smaller number of storage nodes.

Likewise, `async-1` is used as a default in Confuga. The usefulness of multiple transfer slots had limited application in numerous experiments. Some workflows with several small files experienced better performance when the scheduler was able to reduce latency between transfers[3]. A different mechanism, pull transfers, were explored instead to reduce transfer latency in this case. This is discussed in the next chapter.

5.6 Conclusion

I have introduced directed transfers used by Confuga to replicate files across the cluster to fulfill job dependency requirements. Directed transfers are made possible by Confuga exploiting the explicit job namespace descriptions from the workflow and by Confuga limiting the consistency semantics for jobs. I have shown that this

[3]Recall that transfer slots also limit the number of transfers to the storage node a job is scheduled on. So `async-1` will effectively serialize all replication to that storage node.

mechanism of replication by the file system empowers the head node to control load on the cluster and optimize transfers for efficient disk and network utilization.

We will continue to show that the use of directed transfers has a defining impact on Confuga to control load on the cluster in the face of many types of workflows. The next chapter introduces pull transfers which does a comparative and cooperative analysis of the two transfer strategies.

CHAPTER 6

BALANCING DIRECTED AND UNDIRECTED TRANSFERS

6.1 Introduction

Next we will move on to discussion of techniques for managing transfers within an active storage cluster file system. In the context of executing scientific workflows, we will see that this is a first order problem that must be addressed in the design of the system.

Confuga is built around the idea of leveraging the **job namespace** to achieve a stable system. While typical distributed file systems must be designed to support runtime access to any file at any time, Confuga is able to scope job visibility of the global namespace to the job's own defined subset. This idea is fundamental to the design of Confuga. Requiring the declaration of the job namespace allows Confuga to unobtrusively eliminate dynamic transfers by jobs and plan the replication of all job dependencies. This management of transfers empowers Confuga to control load on the cluster.

In this chapter, we will begin by introducing another classification of transfers. *Undirected* transfers resemble more traditional file transfers where the job fetches missing files itself. Confuga implements this using **pull** transfers. Pulls are used by the head node to selectively off-load transfer scheduling and management to storage nodes when a controlled distribution has fewer benefits.

Next, we will examine the comparative benefits of using pushes and pulls to manage transfers within the cluster. While push transfers allow the head node to

control network and disk load on storage nodes, pulls may be used to selectively off-load transfer scheduling and management to storage nodes when a controlled distribution has fewer benefits. This work finds there is a balance to strike between pushes and pulls: the careful use of pull transfers can avoid inefficiencies introduced by centralized management of transfers. This chapter will show:

- Pushes enable full disk and network utilization of storage nodes. Load control of transfers can allow for an efficient spanning tree distribution that optimally distributes files in parallel. Using push transfers, Confuga achieves a 77% speedup over unmanaged replication via pulls. (*Section 6.4*)
- Replicating several large dependencies reduces the mutual interference pulls suffer but performance is still unpredictable. On the other hand, push transfers still eliminate all load instability caused by concurrent transfers in a predictable way. (*Section 6.5*)
- When jobs have several dependencies that must be pulled, it is essential that these transfers occur in a random order to avoid hot-spots. The effects of hot-spots are usually only visible for larger files as pull transfers are more likely to interfere. (*Section 6.6*)
- While push transfers allow for fast distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files. Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead using pull transfers introduces small amounts of interference but individual pull and push transfer bandwidth is mostly unaffected. (*Section 6.7*)

I conclude in Section 6.8 with two representative bioinformatics workflows evaluated using the push and pull transfer mechanisms. Ultimately, we show that a balance of the two mechanisms achieves optimal file distribution leading to 48% and 77% improvements over only push or pull.

6.2 Pull Transfers

Pull transfers are replica transfers which are executed by the storage node executing the job and prior to the execution of the job application. Pulls allow the

Confuga scheduler to delegate transfer management to storage nodes. This frees the scheduler to devote resources elsewhere but pulls may introduce load instability on storage nodes. For example, several nodes may pull from the same replica simultaneously, with deteriorated performance. Additionally, as with push transfers, a job may pull a file from another storage node executing a job.

Pull transfers resemble normal whole-file sequential reads in a typical distributed file system. For example, a job *opens* the file, *looks up* an available replica, *reads* parts of the replica, and then *closes* the file. In Confuga, pull transfers behave similarly but differ in several important ways. First and foremost, a storage node does not decide *which* files are pulled. The Confuga scheduler is free to perform push transfers for some files and leave remaining dependencies to be pulled by the storage node. Second, jobs do not and cannot initiate a transfer during execution. Pulls are performed prior to application execution. Third, because the entire namespace is known, the Confuga scheduler is free to perform replica lookups in batch prior to job dispatch.

Figure 6.1 shows an example of a job performing pull transfers. Pulls use the `getfile` RPC [81] to fetch a copy of a replica from a remote storage node. Pulls are executed as part of setting up the job sandbox by storage nodes.

Since Confuga knows the replicas on each storage node, it is able to write the job description so that each input binds to either a replica on the storage node or to one or more replicas on other storage nodes. For each input file without a local replica, Confuga will include a set of randomly chosen remote replicas specified as a list URLs [9]. The storage node will attempt to fetch each URL for the file until success. The result is placed within the job's sandbox. If a pull ultimately fails to obtain the file from any of the possible replicas, then the job will abort. This failure is responded to by the head node when the job is reaped by either creating a new job or passing the failure up to the workflow manager. Listing 6.1 shows an example job specification with pull transfers.

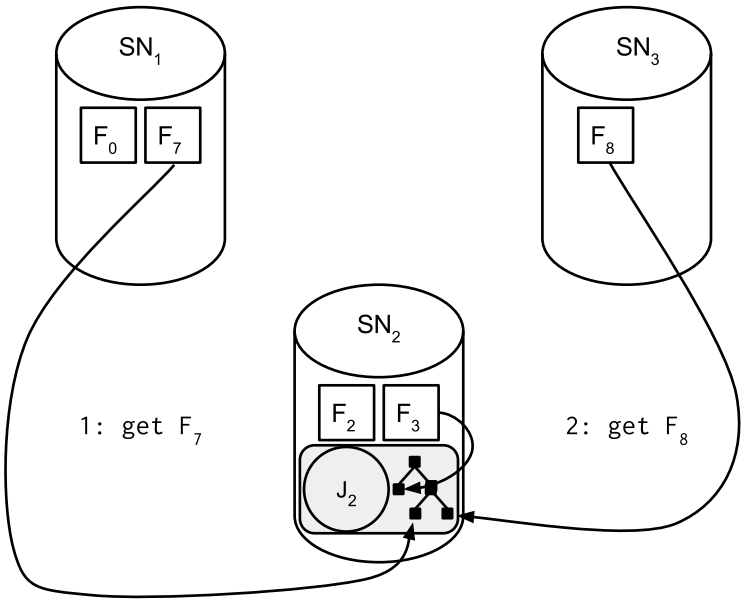


Figure 6.1. Pull Transfers in Confuga

These are transfers performed by the job prior to executing the application. The `getfile` *RPC is used to fetch a replica from another storage node and place it in the job sandbox. Pulls may include several storage nodes (depending on the file replication factor) to alternatively fetch from. This allows the job to robustly handle failed transfers or storage nodes by selecting a different source.*

Confuga also stores the copy of the file saved in the sandbox as a new replica once the job completes. This is done by adding an output file with the same `task_path` in the job's file list. In Listing 6.1, `./finishjob.sh` is saved in the replica namespace just like other outputs. The purpose of saving these pulled replicas is to increase the replication factor of hot files to avoid future transfers. Listing 6.2 shows the wait status for the same job with the output file bound. Its interpolated `serv_path` stored `./finishjob.sh` in the storage node's replica namespace and the file `size` is set. The head node records the new replica with this information.

```
{
   "executable":"/bin/sh",
   "tag":"confuga:48D70B144F39860470A80DDCD6367C4333A2A7D4",
   "arguments":["sh", "-c", "./finishjob.sh 3909"],
   "environment":{"CHIRP_CLIENT_TICKETS":"./.confuga.ticket"},
   "files":[
      {
         "binding":"LINK",
         "serv_path":"/users/pdonnel3/.confuga/ticket",
         "task_path":"./.confuga.ticket",
         "type":"INPUT",
      },
      {
         "binding":"URL",
         "serv_path":"chirp://disc02.crc.nd.edu:9155/users/pdonnel3/.confuga/file/
             D3FD3E767446768EAB13A93E40F5369F82EE4268,chirp://disc10.crc.nd.edu
             :9155/users/pdonnel3/.confuga/file/
             D3FD3E767446768EAB13A93E40F5369F82EE4268",
         "task_path":"./finishjob.sh",
         "type":"INPUT",
         "size":382
      },
      {
         "binding":"LINK",
         "serv_path":"/users/pdonnel3/.confuga/file/%s",
         "task_path":"./finishjob.sh",
         "tag":"confuga-file-pull",
         "type":"OUTPUT",
      },
      ...
   ]
}
```

Listing 6.1: Example Job with Pull Transfers

6.3 Evaluating Transfer Management

Now we will move on to evaluating Confuga's use of push and pull transfers in several workflows. These experiments will explore the benefits gained by controlling cluster load through pushes or relaxing control through pulls. To do this, the experiments are structured to stress the head node's ability to replicate dependencies across the cluster. The cluster hardware used in this chapter is the same as described in Section 1.3.

This effort does not scrutinize the scheduling of jobs (i.e. assigning jobs to storage nodes for execution) which achieves certain well-studied goals like fairness. Scheduling (especially *rescheduling*) can minimize or eliminate data transfers but that analysis is beyond the scope of this work. Our concern in these experiments is how to efficiently manage transfers once jobs are placed to minimize distribution time and

```
{
   "id":8,
   "exit_code":0,
   "exit_status":"EXITED",
   "status":"FINISHED",
   ...
   "files":[
      {
         "binding":"URL",
         "serv_path":"chirp://disc02.crc.nd.edu:9155/users/pdonnel3/.confuga/file/
             D3FD3E767446768EAB13A93E40F5369F82EE4268,chirp://disc10.crc.nd.edu
             :9155/users/pdonnel3/.confuga/file/
             D3FD3E767446768EAB13A93E40F5369F82EE4268",
         "task_path":"./finishjob.sh",
         "type":"INPUT"
      },
      {
         "binding":"LINK",
         "serv_path":"/users/pdonnel3/.confuga/file/
             D3FD3E767446768EAB13A93E40F5369F82EE4268",
         "size":382,
         "tag":"confuga-file-pull",
         "task_path":"./finishjob.sh",
         "type":"OUTPUT"
      },
          ...
   ]
}
```

Listing 6.2: Finished Status of Example Job with Pull Transfers

storage node load.

The 2-stage Producer/Consumer workflow shown in Figure 6.2 is used to evaluate Confuga's ability to distribute dependencies across storage nodes for various scheduler configurations. The function of the producers is to quickly generate the pool of dependencies randomly across all storage nodes. Each consumer is assigned by the scheduler to one of the available storage nodes and dependencies are replicated.

I note that a variation on this workflow is not included where consumers also access files which are unique to their execution (i.e. not shared with other consumers). This variation is generally uninteresting when analyzing push and pull configurations because unique files are only replicated at most once, so therefore it is unlikely for this to introduce significant contention or load.

These experiments limit each storage node to a single transfer slot. As discussed in the previous chapter, this restricts storage nodes to participating in a single push transfer at any given time. Based on the last chapter's experiments, we found this

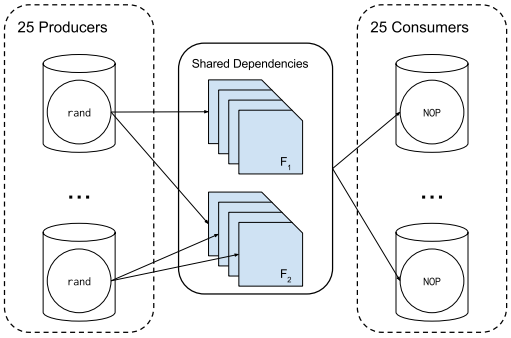


Figure 6.2. Push and Pull Transfer Stress Test Experiment

This producer/consumer workflow is used to stress the Confuga cluster through transfers of several dependencies. Each of the 25 storage nodes produce 1/25 subset of shared dependencies. The scheduler then must completely distribute these files across the cluster for 25 consumers (1 consumer per storage node), which execute no operation.

Workflow Shared Dependencies:
Workflow A: 1·32GB file.
Workflow B: 25·32GB files.
Workflow C: 1·64GB, 2·32GB, 4·16GB, 8·8GB, 16·4GB, 32·2GB, 64·1GB files.

limit provided the best predictable performance for push transfers without impacting global file distribution time. This chapter examines the use of pull transfers to increase parallelism in other ways which do not involve the head node.

6.4 Spanning Tree Distribution

In this section we will test our hypothesis that using push transfers can help control load on the cluster and improve file distribution time. Confuga performs load management by limiting concurrent push transfers for a storage node via transfer slots. This load management will result in a tree distribution of the file. The form of the tree is determined by the transfer load on the cluster. Under conditions where the file may be continually replicated using new replicas as they become available, a spanning tree file distribution [81] develops.

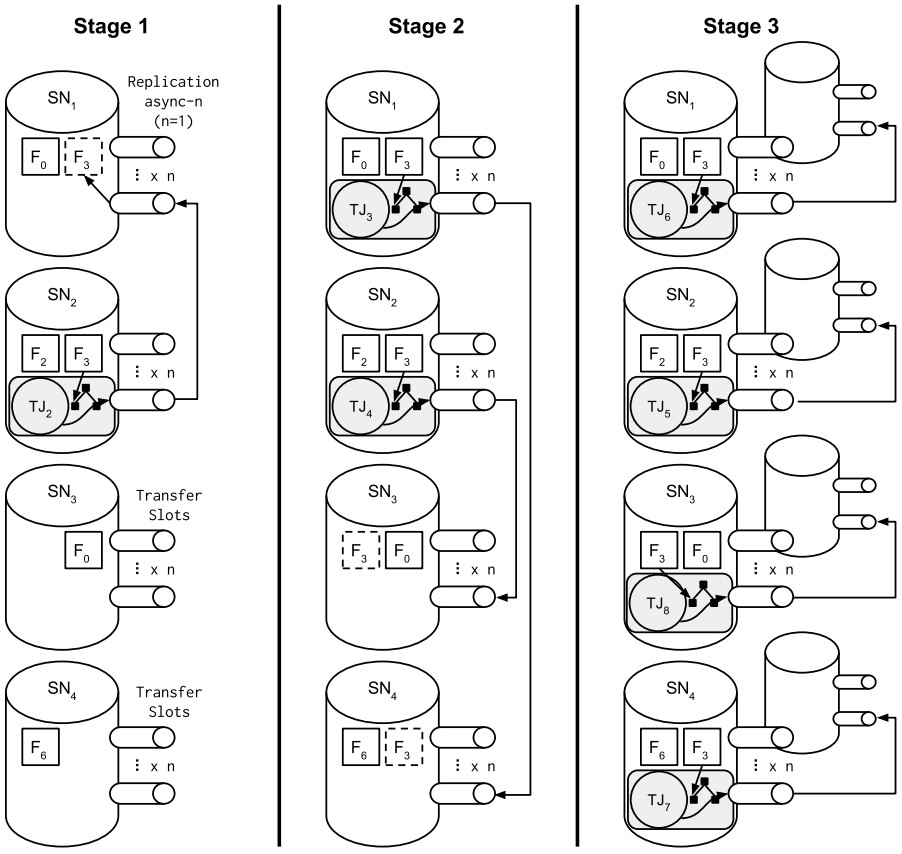


Figure 6.3. Spanning Tree of Push Transfers

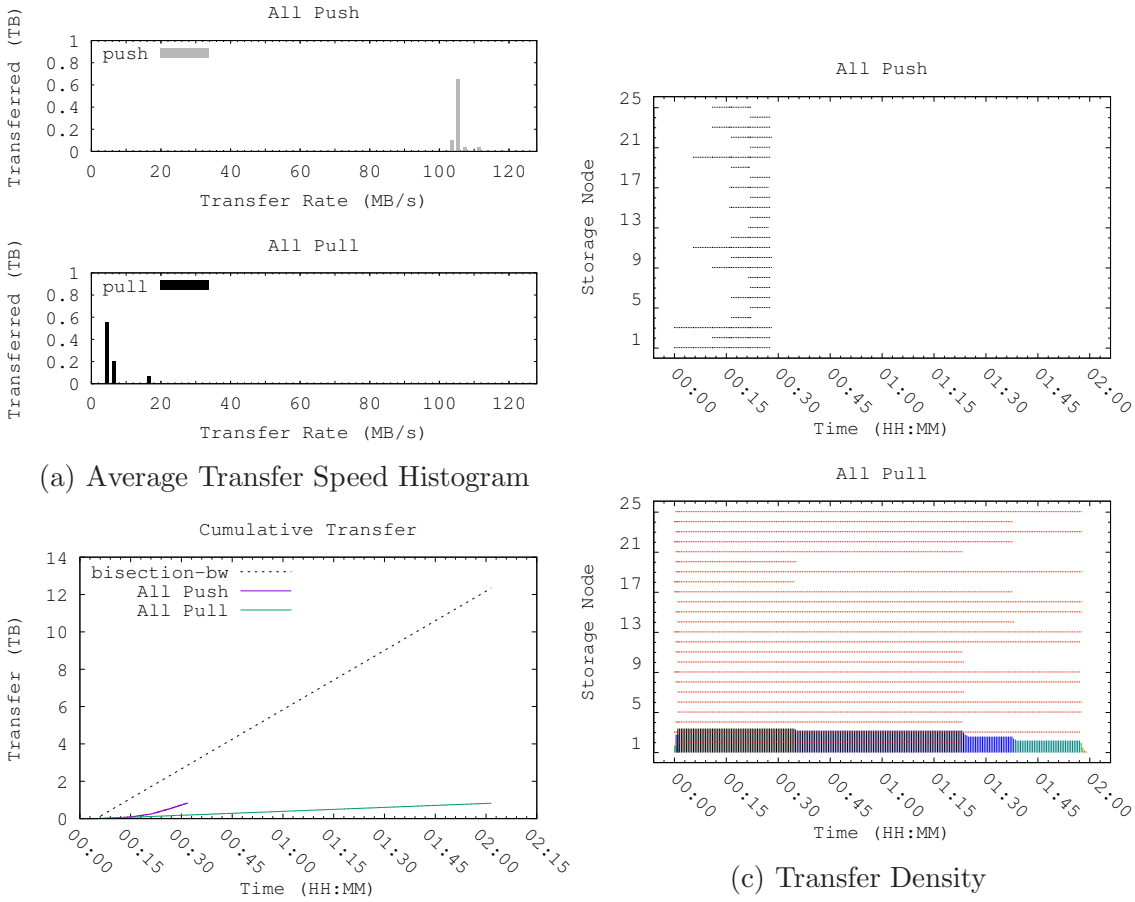
Each push is a Transfer Job (TJ). One transfer slot per SN (n=1).

This process is visualized in Figure 6.3. The Confuga scheduler allocates to each storage node a single transfer slot which limits the storage node to one incoming or outgoing transfer. A transfer job is dispatched by the scheduler which occupies the transfer slot of the storage node it executes on and the transfer slot of its target. The use of transfer slots allows the scheduler to control the number of pushes executing in the cluster and to functionally create a spanning tree distribution for files.

We examine a workflow which requires the distribution of a single large file across all storage nodes. This is done using **Workflow A** described in Figure 6.2, where all consumers require a single shared 32GB file. We look at the two cases where the file is distributed using push transfers or pull transfers.

Figure 6.4 shows the results. For "All Push" in (c), Confuga is visibly able to achieve a spanning tree distribution of the 32GB file using push transfers. At the start, node 1 pushes the file to node 3. At approximately 00:07, the replication finishes and nodes 1 and 3 begin pushing to nodes 11 and 20. And so on. This distribution methodology minimizes storage node load and maximizes individual transfer bandwidth ("All Push" in (a)). On the other hand, using pull transfers causes all of the storage nodes to naïvely herd the single source storage node hosting the 32GB file ("All Pull" in (c)) and thus suffer from low individual transfer bandwidth ("All Pull" in (a)).

Conclusion: Executing push transfers allows the scheduler to efficiently distribute large files while controlling load on the cluster. Storage nodes are able to transfer files using the full disk and network bandwidth. Ultimately, centralized management of transfers allows for an efficient tree distribution of files that maximizes transfer parallelism and minimizes contention. In this case, the file distribution using push transfers benefited from a 77% speedup.



(a) Average Transfer Speed Histogram

(b) Cluster Cumulative Transfer

(c) Transfer Density

Figure 6.4. Workflow A: Single File Spanning Tree Distribution

(a) Average Transfer Speed Histogram *groups transfers by average transfer rate and shows the total bytes transferred for each group.*

(b) Cluster Cumulative Transfer *visualizes the cumulative bytes transferred across the cluster. The* `bisection-bw` *line indicates the maximum bisection transfer bandwidth of the cluster, without saturating the cluster switch and limited by the 140MB/s disk bandwidth.*

(c) Transfer Density *visualizes the ongoing transfers for each storage node for the duration of the workflow. Each row of the y-axis is a storage node in the cluster. The height of each tic in each row indicates the number of ongoing transfers for the storage node. The height of a storage node row is set to 10 concurrent transfers, so some tics exceed the height of a storage node row during heavy activity.*

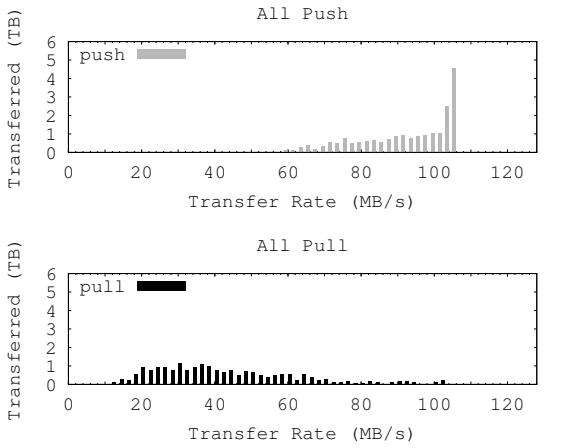
6.5 Concurrent Distribution of Multiple Dependencies

Next, we test that load control from push transfers improves file distribution time even when there are more opportunities for transfer parallelism. We look at a workload where the cluster must fully distribute several large file dependencies across the entire cluster. Pull transfers in the previous workflow suffered because all of the jobs were pulling from the same replica simultaneously, allowing for no transfer parallelism. Here, we look at file distribution in the cluster when jobs pull from several large dependencies. We expect this to be significantly different from the previous Workflow A for two reasons: (1) the first push transfer for each of the several dependencies can be performed concurrently; and (2) the load on individual storage nodes by pull transfers is reduced because not all storage nodes are attempting to pull the same dependency simultaneously. (That is to say, the storage nodes are pulling dependencies in a random order. We look at pull ordering in Section 6.6.)

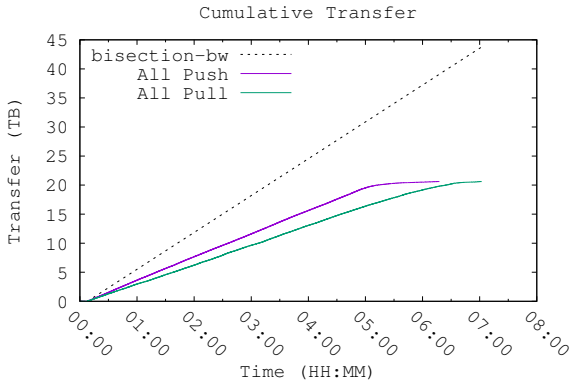
Workflow B expands on Workflow A by increasing to $25 \cdot 32$GB shared files. This also requires all of the 32GB files to be replicated across the entire cluster for each consumer job. Each file is replicated 24 times so the cluster needs to transfer approximately 20TB of data during the course of the workflow. Again, we run the workflow with two configurations: all push and all pull transfers.

Figure 6.5 shows the results for Workflow B. The distribution time for push and pull is virtually the same. The spanning tree distribution used by push transfers has negligible impact because pulls also benefit from transfer parallelism via multiple dependencies. The aggregate cluster bandwidth for both configurations is constant except for a long tail (b). Pulls do marginally worse due to periods of storage node contention resulting in lower transfer bandwidth (a).

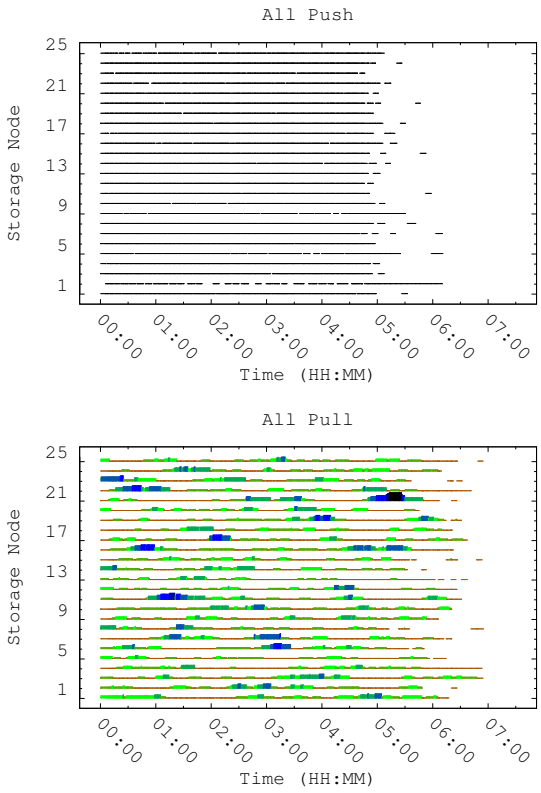
On the other hand, push transfers deliver consistently higher bandwidth compared to pulls by eliminating contention (a) but this does not lead to a significant improvement in distribution time. Because of the odd number of consumer jobs (25)



(a) Average Transfer Speed Histogram



(b) Cluster Cumulative Transfer



(c) Transfer Density

Figure 6.5. Workflow B: Multiple File Concurrent Spanning Tree Distribution

(a) Average Transfer Speed Histogram *groups transfers by average transfer rate and shows the total bytes transferred for each group.*

(b) Cluster Cumulative Transfer *visualizes the cumulative bytes transferred across the cluster. The* `bisection-bw` *line indicates the maximum bisection transfer bandwidth of the cluster, without saturating the cluster switch and limited by the 140MB/s disk bandwidth.*

(c) Transfer Density *visualizes the ongoing transfers for each storage node for the duration of the workflow. Each row of the y-axis is a storage node in the cluster. The height of each tic in each row indicates the number of ongoing transfers for the storage node. The height of a storage node row is set to 10 concurrent transfers, so some tics exceed the height of a storage node row during heavy activity.*

and each storage node having a single transfer slot, only 12 transfer jobs can be scheduled at a time. This limit on push transfers leads to the long tail at the end of the distribution and several transfer gaps visible in "All Push" in (c). Additionally, the spanning tree distributions for all of the dependencies do not progress in lock step. Due to random factors and opportunistic scheduling, some files will finish distribution much earlier in the workflow.

Conclusion: This workflow shows that random pulling of large dependencies across the cluster can achieve comparable performance to structured push transfers. Even so, the pull transfer bandwidth suffers in unpredictable ways. This makes it more difficult for the head node to predict transfer load on storage nodes. Additionally, the hot-spots on the cluster introduce more opportunities for transfer and job failures. Altogether, this makes pull transfers less attractive for distribution of large files.

6.6 Execution Order of Pull Transfers

We will now examine how the execution order of pull transfers order can lead to unanticipated load, causing extreme transfer slow downs. This arises from a common situation in workflows where a group of jobs have one or more shared input file dependencies. These dependencies must be distributed across the cluster as new replicas to support parallel execution of jobs. An unexpected problem is that the order of the pull transfers has a significant impact on reducing contention. If storage nodes perform pull transfers for common input files in the same order – frequently the case when executing a workflow on the cluster – then the storage node hosting the first dependency will suffer uncontrolled load.

To evaluate this, we use the same Workflow B used in Section 6.5 but with a deterministic pull ordering (the order used by the workflow manager specifying the jobs). So each consumer job will pull its dependencies in the same order. We are only

interested in the behavior of the workflow and cluster for pull transfers. For comparison, we include the results of previous experiment which used randomly ordered pulls, with axis ranges adjusted if appropriate.

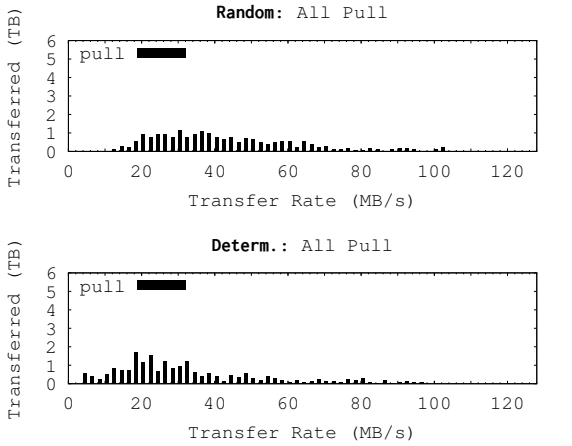
Figure 6.6 shows the results of the experiment. The transfer density of the cluster in (c) is the most telling figure of this experiment. It shows each storage node's transfer activity within the cluster across the duration of the workflow. For a deterministic pull ordering, some storage nodes (beginning with node 20) suffer debilitating transfer load because they host the replica first pulled by other jobs. In contrast, the random ordering has more uniform transfer load across the duration of the workflow with only a few relatively small hot-spots.

The deterministic ordering has the largest impact at the beginning because the first pulls are finished incrementally, not together. Once finished pulling the first dependency from node 20, jobs move on to pulling the next dependency from node 12. As some jobs get ahead of others in progress, there are fewer instances of extreme load on storage nodes. This is also indicated in the aggregate cluster bandwidth (derivative of cumulative) in (b) where there is a slower ramp up in the first 5 hours of the workflow.

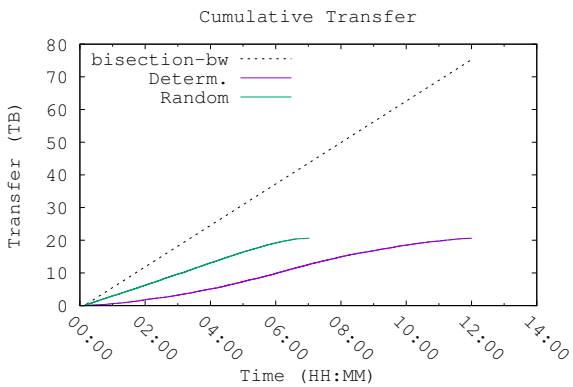
Conclusion: When jobs have several dependencies that must be pulled, it is essential that these transfers occur in a random order to avoid hot-spots. Usually, the negative effects of hot-spots are only visible for larger files as pull transfers are more likely to interfere. We would expect push transfers to be preferred to avoid this problem entirely but pull transfers may still be useful for larger files in certain circumstances.

6.7 Scaling Pull Threshold

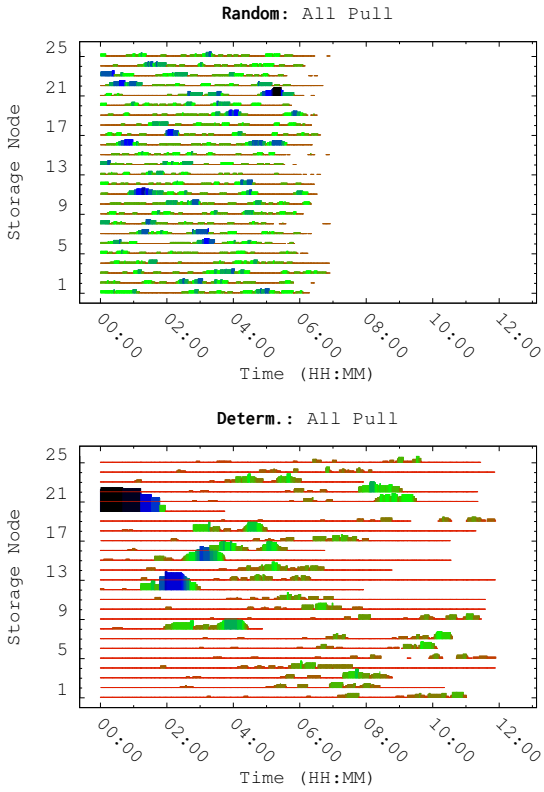
Our next experiment will test whether a balance of push and pull transfers can be achieved, benefiting from load control of pushes and the increased parallelism



(a) Average Transfer Speed Histogram



(b) Cluster Cumulative Transfer



(c) Transfer Density

Figure 6.6. Workflow B: Random vs. Deterministic Pull Transfer Ordering

(a) Average Transfer Speed Histogram *groups transfers by average transfer rate and shows the total bytes transferred for each group.*

(b) Cluster Cumulative Transfer *visualizes the cumulative bytes transferred across the cluster. The* `bisection-bw` *line indicates the maximum bisection transfer bandwidth of the cluster, without saturating the cluster switch and limited by the 140MB/s disk bandwidth.*

(c) Transfer Density *visualizes the ongoing transfers for each storage node for the duration of the workflow. Each row of the y-axis is a storage node in the cluster. The height of each tic in each row indicates the number of ongoing transfers for the storage node. The height of a storage node row is set to 10 concurrent transfers, so some tics exceed the height of a storage node row during heavy activity.*

of pulls. We use the pull threshold – the maximum file size for pull transfers – to achieve this balance. While push transfers allow Confuga to prevent debilitating load on storage nodes as shown in previous experiments, they come with costs. Setting up push transfers requires several round-trip communications with the head node and incurs the cost of setting up a transfer job. Push transfers also force a tree distribution when the effort may not be justified (e.g. for smaller files). On the other hand, pull transfers require minimal head node involvement because the head node includes with the job a list of potential storage nodes to pull from for each file. In short, pushes control load on the cluster at the cost of increased overhead and work for the scheduler while pulls decrease work on the scheduler at the cost of potential hot-spots.

Workflow C evaluates varying the pull threshold such that the work moved from push transfers to pull transfers linearly increases as the pull threshold doubles. This workflow defines shared dependencies $64 \cdot 1\text{GB}$, $32 \cdot 2\text{GB}$, ..., $2 \cdot 32\text{GB}$, $1 \cdot 64\text{GB}$. So each job requires 448GB of data. Each file is replicated 24 times (25 replicas for 25 jobs), resulting in 10.5TB transferred over the course of the workflow. The pull threshold is scaled from 0GB (all push transfers) to 64GB (all pull transfers). As we double the pull threshold, the amount of data moved by pull transfers increases by a constant 64GB per job (1.6TB for all jobs) but the number of push transfers is halved.

Figure 6.7. Workflow C: Average Transfer Speed Histogram

These graphs group transfers by average transfer rate and shows the total bytes transferred for each group.

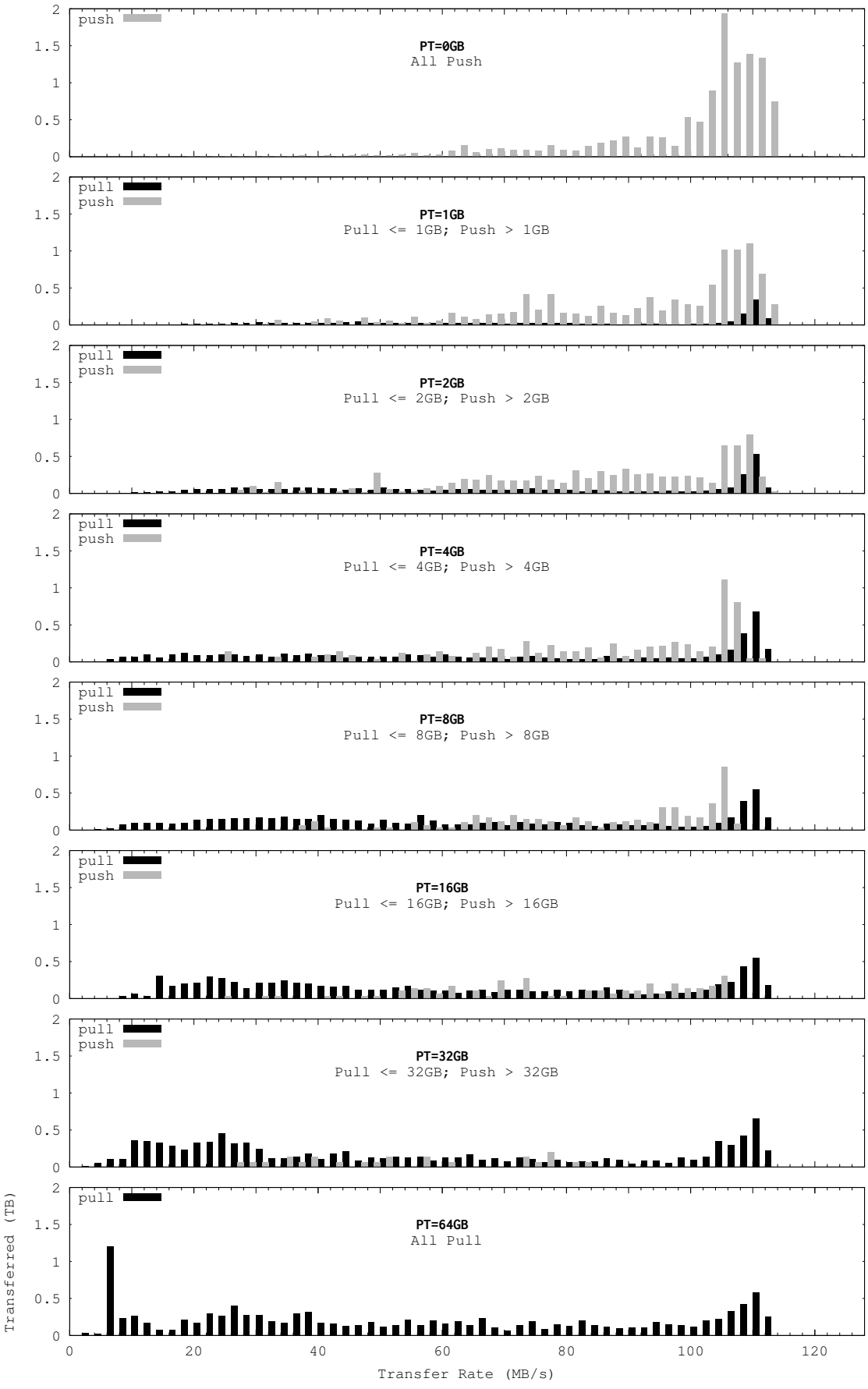
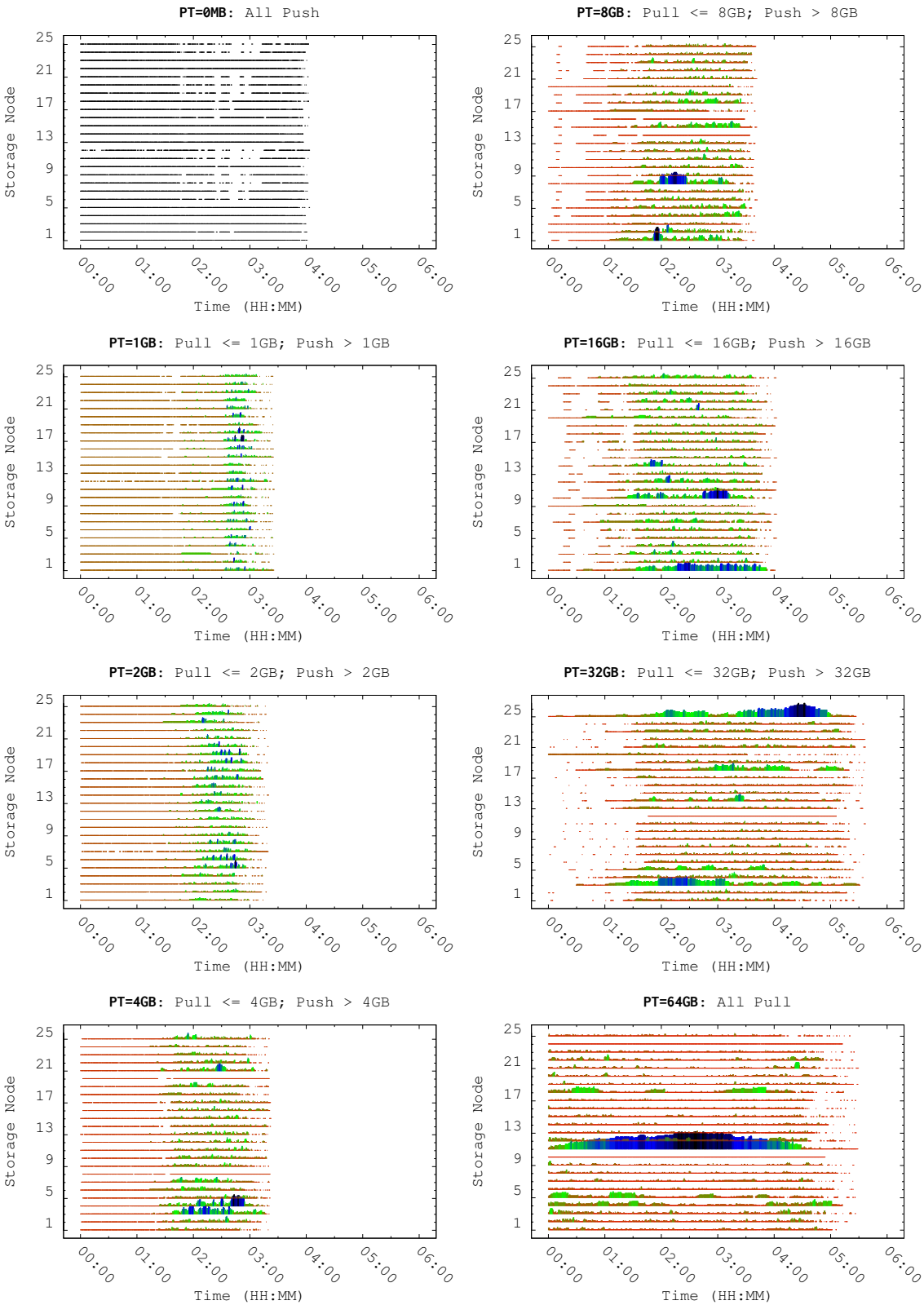

push
PT=0GB
All Push
pull
push
PT=1GB
Pull <= 1GB; Push > 1GB
PT=2GB
Pull <= 2GB; Push > 2GB
PT=4GB
Pull <= 4GB; Push > 4GB
PT=8GB
Pull <= 8GB; Push > 8GB
PT=16GB
Pull <= 16GB; Push > 16GB
PT=32GB
Pull <= 32GB; Push > 32GB
PT=64GB
All Pull
Transferred (TB)
Transfer Rate (MB/s)
0
0.5
1
1.5
2
0
20
40
60
80
100
120

Figure 6.8. Workflow C: Transfer Density

These graphs visualize the ongoing transfers for each storage node for the duration of the workflow. Each row of the y-axis is a storage node in the cluster. The height of each tic in each row indicates the number of ongoing transfers for the storage node. The height of a storage node row is set to 10 concurrent transfers, so some tics exceed the height of a storage node row during heavy activity.


PT=0MB: All Push
PT=8GB: Pull <= 8GB; Push > 8GB
PT=1GB: Pull <= 1GB; Push > 1GB
PT=16GB: Pull <= 16GB; Push > 16GB
PT=2GB: Pull <= 2GB; Push > 2GB
PT=32GB: Pull <= 32GB; Push > 32GB
PT=4GB: Pull <= 4GB; Push > 4GB
PT=64GB: All Pull
Storage Node
Time (HH:MM)
25
21
17
13
9
5
1
00:00
01:00
02:00
03:00
04:00
05:00
06:00

Figures 6.7 and 6.8 show the results of this experiment. In general, increasing the pull threshold causes the cluster to be more loaded with concurrent transfers. As more pulls replace pushes, the time span of managed load arising from push transfers (1 transfer per node) overlaps with the uncontrolled load of pulls (greater than 1 transfer per node). Despite the increased load, using pulls for the smaller files in the workflow leads to significant distribution time improvements despite reduced transfer bandwidth.

Figure 6.7 shows that the introduction of pull transfers can reduce the effectiveness of push transfers. As the pull threshold increases, the push transfers begin to have similar performance to pulls. This is caused by contention for the storage nodes' network and disk bandwidth. Even so, with a heavily loaded cluster the push transfers can have a positive effect on the distribution time. The controlled distribution of the larger files benefits from the parallelism of the tree distribution as well as fewer disk buffer cache misses because there is only ever a single transfer reading the file sequentially.

Additionally, the time taken by the head node to manage smaller transfers may take longer than the time to pull the files from a single source. For `PT=1GB` in Figure 6.7, the bandwidth for several individual pull transfers suffers due to contention and inefficient distribution of the files. Even so, using pulls for the 1GB files avoids stalls caused by slow pushes and allows parallel distribution of the smaller files alongside of the pushes. Consequently, the distribution time for `PT=1GB` gives a 12% improvement over only push transfers.

Figure 6.8 shows the transfers executing in the cluster, visualizing hot-spots. These graphs also help show the two transfer modes (push and pull) used to replicate job dependencies. Generally, the beginning of the pull transfers is indicated by storage nodes having increased concurrent transfer activity as jobs begin simultaneously pulling the replicas from the same storage node. **Note that jobs do not begin**

pulls in lockstep. While most jobs begin pulls at roughly the same time, some jobs may start pulls much earlier once the scheduler has finished push transfers for the dependencies larger than the pull threshold. This is especially noticeable for `PT=32GB` because the node hosting the 64GB dependency – the only dependency which will be pushed – can start its job without delay. The first consumer job is immediately scheduled on and dispatched to storage node 20 hosting the 64GB file. That job then begins by pulling all of its missing dependencies which causes the short transfer activity visible on the other storage nodes (the small tics from 00:00 to 00:30). After the second consumer job is scheduled to node 25, a push transfer is scheduled on storage node 20 which replicates the 64GB dependency to storage node 25. In effect, this transfer job is replicating the 64GB file to node 25 while the first consumer job is pulling its dependencies. The first 64GB push transfer finishes at 00:30 and the last at 02:21. The last 3 hours of the workflow are occupied by pulls.

As the pull threshold is increased, the head node has fewer files it needs to push. This will cause the cluster to be underutilized because a few large pushes delay all other transfers. This is indicated in the transfer gaps visible at the beginning of the workflows for `PT=8GB`, `PT=16GB`, and `PT=32GB`. At these thresholds, there is an increasingly smaller pool of files to push. For `PT=8GB`, there are 7 files which must be pushed: $4 \cdot 16$GB, $2 \cdot 32$GB, and $1 \cdot 64$GB.

Supporting multiple large concurrent transfers also puts pressure on storage node virtual memory. The disk buffer cache is unable to support the simultaneous access forcing the kernel to drop pages and perform disk seeks. For our systems with 32GB of RAM, this is especially noticeable when the pull threshold is increased from `PT=16GB` to `PT=32GB`. The distribution time increases by 40%. During our analysis, we have observed that pull transfers form groups riding the buffer cache as pages load from disk. This is indicated in the below Figure 6.9 which shows the two nodes (18 and 25) hosting the 32GB files which are pulled for `PT=32GB`. There are several groups of

pull transfers for both storage nodes which finish together as the last pages of the 32GB file are read. For example, storage node 25 had five pull transfers complete in the span of a second at approximately 05:15.

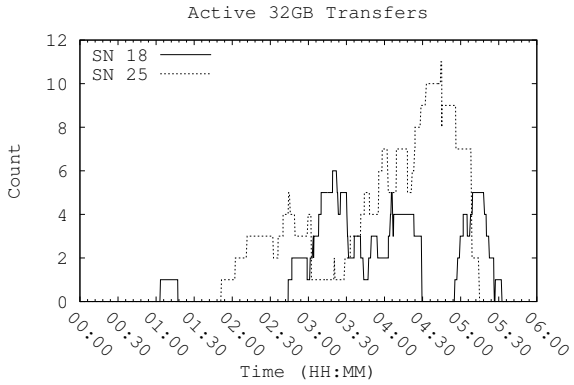


Figure 6.9. Workflow C: Active 32GB Pull Transfers for 32GB Pull Threshold

This graph shows the groups of pull transfers that are delayed together by disk seeks, resulting in periodic sharp drops in active transfers.

When there are several transfers with reads satisfied by the kernel's buffer cache, we would expect a proportional sharing of network bandwidth. This would allow transfers to progress independently but slowly. However, when the buffer cache cannot satisfy all of the files being transferred, we observe significant slowdowns. This suggests the buffer cache should be taken into account by the head node when planning transfers. Generally, the head node should manage this by using one-at-a-time push transfers which benefit from sequential reads or by limiting the net file sizes of concurrent transfers.

Conclusion: This workflow shows that while push transfers allow for fast distri-

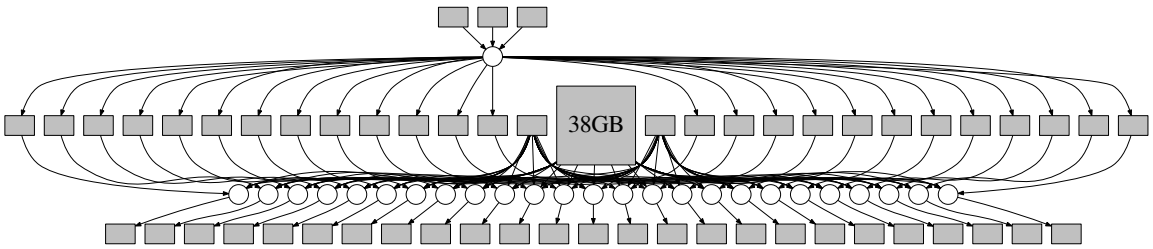


Figure 6.10. IALR Workflow

Jobs are circles and files are boxes. See text for workflow description.

bution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files. Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead using pull transfers introduces small amounts of interference but individual pull and push transfer bandwidth is mostly unaffected.

6.8 Case Study: Bioinformatics

We have evaluated the performance benefits of balanced push and pull transfers in the context of two bioinformatics workflows: iterative alignments of long reads [77] (which we shorten to IALR in this section) and Burrows-Wheeler Aligner alignment (BWA) [48] .

The IALR workflow is a simulation of a new method that iteratively compares PacBio reads to improve a target genome using locality sensitive hashing for fast updates. The workflow is composed of 26 jobs and, like BWA, begins by splitting a genome database of 255MB size into 25 parts. This workflow also has a shared 38GB database of PacBio reads which is required by all 25 jobs performing comparisons because reads that do not align at the start might at completion once two reference sequences are joined and/or updated during execution. The workflow is visualized in

Figure 6.10.

The BWA workflow aligns a number of smaller fragments, or reads, to a reference genome. It is composed of 826 jobs, beginning with a 274 way split of two query read databases each 3.1GB in size. A 265MB reference database is also shared by all jobs. The biological purpose of this specific alignment workflow was to uncover differences in sequenced individual mosquitoes such as single nucleotide polymorphisms (SNPs) relative to reference genomes. Full details are in [28]. The workflow is visualized in Figure 6.11.

Table 6.1 summarizes the results of running both experiments while varying the pull threshold. The third column "Time" is the workflow execution time including any transfers. The fourth column "Total Transfer Time" is the cumulative time in seconds for all transfers (push or pull) not including setup overhead or latency. Figures 6.12 and 6.13 show the detailed graphs of transfers as with the previous experiments.

As expected, the IALR workflow benefits greatly from centrally managed push transfers for the 32GB dependency. This enabled a spanning tree distribution across all the nodes and eliminates node instability, indicated by Figure 6.12b for "All Push". The net result is a reduction in the workflow time-to-complete by 48% from all pulls and an order of magnitude reduction in time spent doing transfers. Because the other transfers have such a small impact compared to the 32GB dependency, there is no significant difference between 256MB pull threshold and all pushes.

The BWA workflow processes several hundred multi-megabyte files which must be distributed across the cluster for parallel execution. The majority of these files are produced from splits of the two 3.1GB queries by the first job. In this situation, using only push transfers leads to significant slowdowns as push transfers are executed serially on the storage node which performed the split (due to its single transfer slot).

TABLE 6.1

BIOINFORMATICS WORKFLOWS

Experiment	Transfer Method	Time (hh:mm:ss)	Total Transfer Time (s)	Pushes	Pulls
IALR	Push All	01:52:00	8891.75	97	0
	Auto; PT=256MB	01:52:13	8976.17	24	74
	Pull All	03:38:16	163802.00	0	96
BWA	Push All	03:33:05	364.34	1946	0
	Auto; PT=256MB	00:49:04	44713.10	25	5991
	Pull All	01:11:19	77975.30	0	6244

So while the all push configuration minimized the number of transfers (1964 pushes) and the total time executing transfers (364.34s), the serial execution of push transfers and overheads introduced by centralized management severely impacts performance.

On the other hand, the use of pulls resulted in a significant improvement in workflow time, despite pulls being less efficient (77975s spent doing transfers vs. 364s) and suffer poor bandwidth (Figure 6.13a for "All Pull"). Limited use of push transfers on larger shared inputs results in a performance improvement of 31% over only pulls and 77% over only pushes. The 256MB pull threshold allows the efficient distribution of the 265MB shared reference genome while freeing up resources up on the storage node which split the queries.

Conclusion: These two bioinformatics workflows show that a balanced use of push and pull transfers reduces time to distribute dependencies for real workloads. Indeed, the two workflows experienced different worst-case behavior depending on the transfer method. However, a hybrid approach to managing transfers in the cluster

achieves as-good or better performance than only using a single methodology while eliminating common load instability.

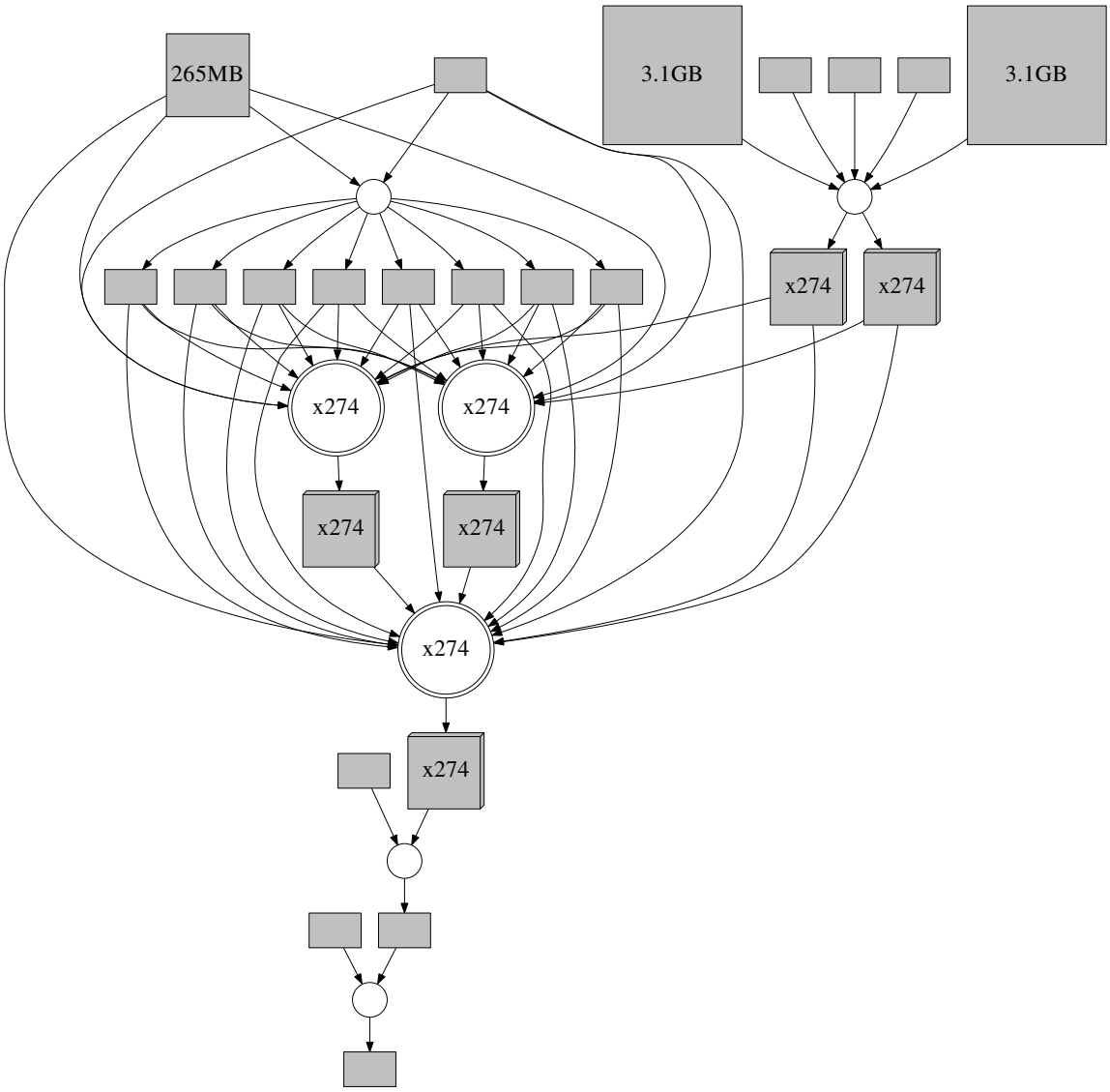
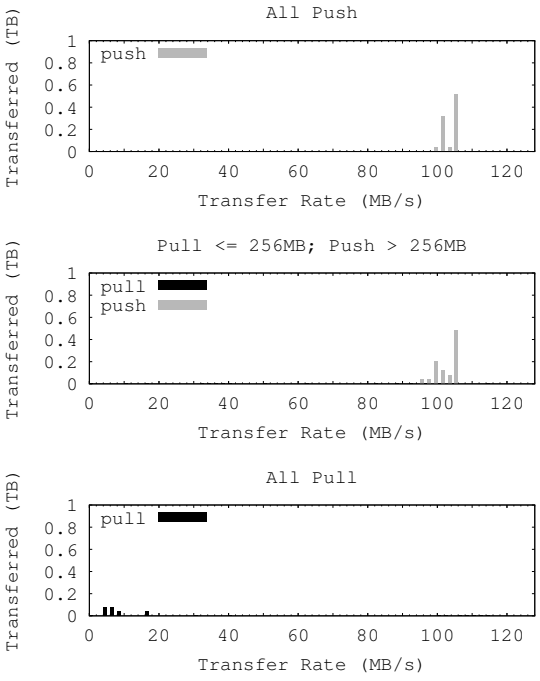
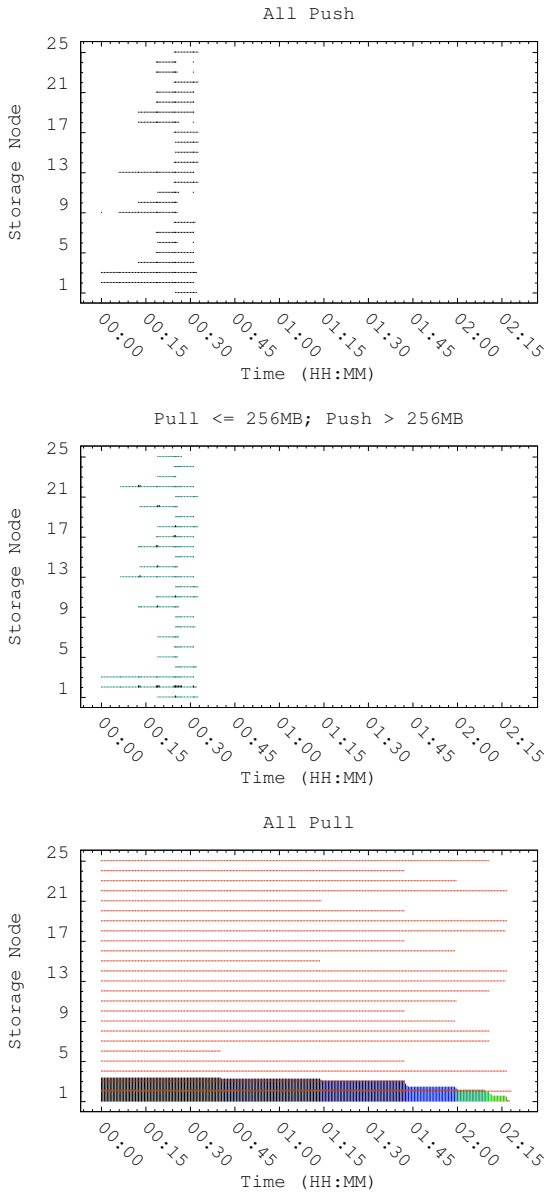


Figure 6.11. BWA Workflow

Jobs are circles and files are boxes. See text for workflow description.

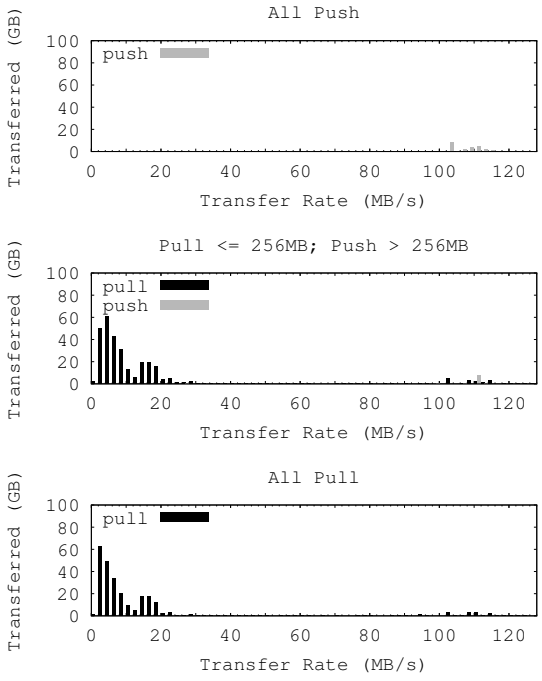


(a) Average Transfer Speed Histogram

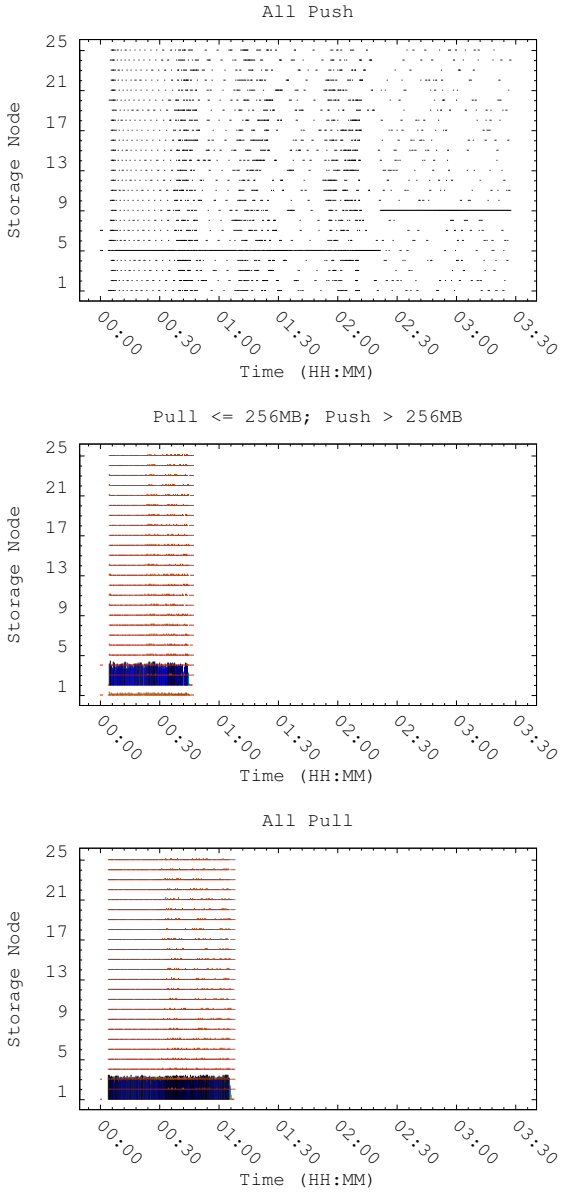


(b) Transfer Density (expl: see Figure 6.4)

Figure 6.12. IALR Workflow: Push and Pull Transfer Comparison



(a) Average Transfer Speed Histogram



(b) Transfer Density (expl: see Figure 6.4)

Figure 6.13. BWA Workflow: Push and Pull Transfer Comparison

6.9 Conclusion

This chapter explored the performance impacts of two data distribution mechanisms, push and pull, used for replicating data dependencies in scientific workflows. It was shown that controlled distribution through push transfers can eliminate typical load instability caused by large common dependencies of workflow jobs. Despite the success of push transfers, delegating transfer management to the storage node through pull transfers still has application. While push transfers allow for fast distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from pull transfers for small files.

Push transfers for smaller files do not give a justifiable improvement to distribution time when there is pressure to transfer other larger files. Instead using pull transfers introduces small amounts of interference but individual pull and push transfer bandwidth is mostly unaffected. Ultimately, a balance of the two approaches achieves optimal file distribution. This is exhibited in two bioinformatics workflows where a careful balance of the two mechanisms leads to 48% and 77% reductions over only push or pull.

The released version of Confuga defaults to a pull threshold of 256MB. Throughout our experiments, this threshold offered a good balance between the two transfer mechanisms. Still, I see this default only as a safe starting point. I expect that some workloads may do better for other thresholds. Indeed, it may be advantageous to allow workflows to specify a threshold for its set of jobs. Confuga may use that threshold or, if it is not considered sane, use its own.

Future work might explore how to parallelize the push and pull transfers for a job. Currently a job performs pull transfer prior to `exec` of the job's executable. It may be advantageous to schedule a separate transfer job which performs the pulls so that it may execute in parallel with pushes. Additionally, executing the pull transfers in a separate job allows the new replicas to become visible earlier and usable for

new transfers. I believe this mechanism would work best for smaller files which can cheaply fly under the radar of larger transfers.

CHAPTER 7

CONCLUSION

Scientific computing continues to be inundated by "big data". Sensors create terabytes of raw data, simulations are written with finer time-scales and proportionally sized outputs, etc. While scientists have been trained to deploy workflows on the Grid or a cluster, today's workflow execution frameworks have struggled to manage large data sets that workflows increasingly depend on. Some have turned to tailored solutions based on structured models like MapReduce [20] but not all workflows are easily ported to these systems.

While getting and constructing large clusters is relatively easy in today's world, the issue of getting data to the execution site (or the reverse) remains a significant problem. I have proposed an active storage cluster file system which considers data distribution a first order problem. Current distributed batch execution schedulers must confront the problem of data locality and data movement to achieve scalable and stable systems. This work has argued that sufficient structural information already exists in scientific workflows to make this possible and has demonstrated a robust solution. The ultimate result is the continued applicability of scientific workflows represented as a directed acyclic graph of tasks chained together by input and output files.

This work introduced the design of an active storage cluster file system that takes advantage of the structural information available in scientific workflows to achieve system stability and high performance. Using the workflow description, it is possible to scope the access of jobs to eliminate dynamic file access and manage transfers

between storage nodes. The cluster is able to avoid common load instability arising from "hot" data by replicating files as needed.

Connecting a traditional workflow manager to an active storage batch execution platform requires careful and explicit management of namespaces. I have shown that it is necessary to setup a mapping between the workflow namespace and the job namespace. This mapping represents a *contract* between the batch system and the workflow manager which isolates jobs from the workflow namespace. Relying on this contract allows the batch system to perform all metadata interactions before and after executing a job.

Unlike traditional distributed file systems which must support dynamic file access, the use of the namespace mapping also empowers the file system to *direct* all transfers within the cluster. This managed approach to transfers allows the cluster to avoid common load instability caused by hot replicas and to limit storage node transfers to optimize for performance. I have introduced two tuning knobs the cluster can use to achieve these goals: transfer slots and concurrent job scheduling constraints.

This work concluded with a study of using traditional *undirected* transfers to limit any negative aspects of centrally managed directed transfers. It was shown that while directed transfers allow for timely and robust distribution of large files through structured and high bandwidth transfers, there is room to tolerate some interference and hot-spots from undirected transfers for smaller files. Ultimately, a balanced approach of the two strategies allows the cluster plan the distribution of job dependencies to achieve high performance without introducing load instability.

7.1 Reflection

7.1.1 Scoping Access to Data

Traditionally, operating system kernels have used namespaces to join resources into a coherent view of the system. UNIX enabled mounting multiple file systems on a virtual file system tree. LOCUS [86] and Plan9 [61] went a step further by placing numerous process resources into this same tree. For Plan9, this enabled the distributed system to resolve access to resources based on *where* the process was.

This powerful idea of *scoping* the resources of a process is taking hold again in data-intensive computing; although, its appearance looks very different and limited. In MapReduce [20] for example, defining the file and access via a structured computation allowed powerful and specific optimizations.

On the other hand, Confuga was able to take advantage of the explicit namespace already defined in scientific workflows. While these specifications were originally used to define the files which must be moved from the submission site to the execution site, they can easily be applied as a restricted namespace which scopes access to the workflow namespace. The use of an explicit namespace for all resources allowed Confuga to perform numerous optimizations which were the study of this work. These optimizations include batched metadata operations, eliminated dynamic transfers, and planned directed transfers.

7.1.2 Joining the Batch and File Systems

As with single machine software, it is often convenient to break distributed systems into pieces with dedicated responsibilities. Early on, file systems were an easy target. This proved convenient for fast development and simpler integration but we have unfortunately lost control of critical information and control in the process. Finding, moving, and replicating data is a recurring problem in distributed sched-

ulers.

Confuga shares the perspective of other big data systems like Hadoop that effective data-locality is achieved by joining the batch system and the file system: the file system is able to respond to data demands through replication and place jobs according to data-locality. A guiding principle of this work is that the file namespace is part of the picture in scheduling decisions and must be considered to achieve locality.

7.1.3 Applicability in other Storage Systems

The optimizations Confuga performs are not completely limited to its architecture. Other distributed file systems would benefit enforcing an explicit namespace with workflow consistency semantics. For example, the Ceph [88] distributed file system could benefit from batching metadata operations despite the distributed metadata server configurations it operates with [1]. Enforcing *read-on-exec* and *write-on-exit* consistency semantics would still eliminate most metadata access which is a persistent problem for POSIX-compatible distributed file systems.

Managing locality of multiple dependencies remains a challenging problem for existing distributed file systems but it is solvable. The preferred mechanism for resolving concurrent access to large files is through reactive caching [4, 19]. However, I would suggest it is also possible to manage this cache in a more controlled way by allowing a scheduler direct control over transfers within the cluster. The benefits of directed transfers are clear when there are large dependencies required by jobs.

[1] Although, currently Ceph suggests limiting metadata servers to one due to certain bugs.

BIBLIOGRAPHY

1. Acharya, Anurag and Uysal, Mustafa and Saltz, Joel. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0. doi: 10.1145/291069.291026. URL `http://doi.acm.org/10.1145/291069.291026`.

2. Alam, Sadaf R. and El-Harake, Hussein N. and Howard, Kristopher and Stringfellow, Neil and Verzelloni, Fabio. Parallel I/O and the Metadata Wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 13–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1103-8. doi: 10.1145/2159352.2159356. URL `http://doi.acm.org/10.1145/2159352.2159356`.

3. Albrecht, Michael and Donnelly, Patrick and Bui, Peter and Thain, Douglas. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1876-1. doi: 10.1145/2443416.2443417. URL `http://doi.acm.org/10.1145/2443416.2443417`.

4. Annapureddy, Siddhartha and Freedman, Michael J. and Mazières, David. Shark: Scaling File Servers via Cooperative Caching. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251203.1251213`.

5. Armbrust, Michael and Fox, Armando and Griffith, Rean and Joseph, Anthony D. and Katz, Randy and Konwinski, Andy and Lee, Gunho and Patterson, David and Rabkin, Ariel and Stoica, Ion and Zaharia, Matei. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, Apr. 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL `http://doi.acm.org/10.1145/1721654.1721672`.

6. Baker, Mary G. and Hartman, John H. and Kupfer, Michael D. and Shirriff, Ken W. and Ousterhout, John K. Measurements of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 198–212, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: 10.1145/121132.121164. URL `http://doi.acm.org/10.1145/121132.121164`.

7. Bell, Gordon and Hey, Tony and Szalay, Alex. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

8. Bent, John and Thain, Douglas and Arpaci-Dusseau, Andrea C and Arpaci-Dusseau, Remzi H and Livny, Miron. Explicit Control in the Batch-Aware Distributed File System. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, volume 4 of *NSDI'04*, pages 365–378, Berkeley, CA, USA, 2004. USENIX Association.

9. Berners-Lee, T. and Masinter, L. and McCahill, M. Uniform Resource Locators (URL), 1994. URL `https://www.ietf.org/rfc/rfc1738.txt`.

10. Bill Allcock and Joe Bester and John Bresnahan and Ann L. Chervenak and Ian Foster and Carl Kesselman and Sam Meder and Veronika Nefedova and Darcy Quesnel and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749 – 771, 2002. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/S0167-8191(02)00094-7. URL `http://www.sciencedirect.com/science/article/pii/S0167819102000947`.

11. Braam, Peter J and Zahir, R. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

12. Brightwell, Ron and Fisk, Lee Ann. Scalable Parallel Application Launch on Cplant. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 40–40, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X. doi: 10.1145/582034.582074. URL `http://doi.acm.org/10.1145/582034.582074`.

13. Bui, Hoang and Wright, Diane and Helm, Clarence and Witty, Rachel and Flynn, Patrick and Thain, Douglas. Towards Long Term Data Quality in a Large Scale Biometrics Experiment. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 565–572, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851559. URL `http://doi.acm.org/10.1145/1851476.1851559`.

14. Bui, Peter and Rajan, Dinesh and Abdul-Wahid, Badi and Izaguirre, Jesus and Thain, Douglas. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

15. Bui, Peter and Yu, Li and Thrasher, Andrew and Carmichael, Rory and Lanc, Irena and Donnelly, Patrick and Thain, Douglas. Scripting distributed scientific workflows using Weaver. *Concurrency and Computation: Practice and Experience*, 24(15):1685–1707, 2012. ISSN 1532-0634. doi: 10.1002/cpe.1871. URL `http://dx.doi.org/10.1002/cpe.1871`.

16. A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and anal-

ysis of large scientific datasets . *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000. ISSN 1084-8045. doi: http://dx.doi.org/10.1006/jnca.2000.0110. URL http://www.sciencedirect.com/science/article/pii/S1084804500901103.

17. Couvares, Peter and Kosar, Tevfik and Roy, Alain and Weber, Jeff and Wenger, Kent. Workflow Management in Condor. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 357–375. Springer London, 2007. ISBN 978-1-84628-519-6. doi: 10.1007/978-1-84628-757-2_22. URL http://dx.doi.org/10.1007/978-1-84628-757-2_22.

18. Crockford, Douglas. The application/json Media Type for JavaScript Object Notation (JSON), 2006. URL https://tools.ietf.org/html/rfc4627.

19. Dahlin, Michael D. and Wang, Randolph Y. and Anderson, Thomas E. and Patterson, David A. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267638.1267657.

20. Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

21. Deelman, Ewa and Singh, Gurmeet and Su, Mei-Hui and Blythe, James and Gil, Yolanda and Kesselman, Carl and Mehta, Gaurang and Vahi, Karan and Berriman, G. Bruce and Good, John and Laity, Anastasia and Jacob, Joseph C. and Katz, Daniel S. Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems. *Scientific Programming*, 13(3):219–237, July 2005. ISSN 1058-9244. URL http://dl.acm.org/citation.cfm?id=1239649.1239653.

22. Donnelly, P. and Bui, P. and Thain, D. Attaching Cloud Storage to a Campus Grid Using Parrot, Chirp, and Hadoop. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 488–495, Nov 2010. doi: 10.1109/CloudCom.2010.74.

23. Donnelly, P. and Hazekamp, N. and Thain, D. Confuga: Scalable Data Intensive Computing for POSIX Workflows. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 392–401, May 2015. doi: 10.1109/CCGrid.2015.95.

24. Donnelly, P. and Thain, D. Fine-Grained Access Control in the Chirp Distributed File System. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 33–40, May 2012. doi: 10.1109/CCGrid.2012.128.

25. Donnelly, Patrick and Thain, Douglas. Design of an Active Storage Cluster File System for DAG Workflows. In *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, DISCS-2013, pages 37–42, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2506-6. doi: 10.1145/2534645.2534656. URL http://doi.acm.org/10.1145/2534645.2534656.

26. Felix, Evan J and Fox, Kevin and Regimbal, Kevin and Nieplocha, Jarek. Active storage processing in a parallel file system. In *Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2006.

27. Fischer, Michael J and Su, Xueyuan and Yin, Yitong. Assigning tasks for efficiency in Hadoop. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 30–39. ACM, 2010.

28. Fontaine, Michael C. et al. Extensive introgression in a malaria vector species complex revealed by phylogenomics. *Science*, 347(6217), 2015. ISSN 0036-8075. doi: 10.1126/science.1258524. URL http://science.sciencemag.org/content/early/2014/11/25/science.1258524.

29. Foster, Ian. *Network and Parallel Computing: IFIP International Conference, NPC 2005, Beijing, China, November 30 - December 3, 2005. Proceedings*, chapter Globus Toolkit Version 4: Software for Service-Oriented Systems, pages 2–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32246-7. doi: 10.1007/11577188_2. URL http://dx.doi.org/10.1007/11577188_2.

30. Foster, Ian and Kesselman, Carl and Tuecke, Steven. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, Aug. 2001. ISSN 1094-3420. doi: 10.1177/109434200101500302. URL http://dx.doi.org/10.1177/109434200101500302.

31. Frey, James and Tannenbaum, Todd and Livny, Miron and Foster, Ian and Tuecke, Steven. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246. ISSN 1573-7543. doi: 10.1023/A:1015617019423. URL http://dx.doi.org/10.1023/A:1015617019423.

32. Gentzsch, W. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36, 2001. doi: 10.1109/CCGRID.2001.923173.

33. Ghemawat, Sanjay and Gobioff, Howard and Leung, Shun-Tak. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

34. Gibson, Garth A. and Nagle, David F. and Amiri, Khalil and Chang, Fay W. and Feinberg, Eugene M. and Gobioff, Howard and Lee, Chen and Ozceri, Berend

and Riedel, Erik and Rochberg, David and Zelenka, Jim. File Server Scaling with Network-attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pages 272–284, New York, NY, USA, 1997. ACM. ISBN 0-89791-909-2. doi: 10.1145/258612.258696. URL http://doi.acm.org/10.1145/258612.258696.

35. Gray, Jim. A transaction model. In *Automata, Languages and Programming*, volume 85, pages 282–298. Springer, 1980.

36. Grochowski, Edward. Emerging trends in data storage on magnetic hard disk drives. *Datatech (September 1998), ICG Publishing*, pages 11–16, 1998.

37. Heshan Lin and Xiaosong Ma and Wuchun Feng and Samatova, N.F. Coordinating Computation and I/O in Massively Parallel Sequence Search. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):529–543, April 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.101.

38. Hoschek, Wolfgang and Jaen-Martinez, Javier and Samar, Asad and Stockinger, Heinz and Stockinger, Kurt. Data management in an international data grid project. In *Grid ComputingGRID 2000*, pages 77–90. Springer, 2000.

39. Howard, John H. and Kazar, Michael L. and Menees, Sherri G. and Nichols, David A. and Satyanarayanan, M. and Sidebotham, Robert N. and West, Michael J. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988. ISSN 0734-2071. doi: 10.1145/35037.35059. URL http://doi.acm.org/10.1145/35037.35059.

40. Ierusalimschy, Roberto and de Figueiredo, Luiz Henrique and Filho, Waldemar Celes. Lua - an Extensible Extension Language. *Softw. Pract. Exper.*, 26(6):635–652, June 1996. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199606)26:6⟨635::AID-SPE26⟩3.0.CO;2-P. URL http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

41. Isard, Michael and Budiu, Mihai and Yu, Yuan and Birrell, Andrew and Fetterly, Dennis. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005. URL http://doi.acm.org/10.1145/1272996.1273005.

42. Isard, Michael and Prabhakaran, Vijayan and Currey, Jon and Wieder, Udi and Talwar, Kunal and Goldberg, Andrew. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629601. URL http://doi.acm.org/10.1145/1629575.1629601.

43. Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, Mark Shellenbaum. The Zettabyte File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST'03, Berkeley, CA, USA. USENIX Association.

44. K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 352–358, 2002. doi: 10.1109/HPDC.2002.1029935.

45. Keeton, Kimberly and Patterson, David A. and Hellerstein, Joseph M. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, Sept. 1998. ISSN 0163-5808. doi: 10.1145/290593.290602. URL `http://doi.acm.org/10.1145/290593.290602`.

46. Kosar, T. and Livny, M. Stork: making data placement a first class citizen in the grid. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 342–349, 2004. doi: 10.1109/ICDCS.2004.1281599.

47. Krishnan, Arun. GridBLAST: a Globus-based high-throughput implementation of BLAST in a Grid computing framework. *Concurrency and Computation: Practice and Experience*, 17(13):1607–1623, 2005. ISSN 1532-0634. doi: 10.1002/cpe.906. URL `http://dx.doi.org/10.1002/cpe.906`.

48. Li, Heng and Durbin, Richard. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

49. Litzkow, M.J. and Livny, M. and Mutka, M.W. Condor - A Hunter of Idle Workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988. doi: 10.1109/DCS.1988.12507.

50. Malewicz, Grzegorz and Austern, Matthew H. and Bik, Aart J.C and Dehnert, James C. and Horn, Ilan and Leiser, Naty and Czajkowski, Grzegorz. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL `http://doi.acm.org/10.1145/1807167.1807184`.

51. Matsunaga, A. and Tsugawa, M. and Fortes, J. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 222–229, Dec 2008. doi: 10.1109/eScience.2008.62.

52. Michael Wilde and Mihael Hategan and Justin M. Wozniak and Ben Clifford and Daniel S. Katz and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633 – 652, 2011. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2011.05.005. URL `http://www.sciencedirect.com/science/article/pii/S0167819111000524`.

53. Moretti, C. and Hoang Bui and Hollingsworth, K. and Rich, B. and Flynn, P. and Thain, D. All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. *Parallel and Distributed Systems, IEEE Transactions on*, 21(1):33–46, Jan 2010. ISSN 1045-9219. doi: 10.1109/TPDS.2009.49.

54. Nagle, David and Serenyi, Denis and Matthews, Abbie. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 53–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3. doi: 10.1109/SC.2004.57. URL `http://dx.doi.org/10.1109/SC.2004.57`.

55. O'Sullivan, Bryan. Distributed revision control with Mercurial. *Mercurial project*, 2007.

56. P. Carns and S. Lang and R. Ross and M. Vilayannur and J. Kunkel and T. Ludwig. Small-file access in parallel file systems. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, May 2009. doi: 10.1109/IPDPS.2009.5161029.

57. P. Troger and H. Rajic and A. Haas and P. Domagalski. Standardization of an API for Distributed Resource Management Systems. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 619–626, May 2007. doi: 10.1109/CCGRID.2007.109.

58. Pamela Delgado and Florin Dinu and Anne-Marie Kermarrec and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL `https://www.usenix.org/conference/atc15/technical-session/presentation/delgado`.

59. Peter Kunszt and Erwin Laure and Heinz Stockinger and Kurt Stockinger. File-based replica management . *Future Generation Computer Systems*, 21(1): 115 – 123, 2005. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2004.09.017. URL `http://www.sciencedirect.com/science/article/pii/S0167739X04001487`.

60. Piernas, Juan and Nieplocha, Jarek and Felix, Evan J. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362660. URL `http://doi.acm.org/10.1145/1362622.1362660`.

61. Pike, Rob and Presotto, Dave and Thompson, Ken and Trickey, Howard and Winterbottom, Phil. The Use of Name Spaces in Plan 9. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, EW 5, pages 1–5, New York, NY, USA,

1992. ACM. doi: 10.1145/506378.506413. URL http://doi.acm.org/10.1145/506378.506413.

62. Quinlan, Sean and Dorward, Sean. Venti: A New Approach to Archival Storage. In *FAST*, volume 2, pages 89–101, 2002.

63. R. Buyya and D. Abramson and J. Giddy. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 283–289 vol.1, May 2000. doi: 10.1109/HPC.2000.846563.

64. Raicu, Ioan and Foster, Ian T. and Zhao, Yong and Little, Philip and Moretti, Christopher M. and Chaudhary, Amitabh and Thain, Douglas. The Quest for Scalable Support of Data-intensive Workloads in Distributed Systems. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 207–216, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: 10.1145/1551609.1551642. URL http://doi.acm.org/10.1145/1551609.1551642.

65. Raicu, Ioan and Zhao, Yong and Foster, Ian T. and Szalay, Alex. Accelerating Large-scale Data Exploration Through Data Diffusion. In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, pages 9–18, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-154-5. doi: 10.1145/1383519.1383521. URL http://doi.acm.org/10.1145/1383519.1383521.

66. Riedel, Erik and Gibson, Garth A. and Faloutsos, Christos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1-55860-566-5. URL http://dl.acm.org/citation.cfm?id=645924.671345.

67. Roselli, Drew S and Lorch, Jacob R and Anderson, Thomas E and others. A Comparison of File System Workloads. In *USENIX annual technical conference, general track*, pages 41–54, 2000.

68. Rosenblum, Mendel and Ousterhout, John K. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992. ISSN 0734-2071. doi: 10.1145/146941.146943. URL http://doi.acm.org/10.1145/146941.146943.

69. Ross, Robert B and Thakur, Rajeev and others. PVFS: A parallel file system for Linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000.

70. Rumble, Stephen M and Lacroute, Phil and Dalca, Adrian V and Fiume, Marc and Sidow, Arend and Brudno, Michael. SHRiMP: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

71. S. Jha and H. Kaiser and A. Merzky and O. Weidner. Grid Interoperability at the Application Level Using SAGA. In *e-Science and Grid Computing, IEEE International Conference on*, pages 584–591, Dec 2007. doi: 10.1109/E-SCIENCE.2007.39.

72. S. Vazhkudai and S. Tuecke and I. Foster. Replica selection in the Globus Data Grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 106–113, 2001. doi: 10.1109/CCGRID.2001.923182.

73. Sandberg, Russel and Goldberg, David and Kleiman, Steve and Walsh, Dan and Lyon, Bob. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

74. Schmuck, Frank and Haskin, Roger. GPFS: A Shared-disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 16–16, Berkeley, CA, USA, 2002. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1973333.1973349`.

75. Seung Woo Son and Lang, S. and Carns, P. and Ross, R. and Thakur, R. and Ozisikyilmaz, B. and Kumar, P. and Wei-Keng Liao and Choudhary, A. Enabling active storage on parallel I/O software stacks. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12, May 2010. doi: 10.1109/MSST.2010.5496981.

76. Shvachko, K. and Hairong Kuang and Radia, S. and Chansler, R. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010. doi: 10.1109/MSST.2010.5496972.

77. Steele, A. and S.J. Emrich. pbSandwich: Scaffolding Draft Genomes with Long Reads. In *7th International Conference on Bioinformatics and Computational Biology (BICoB)*, 2015.

78. Szeredi, Miklos. FUSE: Filesystem in Userspace, 2005. URL `http://fuse.sourceforge.net`.

79. Thain, Douglas and Livny, Miron. Parrot: Transparent User-Level Middleware for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6 (3):9–18, 2005.

80. Thain, Douglas and Moretti, Christopher. Efficient Access to Many Small Files in a Filesystem for Grid Computing. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 243–250, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1559-5. doi: 10.1109/GRID.2007.4354139. URL `http://dx.doi.org/10.1109/GRID.2007.4354139`.

81. Thain, Douglas and Moretti, Christopher and Hemmes, Jeffrey. Chirp: a practical global filesystem for cluster and Grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009. ISSN 1570-7873. doi: 10.1007/s10723-008-9100-5. URL http://dx.doi.org/10.1007/s10723-008-9100-5.

82. Thain, Douglas and Tannenbaum, Todd and Livny, Miron. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005. ISSN 1532-0634. doi: 10.1002/cpe.938. URL http://dx.doi.org/10.1002/cpe.938.

83. Thereska, Eno and Ballani, Hitesh and O'Shea, Greg and Karagiannis, Thomas and Rowstron, Antony and Talpey, Tom and Black, Richard and Zhu, Timothy. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522723. URL http://doi.acm.org/10.1145/2517349.2522723.

84. Torvalds, Linus and Hamano, Junio. Git: Fast version control system. *URL http://git-scm. com*, 2010.

85. Ungureanu, Cristian and Atkin, Benjamin and Aranya, Akshat and Gokhale, Salil and Rago, Stephen and Calkowski, Grzegorz and Dubnicki, Cezary and Bohra, Aniruddha. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST*, pages 225–238, 2010.

86. Walker, Bruce and Popek, Gerald and English, Robert and Kline, Charles and Thiel, Greg. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 49–70, New York, NY, USA, 1983. ACM. ISBN 0-89791-115-6. doi: 10.1145/800217.806615. URL http://doi.acm.org/10.1145/800217.806615.

87. Weber, Ralph O. Information technology-SCSI object-based storage device commands (OSD). *Technical Council Proposal Document T*, 10:92, 2004.

88. Weil, Sage A. and Brandt, Scott A. and Miller, Ethan L. and Long, Darrell D. E. and Maltzahn, Carlos. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485.

89. Weil, Sage A. and Brandt, Scott A. and Miller, Ethan L. and Maltzahn, Carlos. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188582. URL http://doi.acm.org/10.1145/1188455.1188582.

90. Welch, Brent. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

91. White, Tom. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

92. Woodard, Anna and Wolf, Matthias and Mueller, Charles and Valls, Nil and Tovar, Ben and Donnelly, Patrick and Ivie, Peter and Anampa, Kenyi Hurtado and Brenner, Paul and Thain, Douglas and Lannon, Kevin and Hildreth, Michael. Scaling Data Intensive Physics Applications to 10k Cores on Non-dedicated Clusters with Lobster. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 322–331, Sept 2015. doi: 10.1109/CLUSTER.2015.53.

93. Yu, Li and Moretti, Christopher and Emrich, Scott and Judd, Kenneth and Thain, Douglas. Harnessing Parallelism in Multicore Clusters with the All-pairs and Wavefront Abstractions. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: 10.1145/1551609.1551613. URL `http://doi.acm.org/10.1145/1551609.1551613`.

94. Zaharia, Matei and Borthakur, Dhruba and Sen Sarma, Joydeep and Elmeleegy, Khaled and Shenker, Scott and Stoica, Ion. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755940. URL `http://doi.acm.org/10.1145/1755913.1755940`.

95. Zaharia, Matei and Chowdhury, Mosharaf and Franklin, Michael J and Shenker, Scott and Stoica, Ion. Spark: Cluster Computing With Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

This document was prepared & typeset with LaTeX 2$_\varepsilon$*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2013[2013/04/16]) provided by Sameer Vijay and updated by Megan Patnott.*